

AD-A238 048



RL-TR-91-76, Vol III (of four)
Final Technical Report
June 1991



2

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES Technical Reports

Stanford University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 5291

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

91-04337

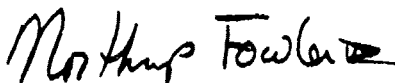


91 7 8 024

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-76, Volume III (of four) has been reviewed and is approved for publication.

APPROVED:



NORTHRUP FOWLER III
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



RONALD RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(COE) Griffiss AFB, NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES,
Technical Reports

Edward A. Feigenbaum
Robert Engelmere
H. Penny Nil
James P. Rice

Contractor: Stanford University
Contract Number: F30602-85-C-0012
Effective Date of Contract: 14 March 1985
Contract Expiration Date: 31 March 1990
Short Title of Work: Concurrent Expert Systems
Architecture
Program Code Number: OE20
Period of Work Covered: Mar 85 - Mar 90
Principal Investigator: Edward A. Feigenbaum
Phone: (415) 723-4878
RL Project Engineer: Northrup Fowler III
Phone: (315) 330-7794

Accession For	
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<input type="checkbox"/>
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Northrup Fowler III, RL (COE), Griffiss AFB NY 13441-5700 under Contract F30602-85-C-0012.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204 Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE June 1991		3. REPORT TYPE AND DATES COVERED Final Mar 85 - Oct 90	
4. TITLE AND SUBTITLE EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES, Technical Reports				5. FUNDING NUMBERS C - F30602-85-C-0012 PE - 62301E PR - E291 TA - 00 WU - 01	
6. AUTHOR(S) Edward A. Feigenbaum, Robert Engelmores, H. Penny Nii and James P. Rice					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Knowledge Systems Laboratory Stanford University 701 Welch Rd, Bldg C Palo Alto CA 94304				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington VA 22209-2308				10. SPONSORING/MONITORING AGENCY REPORT NUMBER Rome Laboratory (COE) Griffiss AFB NY 13441-5700 RL-TR-91-76, Vol III (of four)	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Northrup Fowler III/COE/(315) 330-7794					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This final report documents the results of a five-year investigation of methods for achieving higher performance for knowledge-based systems through the design of innovative software and hardware systems architectures. Volume I summarizes the work performed and lessons learned, and serves as an annotated index to the set of over 50 project technical reports. Volumes II through IV contain the project technical reports. NOTE: Rome Laboratory/RL (formerly Rome Air Development Center/RADC)					
14. SUBJECT TERMS Multiprocessor Architectures, Artificial Intelligence, Blackboard Systems				15. NUMBER OF PAGES 470	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UL	

Table of Contents for Volume 3

[Hailperin 88]	Load Balancing for Massively Parallel Soft-Real-Time Systems.	3-1
[Maegawa 90]	The Parallel Solution of Classification Problems.	3-20
[Muliawan 89]	Performance Evaluation of a Parallel Knowledge-Based System.	3-48
[Nakano 87]	Experiments with a Knowledge-Based System on a Multiprocessor.	3-149
[Nii 86]	CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving.	3-196
[Nii 88a]	Frameworks for Concurrent Problem Solving: A Report on CAGE and Poligon.	3-205
[Nii 88b]	Experiments on Cage and Poligon: Measuring the performance of Parallel Blackboard Systems.	3-233
[Nii 89]	Signal Understanding and Problem Solving: A Concurrent Approach to Soft Real-Time Systems.	3-298
[Noble 88a]	AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor.	3-309
[Noble 88b]	ELMA Programmer's Guide.	3-409
[Okuno 87]	Parallel Execution of OPS5 in QLISP.	3-443

**Load Balancing for Massively-Parallel
Soft-Real-Time Systems**

Max Hailperin

**Department of Computer Science
Stanford University
Stanford, CA 94305**

To appear in condensed form in:

Frontier '88: The Second Symposium on the Frontiers of Massively Parallel Computation

Load Balancing for Massively-Parallel Soft-Real-Time Systems

Max Hailperin*
Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, CA 94305

August 30, 1988

Abstract

Global load balancing, if practical, would allow the effective use of massively-parallel ensemble architectures for large soft-real-time problems. The challenge is to replace quick global communications, which is impractical in a massively-parallel system, with statistical techniques. In this vein, we propose a novel approach to decentralized load balancing based on statistical time-series analysis. Each site estimates the system-wide average load using information about past loads of individual sites and attempts to equal that average. This estimation process is practical because the soft-real-time systems we are interested in naturally exhibit loads that are periodic, in a statistical sense akin to seasonality in econometrics. We show how this load-characterization technique can be the foundation for a load-balancing system in an architecture employing cut-through routing and an efficient multicast protocol.

*To appear in condensed form in *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation*. This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation. This work was also supported by DARPA Contracts F30602-85-C-0012 and MDA903-83-C-0335, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and Digital Equipment Corporation.

1 Introduction

Our research group, the Stanford Knowledge Systems Laboratory Advanced Architectures Project, is exploring the construction of massively-parallel, object-oriented, knowledge-based, soft-real-time signal-interpretation systems. It seemed clear early on that some sort of adaptive load-distribution scheme would be necessary to allocate resources to such dynamic systems. Otherwise, in order to assure acceptable real-time performance, the system could only be lightly loaded, and the large-scale signal-interpretation problems the massive parallelism was intended to allow would not be possible. The remainder of this section explains why we desire a scheme which globally balances loads by migrating objects, and how we can exploit the somewhat periodic nature of our systems' loads to do global balancing in a manner appropriate to thousands of processing elements.

Much discussion in the load-distribution literature recently has centered on the choice of load balancing *vs.* load sharing [14]. While load balancing strives to keep all sites equally loaded, load sharing merely tries to prevent unnecessary idleness. Load balancing is appropriate to object-oriented real-time systems because

- real-time systems need to prevent long waits for processing; load balancing, by reducing the variance as well as the average of waiting times better achieves this; also,
- migrating objects to balance current load tends to also balance the future arrival of additional work at sites.

Traditionally, decentralized adaptive load-balancing systems have been local: they balance loads in small neighborhoods (the neighborhoods may be logical rather than physical), and rely on repeated local adjustments to achieve global balance. (For a clear example, see the descriptions of diffusion in [12,13].) We find this inappropriate to our circumstances because

- modern interconnection networks employing cut through or wormhole routing reduce the importance of locality [7],
- local techniques can fall prey to oscillation and wave-front-like propagation in the face of non-ideal conditions, and
- local techniques have difficulty responding quickly enough for dynamic and time-critical systems.

A global load-balancing system must somehow allow each site to estimate the current (or near-future) system-wide total load, in order that it may acquire or jettison sufficient work to bring its own load to the system-wide average. This seems incompatible with the constraints of a massively-parallel system: a site in a massively-parallel system must wait a considerable time to acquire global knowledge.

This apparent contradiction can be reconciled by using a stochastic time-series model to use prior load information to predict current loads. However, this approach is useless in most computer systems, as their loads are not very predictable.

Luckily, the real-time systems we are interested in (and many others) exhibit a different behavior. Their loads are periodic—not rigidly so, but rather in the same loose, statistical sense as many economic variables are seasonal. This periodicity is induced by sampled or scanned inputs and by sample-to-sample or scan-to-scan consistency in the outside world. Periodicity makes the loads more predictable, at least for lead times not greater than the period. As the period is generally relatively long, each site can have complete knowledge of loads at least through one period ago. This allows reasonably accurate prediction of current (or near-future) system-wide loads.

Notice that the statistical nature of this approach makes it appropriate to massively-parallel systems with thousands of processing elements:

- The large number of sites makes more straightforward methods employing global communications impractical.
- On the other hand, the large number of sites is necessary to make the statistical methods valid.

We are not suggesting this approach for real-time systems which are rigidly periodic; more direct use can be made of their periodicity. For example, Yan's "post-game analysis" method [17] could be used to successively refine a quasi-static mapping.

2 An Example Time Series

In this section we examine the evolution over time of the system-wide load in one of our real-time systems—an aircraft tracking and classification system [16]. We show that a simple stochastic model reasonably approximates this time series, that it is consistent with a common-sense understanding of the

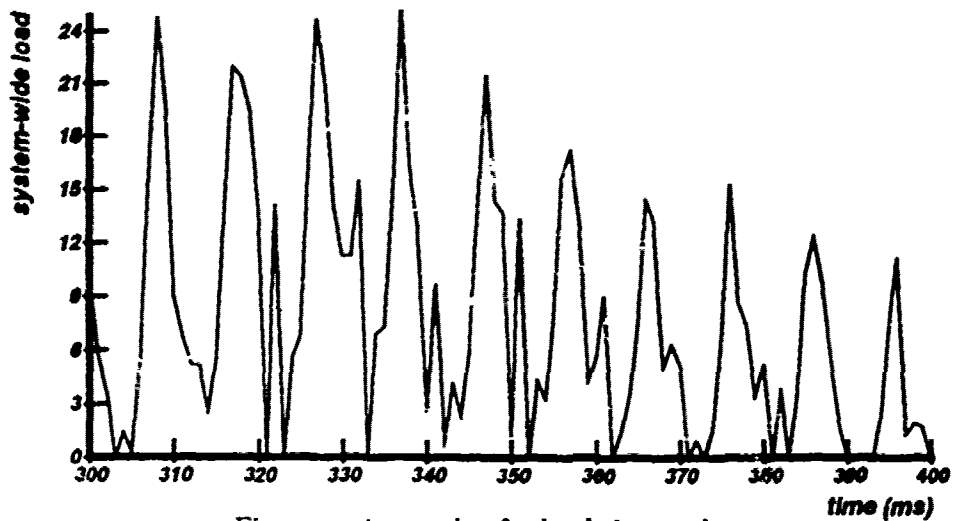


Figure 1: A sample of a load time series.

system, and that it allows moderately accurate prediction without recent complete information. Two notes are in order:

- Only the earliest, simplest, most data-driven stage of the system was operational when this data was taken; this results in a more regular time series than would otherwise be the case. In particular, diagnostic tests show our model to be incomplete, in that it misses a couple of sub-periods caused by the structure of the computation. We expect the structure of a complete system to be complex enough not to show through in the load time series.
- The plots in Figures 1 and 4 below show only a typical interval out of the larger time series which was analyzed.

Figure 1 shows the load over ten periods; each period is ten time quanta long, and the load value for each quantum is an average total of task queue lengths over that quantum. Notice that the pattern gradually shifts from period to period. Also, notice that as the observed activity diminishes, the system's performance varies from not quite keeping up with the input to having a relatively long period of quiescence between cycles. It is characteristic of real-time systems that they are sized so as to perform acceptably during peak periods, even if this means idleness at other times; this allows the periodicity of the input to show through as a periodicity of the load.

The sub-periods referred to above are also visible in the graph—the coarse sampling and small excerpt obscure it somewhat, but each major peak is followed by two smaller peaks whose sizes correlate with each other and that of the major peak.

2.1 Stochastic model

We analyzed this series using the methods of Box and Jenkins [3]¹, and identified as a suitable first-cut model for it a multiplicative integrated moving average (IMA) process of orders $(0, 1, 1) \times (0, 1, 1)_{10}$. This model has the form:

$$z_t = z_{t-1} + z_{t-10} - z_{t-11} + a_t - \theta a_{t-1} - \Theta a_{t-10} + \theta \Theta a_{t-11},$$

where z_t is the system-wide load, a_t is a white-noise series, and θ and Θ are parameters. The structure of this process is more evident when written using the backwards shift operator B :

$$(1 - B)(1 - B^{10})z_t = (1 - \theta B)(1 - \Theta B^{10})a_t.$$

Adding the constraint that loads must be non-negative improves this basic model.

This model, while suggested by statistical evidence, is also plausible in terms of the mechanism of the system. The non-periodic component of the model essentially states that the load persists, except that it is subject to random perturbations. Some fraction (θ) of each random perturbation is of short-term effect only, while the remainder lasts until counteracted: this fits well with a birth-death view of processes. The periodic component of the model is identical in form, and can be similarly justified: the aircraft under observation (and thus the load pattern) remain constant except for random perturbations, some fraction $(1 - \Theta)$ of which are long-lasting entries or departures from the field of observation.

This model belongs to the broad class of stochastic processes known as ARMA (autoregressive-moving average) processes. It is interesting to ask why this particular ARMA process should be chosen—might others not fit as well? The answer is partially that this is the simplest periodic ARMA process whose periodic and non-periodic components are both:

- non-stationary (i.e., they have no fixed level).

¹The equations in this section are reproduced with minor changes in notation from [3].

- stable (i.e., they don't grow explosively), and
- homogeneous (i.e., everywhere self-similar except for level).

Naturally a higher-order process could be used, which would fit better. However, it is generally preferable to use the simplest suitable model. Another possibility would be to drop the requirement of level independence by expanding the model to include a stationary autoregressive operator, i.e. by making it ARIMA (autoregressive-integrated moving average) rather than merely IMA. It can be argued that a busier system will spawn more processes, or alternatively that a busier system will run more processes to completion. We left this component out of our model because

- in a loaded system, the activity is not proportional to the load (as additional load means additional waiting tasks, rather than additional running tasks), and
- the statistical evidence does not unambiguously suggest such a component.

Diagnostic tests, as suggested by Box and Jenkins, showed that the model was only roughly fitting, due in part to the unmodeled sub-periods. This is especially evident in the cumulative periodogram of residuals, reproduced in Figure 2: the bulge around frequency 0.25 (period 4) shows that the model misses some periodicity in that neighborhood. (A cumulative periodogram shows an integrated power spectrum. A perfectly fitting model would leave white-noise residuals with a flat power spectrum and hence a straight diagonal cumulative periodogram.) Even the cumulative periodogram of ideal white-noise residuals might, because of the limited sample size, deviate outside the dashed lines approximately 25% of the time (the limit lines are calculated from the Kolmogorov-Smirnov test). Therefore, as the bulge just reaches the 25% limit line, it can't be considered an especially serious failure of the model. On the other hand, other statistical evidence and our understanding of the system indicate that the model is genuinely incomplete, rather than the bulge merely being an artifact of the limited sample size. We felt that incorporating these sub-periods into the model would be artificial, however, both because they are an artifact of the simplicity of the sample system, and also because they are not *a priori* known (or necessarily constant), unlike the externally imposed period.

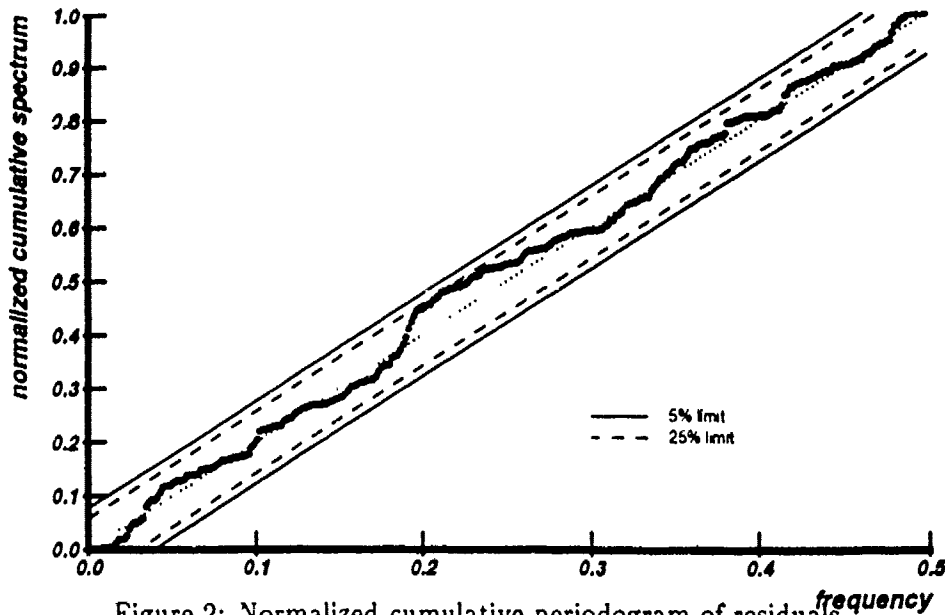


Figure 2: Normalized cumulative periodogram of residuals.

2.2 Forecasting

The non-periodic component of the model is that which is conventionally used for aperiodic computer systems; it gives rise to the familiar exponentially-weighted average forecast function. The periodic component in effect adds an exponentially-weighted average of corrections to this forecast, derived from the experience at corresponding points in earlier periods. Formally, the best one-step-ahead forecast possible for the model is found by assigning weights π_j to the loads j steps earlier, where

$$\begin{aligned} \pi_j &= \theta^{j-1}(1-\theta), \quad j = 1, \dots, 9 \\ \pi_{10} &= \theta^9(1-\theta) + (1-\Theta) \\ \pi_{11} &= \theta^{10}(1-\theta) + (1-\theta)(1-\Theta) \\ \pi_j &= \theta\pi_{j-1} + \Theta\pi_{j-10} - \theta\Theta\pi_{j-11}, \quad j \geq 12. \end{aligned}$$

Depending on the relationship between θ and Θ , the heaviest weight in the forecast may either be on the most recent value, or on the one a period ago. In the aircraft tracking case (and many others, we speculate), there is more consistency from period to period than from instant to instant (as aircraft are more inertial than processes). This leads to the weights

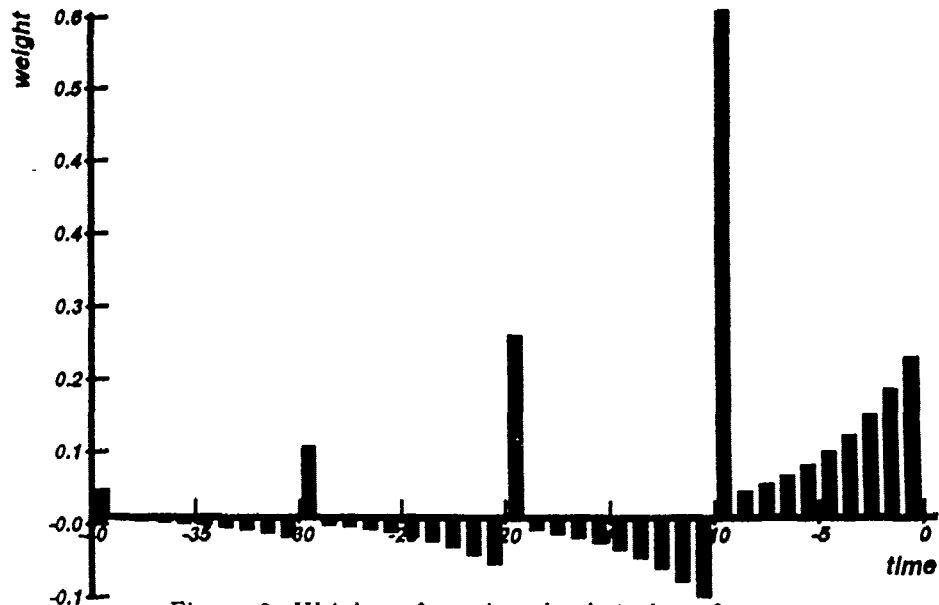


Figure 3: Weights of previous loads in best forecast.

illustrated in Figure 3, which were computed from the values for θ and Θ that best fit our sample series.

Forecasts can also be computed directly from the difference equation we used to define the model. In either case, forecasts for greater lead times can be calculated by repeated use of the step-ahead formula. (By lead time we mean the time from when the total load is last known to when the forecast is for.)

Since the period (in this case, the scan time of a radar) is long relative to the communication latencies of the system, it is reasonable to suppose that each site can have complete knowledge of all other sites' loads at least up until one period earlier, with diminishing knowledge thereafter. It should be possible in principle to make some use of the more recent, incomplete, information to improve the forecast, given a model of the load distribution with load balancing. In the next section we address this problem and show a heuristic solution. However, Figure 4 shows that even forecasts made using only data up through one period in advance are usually moderately accurate.

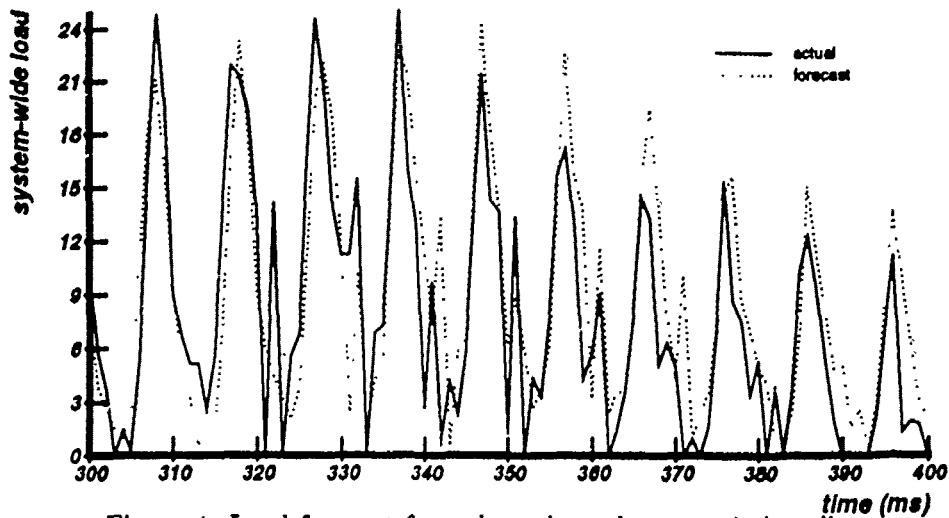


Figure 4: Load forecast from data through one period earlier.

2.3 How typical is this example?

Though this section presented a case study of a single time series taken from a single application, we believe the basic features are common to other systems as well. Preliminary results from experimentation with a passive radar interpretation system [4] confirm this belief. The IMA $(0, 1, 1) \times (0, 1, 1)_p$ model used here may well suit many such systems, though its suitability should of course be tested in each case. As well as testing the suitability of the model to a particular application, it is necessary to tune the parameters using sample time series. Systems with more than one period, for example from heterogeneous sensors, would necessitate a straightforward extension of the model.

One potential stumbling block in generalizing this technique to more realistic systems is that higher-level processing tends to be triggered by significant changes in the input (or by the lack of expected changes), rather than by the input itself. For example, a system that not merely tracks aircraft, but also attempts to deduce possible objectives, would reconsider the objective of an aircraft that sharply turned, or that failed to turn when it was expected to. This reduces the scan-to-scan consistency of the load. It remains to be seen how troublesome this is; clearly this depends on how much of the processing is special-case. When this issue came up in a discussion with a group familiar with actual systems, the consensus was that the load

on present-day systems is indeed quite periodic [15].

3 Incorporating Incomplete Information

The simple stochastic model presented in the preceding section only allows load information old enough to be complete (i.e. available from all sites) to be used. In this section we refine our model to allow incomplete information (i.e., more recent loads from some sites) to be employed. We formulate the problem, show an exact but impractical solution, and then present provably good practical heuristic approximations.

3.1 The problem

In order to understand what use a site can make of recent but incomplete information, we must refine our model to include how the system-wide total load is divided among the N sites. A simple, plausible version of this is to assume that the sites are independent instantaneously, but in the longer-term are successfully balanced. Formally, the model we have in mind is

$$z_{i,t} = a_{i,t} + \frac{z_{t-1} + z_{t-10} - z_{t-11} - \theta a_{t-1} - \Theta a_{t-10} + \theta \Theta a_{t-11}}{N},$$

where we use $z_{i,t}$ for the load of site i at time t (with $z_t = \sum_i z_{i,t}$) and similarly for $a_{i,t}$ and a_t (the $a_{i,t}$ are independently normally distributed, with variance σ_a^2).

As long as all $z_{i,t}$ are known, the $a_{i,t}$ can be calculated, and thus used for forecasting. When the information is incomplete, the deviation of the known $z_{i,t}$ from the step-ahead forecasts can no longer be attributed solely to their corresponding $a_{i,t}$, but rather will also include the persistent fraction of earlier unknown perturbations. The problem is to find the expected division between these two sources of perturbation, as the expected value of each $a_{i,t}$ should be incorporated into the forecast in its own way.

3.2 Exact solution

This problem can be solved by applying Bayes's theorem:

- We are given as a prior distribution for the $a_{i,t}$ that they are independently normally distributed with some variance σ_a^2 .

- We make observations which imply a joint likelihood for the $a_{i,t}$ that is uniform where certain linear combinations of them (given below) equal the known $z_{i,t}$ and zero elsewhere.
- We would like to find the posterior joint distribution of the $a_{i,t}$, specifically its expected value, for use in forecasting.

The non-zero regions of the likelihood function can be found by rewriting the equation for $z_{i,t}$ in terms of the $a_{i,t}$ alone, using the summation operators $S = (1 + SB)$ and $S_{10} = (1 + S_{10}B^{10})$:

$$z_{i,t} = a_{i,t} + \frac{((1 - \theta)SB + (1 - \Theta)S_{10}B^{10} + (1 - \theta)(1 - \Theta)SS_{10}B^{11})a_t}{N}$$

The posterior distribution can readily be written using Bayes's theorem, provided one is willing to leave some messy integrals in it. Unfortunately, this leaves numerical integration as the only way to find the needed expected value. This seems to be too much work to expect a load-balancing system to perform each time interval. What is needed is a pre-posterior analysis—a general analysis done in advance, into which specific numbers can be plugged at run time. Unfortunately, we know of no such approach to this problem in the general case. In the next subsection we consider heuristic approximations appropriate to our intended implementation. The analysis above serves as the standard by which the heuristics are judged, as well as suggesting them.

3.3 Heuristic approximations

The simplest heuristic is to simply assume that the full deviation of each known load $z_{i,t}$ from its step-ahead forecast is purely its corresponding $a_{i,t}$. This heuristic is actually the truth (given our model) for the first time-quantum with incomplete information, and can be shown to be a conservative approximation provided there is less than a period of incomplete information. By a conservative approximation, we mean that this heuristic is guaranteed to be more accurate than simply ignoring the incomplete information. This is because mistaking the retained portion of prior perturbations for current perturbation leads to it's being erroneously re-multiplied by $(1 - \theta)$, i.e. underestimated.

We can improve this approximation by taking advantage of one feature of our intended implementation. The implementation we suggest in section 5 uses a randomized style of information spreading known as "rumor mongering" which spreads each site's load information to an exponentially widening

fraction of the other sites. Thus the amount of load information a site has drops off exponentially with recency, and only the earliest incomplete load information is of any real significance.

In particular, for realistic parameters (e.g. a spreading factor of eight) the only significant improvement that could be made in the above simple heuristic would be to better account for the deviations observed in the second incomplete-information time-quantum. Moreover, this division between the first two incomplete-information time-quanta need not make use of information from later time-quanta, as such information would be very weak under these assumptions. This leaves a tractable two-quanta version of the general problem of the preceding subsection.

The $a_{i,t}$ from the N_n non-reporting sites of the first quantum can be lumped together, as can those from the N_r reporting sites of the second quantum. This is because of the symmetry amongst them. We will call the contribution of the former to the second-quanta deviations X and that of the latter Y . Our prior distributions for them are independent, normal, both have mean zero, and (by elementary probability theory) have the variances

$$\begin{aligned}\sigma_x^2 &= \frac{N_r^2}{N^2}(1-\theta)^2 N_n \sigma_a^2 \\ \sigma_y^2 &= N_r \sigma_a^2.\end{aligned}$$

We know that X and Y sum to the observed deviation, δ , of the second-quanta loads from their step-ahead forecasts. Therefore, the posterior distribution from Bayes's theorem gives us the following posterior expected values:

$$\begin{aligned}E(X) &= \frac{\int_{-\infty}^{\infty} x e^{-x^2/2\sigma_x^2 - (\delta-x)^2/2\sigma_y^2} dx}{\int_{-\infty}^{\infty} e^{-x^2/2\sigma_x^2 - (\delta-x)^2/2\sigma_y^2} dx} \\ &= \delta \frac{\sigma_x^2}{\sigma_x^2 + \sigma_y^2} \\ E(Y) &= \delta \frac{\sigma_y^2}{\sigma_x^2 + \sigma_y^2}.\end{aligned}$$

Thus we can readily at run time use the observed values of δ , N_n , and N_r to calculate a very good approximation to the best forecast possible with the available information.

4 Precision of Forecasts

In this section we analyze the potential for practical utility of our load-characterization scheme. We show that for the large numbers of sites characteristic of massively parallel architectures, our scheme provides load estimates which are accurate enough to be useful for load balancing.

We can use the model of section 2 to calculate probability limits of forecasts—that is, the region around the forecast in which the actual system-wide load will lie some specified fraction of the time. Additionally, the more detailed model of section 3 specifies how the individual sites' loads can be expected to be distributed about the system-wide average load. What is most interesting is combining these two, in order to determine

- what fraction of the sites can be expected to be over- or under-loaded at some significance level, and
- how much relative error can be expected in the amount of work transferred between sites, due to erroneous forecasts.

Happily, we show that the accuracy of the forecasts relative to the standard-deviation of the site loads goes up with the square-root of the number of sites, so that for massively-parallel systems the uncertainty in the forecasts is unproblematic (assuming the validity of the model).

4.1 Probability limits of forecasts

The conditional probability distribution of the system-wide load about its forecast value is simply the sum of those of the a_i not included in the forecast. The error in the forecast will thus be normally distributed with mean zero and variance increasing with lead-time. For the IMA $(0, 1, 1) \times (0, 1, 1)_p$ model, if the forecast is made using complete information only, with lead time $l < p$, the variance is

$$V(l) = (1 + (l-1)(1-\theta)^2)N\sigma_a^2.$$

We can use the above formula to calculate approximate probability limits for the forecasts by substituting an estimate for σ_a . One approach would be to estimate it using the sample standard deviation from prior runs. Prior to the introduction of load balancing, the detailed model of section 3 certainly doesn't apply, but the system-wide model of section 2 presumably does, at least approximately. Therefore, the sample variance of the system-wide load

should be used as an initial estimate for $N\sigma_a^2$, rather than starting with the sample variance of individual site loads.² If the system-wide load sample standard deviation is s , then we can estimate that with probability ϵ the actual load differs from the lead l forecast by more than

$$u_{\epsilon/2}s\sqrt{1+(l-1)(1-\theta)^2}$$

where $u_{\epsilon/2}$ is the $\epsilon/2$ -tail-area point of the unit normal distribution. Notice that these bounds are for the total load—the standard deviation, and hence probability limits, for the average load are smaller by a factor of N .

4.2 Comparison with the distribution of site loads

Our model asserts that the loads of the individual sites at any time are normally distributed about the system-wide average load with standard deviation σ_a . We can compare this with the standard deviation of the lead l conditional probability distribution of the average load, which we derived in the previous subsection. The latter is larger by a factor of

$$\frac{\sqrt{1+(l-1)(1-\theta)^2}}{\sqrt{N}}$$

the factor of \sqrt{N} results from averaging N independent deviates.

This implies that for large systems the forecasts will be accurate enough to be useful. For example, if the system of section 2 could be spread among 1024 sites, even one-period-ahead forecasts would have a factor of 27 lower standard deviation than the site loads. Thus virtually all apparent over- or under-loads would be statistically significant, and the relative error in the amount of work transferred would be small (roughly 1/27).

5 Load-balancing Mechanism

In this section we outline a load-balancing scheme employing the load-characterization methodology of the preceding sections. Our scheme relies on a "rumor mongering" style of information spreading [9], which is appropriate to our architecture. We show that the mechanism not only allows sites to assess their load with respect to the system-wide average, but also

²We only wrote the formula in terms of the per-site σ_a^2 in order to be notationally consistent with section 3.

allows overloaded sites to reliably find sufficiently underloaded sites to which objects can be migrated.

If each site stores its knowledge of all sites' load histories, then they can spread their information around by a process of "rumor mongering"—that is, by randomly sharing information [10,1,2,9]. Naturally, the histories can be compressed by discarding information old enough to be scarcely relevant and by combining together loads from all sites where they all are known. Some information may be young enough to be relevant to forecasting, but old enough to be well-known. This information can be retained but not passed on. [9] has a good discussion of such issues.

Our CARE ensemble architecture [8] uses a cut-through interconnection network, so latency is not proportional to distance (in the absence of contention). Additionally, it supports an efficient multicast protocol [5]. Therefore, we suggest that the information spreading be achieved by each site periodically multicasting its information to a random sample of the other sites. While the number of sites that each site will hear from in any given period varies, it can be shown that the distribution (a binomial distribution, rapidly approaching a Poisson distribution) is such that a paucity of information will be rare, even with a quite moderate sample size, e.g. eight.

Upon receiving a load-information message, a site should integrate the information into its own knowledge, and then use the time-series model (provided *a priori* based on experiments with the particular system) to estimate the current system-wide average load with probability limits. It should then compare this predicted average with its own current load, and with the load of the sender at the time of the sending. If the recipient appears significantly underloaded and the sender appears significantly overloaded, a request for work should be sent back.

This is a combination of random gossiping to distribute the information needed to decide whether and how much work to transfer, together with polling/bidding to match up the participating sites. As with all bidding schemes, some precautions are needed to avoid races. The underloaded site should not place any other requests for work until it receives work or an apology from the overloaded site. As the inter-arrival time for messages from overloaded sites should be high relative to the round-trip message time, few conflicts should occur.

The bidding could be reversed (overloaded sites could ask underloaded sites to accept work), but this would require that an extra message be sent. The system as we present it can best be classified as receiver-initiated [11], though in a sense the sender initiates the process by multicasting its load

information. This confusion of terminology results from our integration of the global-information-spreading and partner-seeking components of the mechanism.

It should be rare that an overloaded site cannot find enough total underload among the sites it samples to match its own overload. For example, suppose that the loads are normally distributed (as they are in the model of section 3), and that the sample size is eight. Of the eight sites sampled, it can be expected that four will be underloaded. The expected value of the absolute value of a normal deviate is $2/\sqrt{2\pi}$, or about .8 standard deviations, so the four underloaded sites will on the average have approximately 3.2 standard deviations worth of underload. But the originating site must really be far out on the tail of the distribution to have more than 3.2 standard deviations worth of overload. Notice that it is impossible to make as strong a statement in the reverse direction—this is an additional reason to favor a receiver-initiated transfer (it is more important for overloaded sites to reliably find underloaded sites than the converse).

The only aspect of load balancing not addressed by this mechanism is the choice of which objects to migrate. Here again the real-time nature of the system must be addressed. In general neither the highest- nor lowest-priority objects are best migrated, so as to neither unfairly advance a low-priority object nor hold up (due to migration time) a high-priority object. Chang addresses these issues in [6].

6 Acknowledgments

Anoop Gupta, Bruce Delagi, Harold Brown, and John Hennessy provided valuable feedback on this work. The entire Advanced Architectures Project made this work possible, by providing the technical context; many members additionally helped me with numerous specific difficulties. In this context, I'd like to specifically thank Greg Byrd, Bruce Delagi, Sayuri Nishimura, Alan Noble and Nakul Saraiya.

References

- [1] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel. Processes migrate in Charlotte. Technical Report 655, Computer Sciences Department, University of Wisconsin-Madison, August 1986.

- [2] Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software—Practice and Experience*, 15(9):901-913, September 1985.
- [3] George E. P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day Inc., 1976.
- [4] Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An experiment in knowledge-based signal understanding using parallel architectures. Technical Report STAN-CS-86-1136, Department of Computer Science, Stanford University, October 1986.
- [5] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. A dynamic, cut-through communications protocol with multicast. Technical Report STAN-CS-87-1178, Department of Computer Science, Stanford University, September 1987.
- [6] Hung-Yang Chang. Dynamic scheduling algorithms for distributed soft real-time systems. Technical Report 728, Computer Sciences Department, University of Wisconsin-Madison, 1987.
- [7] William J. Dally. Wire-efficient VLSI multiprocessor communications networks. In *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 391-415. The MIT Press, 1987.
- [8] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An instrumented architectural simulation system. In *Artificial Intelligence and Simulation: The Diversity of Applications*. The Society for Computer Simulation International, February 1988.
- [9] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1-12, August 1987.
- [10] Zvi Drezner and Amnon Barak. A probabilistic algorithm for scattering information in a multicomputer system. Technical Report CRL-TR-15-84, Computing Research Laboratory, University of Michigan, March 1984.

- [11] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53-68, March 1986.
- [12] Robert H. Halstead, Jr. and Stephen A. Ward. The MuNet: A scalable decentralized architecture for parallel computation. In *Proc. 7th Annual Symposium on Computer Architecture*, pages 139-145, May 1980.
- [13] Paul Hudak and Benjamin Goldberg. Experiments in diffused combinator reduction. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 167-176, August 1984.
- [14] Phillip Krueger and Miron Livny. Load balancing, load sharing and performance in distributed systems. Technical Report 700, Computer Sciences Department, University of Wisconsin-Madison, August 1987.
- [15] Personal communication. September 10, 1987. Discussion with members of MIT Lincoln Laboratories Machine Intelligence Group.
- [16] Russell Nakano and Masafumi Minami. Experiments with a knowledge-based system on a multiprocessor. Technical Report STAN-CS-87-1188, Department of Computer Science, Stanford University, October 1987.
- [17] Jerry C. Yan. Managing and measuring two parallel programs on a multiprocessor. Technical Report CSL-TR-87-333, Computer Systems Laboratory, Stanford University, June 1987.

The Parallel Solution of Classification Problems

by
Hirotoishi Maegawa

Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, California 94305

and

Corporate Research Laboratories
Sony Corporation
6-7-35 Kitashinagawa
Shinagawa, Tokyo 141, Japan

*This research was supported by DARPA Contract F30602-85-C-0012
and NASA Ames Contract NCC 2-220-S1.*

Abstract

We developed a problem solving framework called ConClass capable of classifying continuous real-time problems dynamically and concurrently on a distributed system. ConClass provides an efficient development environment for describing and decomposing a classification problem and synthesizing solutions. In ConClass, designed concurrency of decomposed subproblems effectively corresponds to the actual distributed computation components. This scheme is useful for designing and implementing efficient distributed processing, making it easier to anticipate and evaluate the system behavior. ConClass system has an object replication feature in order to prevent a particular object from being overloaded. An efficient execution mechanism is implemented without using schedulers or synchronization schemes liable to be bottlenecks. In order to deal with an indeterminate amount of problem data, ConClass dynamically creates object networks to justify hypothesized solutions and thus achieves a dynamic load distribution. We confirmed the efficiency of parallel distributed processing and load balancing of ConClass with an experimental application.

1. Introduction

In this paper we describe a framework for the parallel solution of classification problems. We developed a framework called ConClass (Concurrent Classification) on a distributed-memory multiprocessor system. ConClass is based on the inherent parallel characteristics of classification problems and is capable of solving real-time problems continuously and concurrently.

Classification is one of the more commonly used problem solving methods and has been used in diverse areas such as engineering, biology, and medicine. The classification problem solving model provides a high-level structure for the decomposition of problems, making it easier to recognize and represent similar problems. Classification is the act of identifying an unknown phenomenon as a member of a known class of phenomena. The process of identification is that of matching the observations of an unknown phenomenon to the features of known classes. Classification problem solving is described in detail in [Clancey 84, 85].

Most previous systems solve classification problems in series in terms of classification processes and deal with static data [Buchanan 84, Bennett 78, Rich 79, Brown 82]. Related work on classification is based on static aspects as well [Cohen 85, Bylander 86]. Recent AI research has, however, focused on real-time problem solving such as that surveyed in [Laffey 88]. For example, problems from the engineering field are dynamic in domains such as continuous signal understanding and manufacturing diagnostics.

Our motivation was to develop a framework capable of describing and solving continuous real-time classification problems in parallel. We implemented the ConClass system based on the inherent parallel nature of classification problem solving.

Another area of our research interest concerned finding out the fundamental requisites of parallel distributed classification and implementing an efficient framework based on such intrinsic features. ConClass achieved efficient parallel computation and linear speedup against the number of processing elements.

We developed ConClass on a simulated distributed-memory multiprocessor system called CARE [Deligi 87a] using a distributed processing language called LAMINA [Deligi 87b].

We implemented an experimental classification system in ConClass to evaluate the performance of ConClass. This system classifies observed aircraft by using continuous abstract radar signal data.

This research is a part of the Advanced Architectures Project at the Knowledge Systems Laboratory at Stanford University, a section of which has been dedicated for research of distributed processing [Rice 89].

In subsequent sections we describe the methodology of parallel classification problem solving, the implementation of ConClass on a distributed system, and finally, an evaluation using the experimental application.

2. Parallel Classification Methodology

2.1. Problem Decomposition

A classification problem can be structured as a directed acyclic graph whose nodes are decomposed subproblems. A classification solution of decomposed subproblem can be supplied to other subproblem solvers. A solver may synthesize other classification solutions. Propagation of problem data and solutions is hierarchically organized in this manner. Thus, classification problem solving can be organized intrinsically hierarchical and distributed.

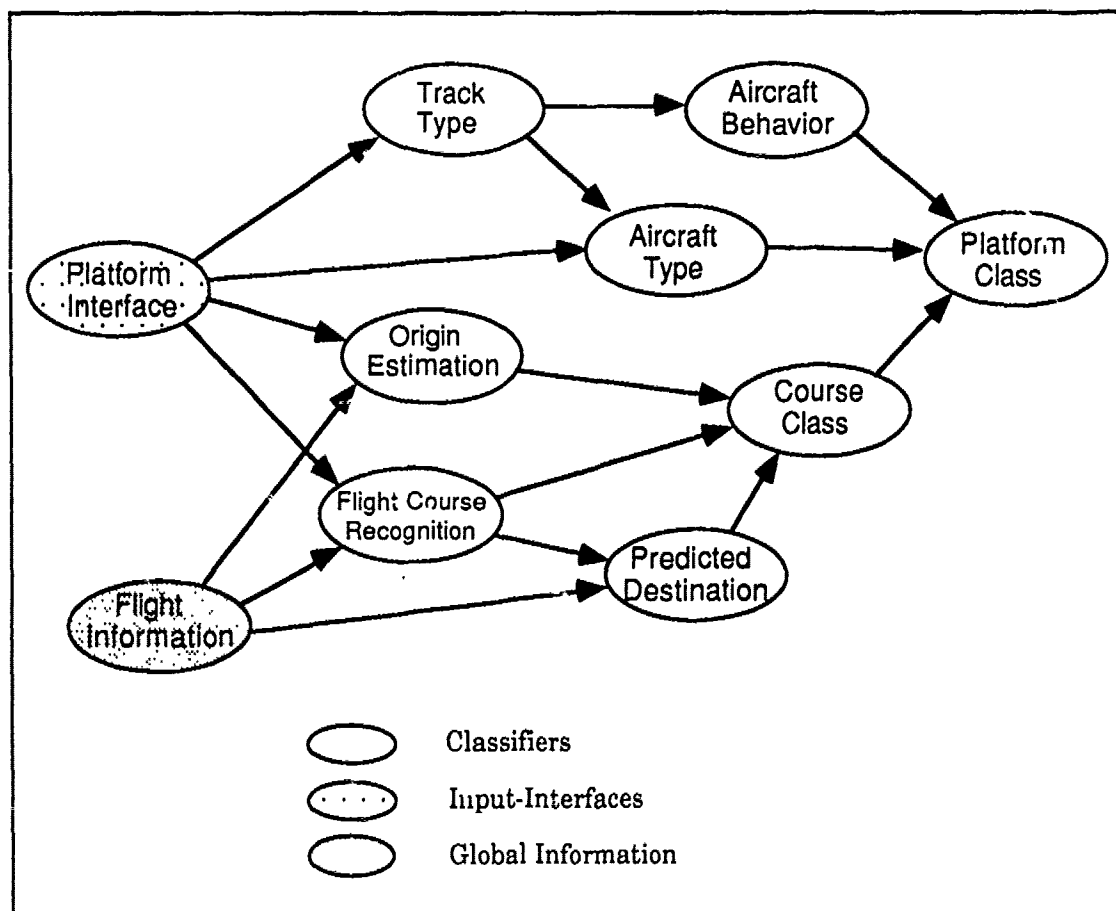


Figure 2.1. Hierarchical Classification Configuration

Figure 2.1 shows an example of a problem solving system which classifies aircraft represented by radar signals. We denote such an aircraft problem object a *platform*. This system solves classification subproblems such as aircraft type and flight course recognition and then provides final classifications such as commercial, military, or a smuggler's aircraft. Example solutions are shown in Figure 2.3. Attribute values of platforms change over time and the classification state of the system varies according to these data. The solutions may change due to global information such as flight plans and ground circumstances as well. Classification system can accept different kinds of problem inputs such as aircraft location and velocity, observed maneuverability, and radar signature.

Thus, classification problem solving has characteristics suitable for parallel processing using decomposed subproblem solvers especially for continuous dynamic problems. We implemented ConClass using such inherent characteristics.

The parallel processing in ConClass is designed using decomposed classification subproblem solvers. The ConClass system represents those problem solvers as parallel processing elements and allocates them directly on computational hardware components. This decomposition scheme makes designed concurrency effectively correspond to actual parallel computation. The scheme makes it easier to anticipate and evaluate the system behavior for obtaining efficient concurrency. We call a decomposed problem solver a *classifier*. We call the problem solving network consisting of the classifiers *the classifier network*. A classifier whose classification is derived by other classifiers is denoted a super-classifier of those classifiers. A classifier whose solutions is synthesized in other classifiers is called a sub-classifier of those classifiers. Classifiers can act concurrently and dynamically when problem data and solutions are propagated over time.

Problems in ConClass may be created dynamically such as an aircraft platform captured by a radar system. In addition, ConClass is capable of manipulating multiple sources of a continuous problem. ConClass has an object which links problem objects to the entrance classifiers. We call the linking object *an interface-object*. When a problem is created dynamically, the problem object receives references to the entrance classifiers from an interface-object and starts sending them problem data.

A classifier has known classes of phenomena into which problem objects are classified as solutions. We call such a class a *classification-category*. If a classifier succeeds in classifying, it sends the solution to its abstract classifiers, that is, super-classifiers. The abstract classifiers classify the problem by using the solution and propagate their classification solutions in the same manner. Each classification computation is a decomposed subproblem solving mentioned above. The ConClass classification system may have more than one of the most abstract classifiers to obtain different kinds of solutions to the entire classification problem.

2.2. Load Distribution

The research goal of ConClass is to implement efficient distributed processing as well as to develop a framework for describing and decomposing classification problems. In order to distribute decomposed problem solvings, we have two schemes: replication of objects and dynamic distribution of problem solving tasks.

A classifier acts when it receives a solution from its sub-classifier and when the sub-classifier changes the solution. Therefore, classifiers that are lower in the hierarchy usually execute a larger amount of classifications than those higher in the hierarchy. Even in the same level of the hierarchy, classification computations may differ between the classifiers. In order to achieve efficient load balancing to such objects which can have varying

Figure 2.1 shows an example of a problem solving system which classifies aircraft represented by radar signals. We denote such an aircraft problem object a *platform*. This system solves classification subproblems such as aircraft type and flight course recognition and then provides final classifications such as commercial, military, or a smuggler's aircraft. Example solutions are shown in Figure 3. Attribute values of platforms change over time and the classification state of the system varies according to these data. The solutions may change due to global information such as flight plans and ground circumstances as well. Classification system can accept different kinds of problem inputs such as aircraft location and velocity, observed maneuverability, and radar signature.

Thus, classification problem solving has characteristics suitable for parallel processing using decomposed subproblem solvers especially for continuous dynamic problems. We implemented ConClass using such inherent characteristics.

The parallel processing in ConClass is designed using decomposed classification subproblem solvers. The ConClass system represents those problem solvers as parallel processing elements and allocates them directly on computational hardware components. This decomposition scheme makes designed concurrency effectively correspond to actual parallel computation. The scheme makes it easier to anticipate and evaluate the system behavior for obtaining efficient concurrency. We call a decomposed problem solver a *classifier*. We call the problem solving network consisting of the classifiers the *classifier network*. A classifier whose classification is derived by other classifiers is denoted a super-classifier of those classifiers. A classifier whose solutions is synthesized in other classifiers is called a sub-classifier of those classifiers. Classifiers can act concurrently and dynamically when problem data and solutions are propagated over time.

Problems in ConClass may be created dynamically such as an aircraft platform captured by a radar system. In addition, ConClass is capable of manipulating multiple sources of a continuous problem. ConClass has an object which links problem objects to the entrance classifiers. We call the linking object an *interface-object*. When a problem is created dynamically, the problem object receives references to the entrance classifiers from an interface-object and starts sending them problem data.

A classifier has known classes of phenomena into which problem objects are classified as solutions. We call such a class a *classification-category*. If a classifier succeeds in classifying, it sends the solution to its abstract classifiers, that is, super-classifiers. The abstract classifiers classify the problem by using the solution and propagate their classification solutions in the same manner. Each classification computation is a decomposed subproblem solving mentioned above. The ConClass classification system may have more than one of the most abstract classifiers to obtain different kinds of solutions to the entire classification problem.

2.2. Load Distribution

The research goal of ConClass is to implement efficient distributed processing as well as to develop a framework for describing and decomposing classification problems. In order to distribute decomposed problem solvings, we have two schemes: replication of objects and dynamic distribution of problem solving tasks.

A classifier acts when it receives a solution from its sub-classifier and when the sub-classifier changes the solution. Therefore, classifiers that are lower in the hierarchy usually execute a larger amount of classifications than those higher in the hierarchy. Even in the same level of the hierarchy, classification computations may differ between the classifiers. In order to achieve efficient load balancing to such objects which can have varying

instantiates the classification-category corresponding to the hypothesis as a newly created computation object. This production scheme of an initial hypothesis for a given problem on the classifier network is the initial hypothesis formation as shown in Figure 2.2. We call an instantiated object of a classification-category a *classified-instance*.

Dynamic Hypothesis Maintenance:

One of the most abstract classifiers, which forms a hypothesis, makes its sub-classifiers instantiate the classification-categories which derive this hypothesis. These classifiers propagate the instantiation of classification-categories to their sub-classifiers in the same manner. When the concrete classifiers make classified-instances, the problem object which receives the hypothesis obtains links to those classified-instances. The reason for creating classified-instances backwards in this method is so that only required classified-instances are instantiated. The set of created classified-instances is an instance network.

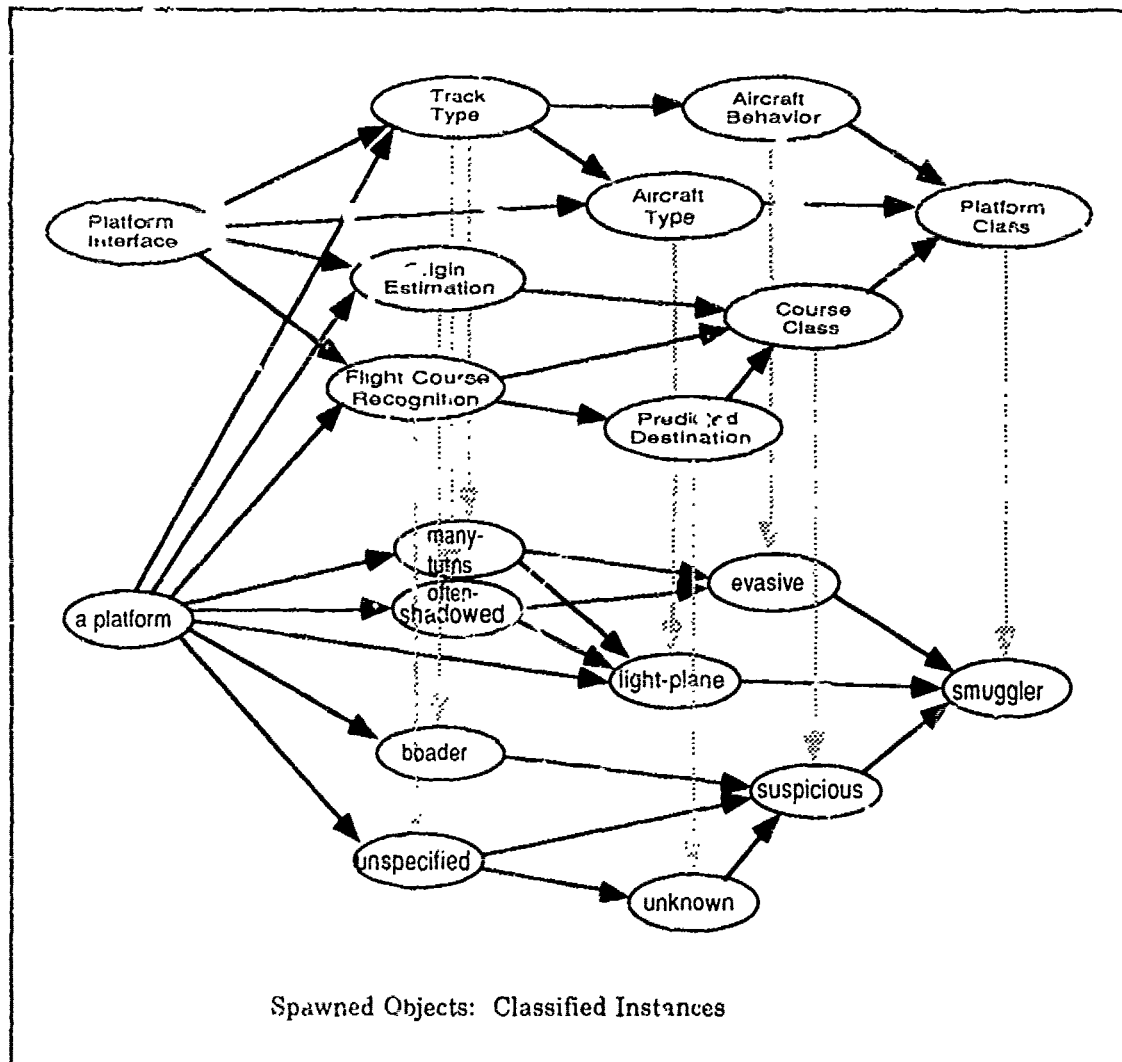


Figure 2.3. Dynamic Hypothesis Maintenance

Problem data is propagated through its instance network to justify the classification solutions in the classified-instances. If a classification of a classified-instance gets disproved, the classified-instance discards itself, propagates the negated solution to its

super-classifiers and super-classified-instances, and notifies its sub-classifier of its elimination. These sub-classifiers discard their classified-instances which have derived only the disproved classification solution. If a sub-classified-instance derives another classification solution, it is retained. If a super-classified-instance is discarded by the negation, it executes the same procedure. When the most abstract classified-instance is discarded, the hypothesis as a problem solution is denied. This scheme of instantiating, justifying, and discarding an instance network is the dynamic hypothesis maintenance as shown in Figure 2.3.

While instantiating and discarding an instance network, the problem may vary continuously. When a classifier receives a solution from its sub-classifier after creating an classified-instance, the problem data for the corresponding classification-category is forwarded to the classified-instance. After a classifier recognizes an instantiation in its super-classifier, it sends the problem data to the super-classified-instance directly. If a classified instance is eliminated, the problem data is forwarded to its classifier. A classified-instance is acting for a while for the data forwarding even after it is discarded. The classified-instance is actually discarded when its sub-classifier discards its reference.

A problem which has a hypothesis may succeed in forming another hypothesis so that the classifier network continues to work classifying the problem. In this circumstance a classifier does not invoke classifications for the instantiated classification-categories. Therefore, the classifier network can reduce its computation load after the hypothesis has been formed. When a problem has more than one hypothesis simultaneously, those classified-instances needed for both of these hypotheses are shared on the same instance network.

An instance network is organized only after a solution is formed in one of the most abstract classifiers, in order to create longer-lived classified-instances. Less abstract classification acts more frequently due to the problem propagation scheme of ConClass. If a classifier lower in the hierarchy instantiates a classified-instance, it may be quickly eliminated by a reclassification. Creation of such an ephemeral distributed object is expensive to manage.

3. Implementation of ConClass

3.1. Computational Environment

We developed ConClass on a simulated distributed-memory multiprocessor system called CARE [Delagi 87a] on a Lisp machine, Explorer¹, and implemented ConClass in a distributed processing language called LAMINA [Delagi 87b].

CARE is a distributed-memory, asynchronous message-passing architecture. CARE is simulated by a general, event driven, highly instrumented system called SIMPLE. CARE models 1 to 1000 processor-memory pairs communicating via a packet-switched cut-through interconnection network. Message delivery between processing elements is reliable, but messages are not guaranteed to arrive in the order of origination.

LAMINA is the basic language interface to CARE and consists of Common Lisp [Steele 84] and Flavors [Weinreb 80] with extensions. The extensions provide primitive mechanisms and language syntax for expressing and managing computational locality in each processing element and concurrency between processing elements. Three styles of

¹ CARE currently runs on Explorer, Symbolics, Sun-3, and DEC Station 3100.

programming are supported: functional, shared-variable, and object-oriented. ConClass system is implemented in the object-oriented language subset of LAMINA where we represented ConClass objects such as classifiers by means of LAMINA objects.

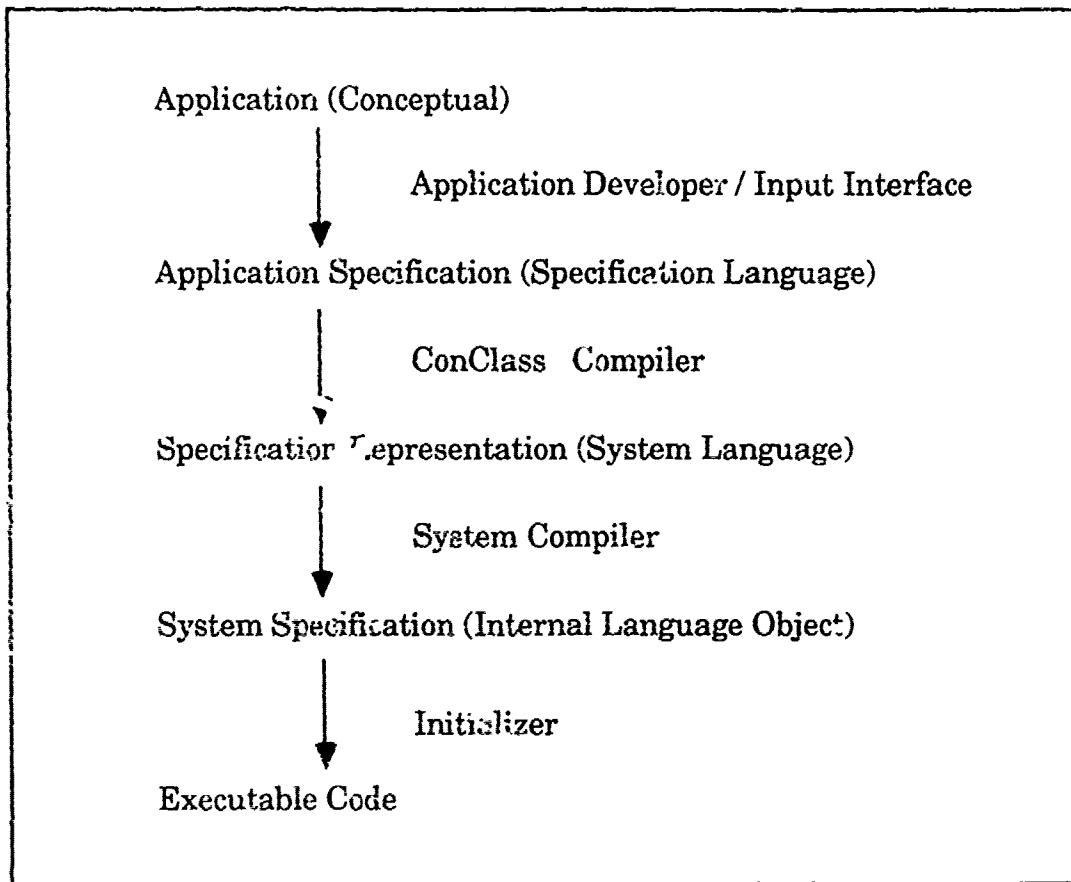


Figure 3.1. Representation Specification Hierarchy

3.2. Problem Description and Solving

Figure 3.1 shows the hierarchy of representation specifications and execution components in ConClass system. ConClass provides application developers with an environment for describing and decomposing classification problems. Figure 3.2 shows an example classifier definition in the application specification. The ConClass compiler translates application descriptions to object definition representations.² The initializer initiates LAMINA objects according to the definitions and allocates them on CARE processing elements.

ConClass system provides a development environment where application developers can specify classifier definitions and relationship between classifiers and interface-objects. Classification in a classifier is composed of classification-categories. A known class of phenomenon into which unknown phenomena are classified is defined by describing *templates* in the classification-category. A *template* is a conjunction of attribute values used

² We do not have the ConClass compiler implemented to date. We specified the experimental system described in section 4 directly in the object specification representation. However, the scheme for representing and decomposing problems was efficient for development.

in the classification-category and its classifier. An attribute can specify its condition by means of a value, a set of values, or a range of numerical values. A template succeeds in matching a problem object when the problem object's attributes satisfy the template value conditions. An *attribute* is problem data or a classification solution brought by the other classifiers or problem objects. An *attribute* can also be computed from other attributes anew in a classifier. A classification-category may have more than one template and it succeeds in classifying if the conditions of one of the templates are satisfied.

```

#| classifier definition |#
(Speed
 :Classification-Categories (Slow-Current-Speed
                             Medium-Current-Speed
                             Fast-Current-Speed
                             Slow-Max-Speed
                             Medium-Max-Speed
                             Fast-Max-Speed)
 :Classifier-Inputs ((X-Position Input-Interface)
                    (Y-Position Input-Interface)
                    (Z-Position Input-Interface)
                    (X-Velocity Input-Interface)
                    (Y-Velocity Input-Interface)
                    (Z-Velocity Input-Interface))
 :Classifier-Database ()
 :Classifier-Attributes (Current-Speed Max-Speed)
 :Super-Classifiers (Track-Type Current-Platform-Behavior)
 :Interface-Objects (Input-Interface)
 :Output-Objects ()
 :Locations (25 26 27 28)
 :Dynamic-Site-Positions ())

#| classification-category definition |#
(Slow-Current-Speed
 :Classifier Speed
 :Category-Inputs ()
 :Category-Database ()
 :Category-Attributes ()
 :Classification-Templates
 ;; templates: (template ...)
 ;; template: (template-slot ...)
 ;; template-slot: (var (capture-values lock-values)
                      (capture-confidence lock-confidence))
 ;; values: (:set value ... ),
            (:range (:open value) (:close value)),
            (:range t (:open value)), ...
 ((1.0 (Current-Speed ( (:range (:open :infinity-) (:close 5000))
                             (:range (:open :infinity-) (:close 6000)))
                      (.7 .5))))))

```

Figure 3.2. Sample Classifier Definition

Problems manipulated in ConClass can be continuous and dynamic. If problem data causes attributes to vary around the threshold of template conditions, a classification may change frequently. Therefore, a template condition can be specified by a set of two kinds of values which we call *capture value* and *lock value*. A *capture value* and a *lock value* are used when

unknown problem objects are classified and when classified solutions are justified, respectively. The range of a lock value needs to be larger than that of a capture value.

Classification solutions and attributes can carry confidence values. When a template's condition is satisfied, its classification confidence is the minimum value of attribute confidences in the template. A template can specify a minimum requirement on its confidence value, which must be satisfied for a successful classification. If the conditions of more than one template are satisfied, the classification confidence takes the maximum value of those template confidences. Some kinds of symbolic confidences are allowed to be used. This scheme is one of the most conservative methods to calculate confidences. A detailed description of confidence is illustrated in [Buchanan 84].

When a classifier succeeds in classifying, if it is one of the most abstract classifiers, it instantiates an initial hypothesis. Otherwise, it propagates the classification solution with specified attribute values to its super-classifiers. When a classification confidence is changed significantly, the change is propagated to the super-classifiers or to the super-classified-instances.

Application developers describe definitions of classifiers involving classification-categories and relationships between classifiers and interface-objects. Developers also define attributes of interface-objects. Developers need to define procedures for evaluating attributes and those confidences used in matching problem objects to the templates. ConClass generates the definition of a classified-instance according to the definitions of the corresponding classification-category and its classifier. The example shown in Figure 3.2 is the definition of a classifier with one of its classification-categories, which is to classify aircraft speed. This is a definition used in the experimental application described in Section 4.

3.3. Special Internal Controls

ConClass does not use physical synchronization schemes which may result in a saturation effect. ConClass incorporates embedded control features to manage a variety of asynchronous aspects of distributed processing.

We can use managers or schedulers responsible for creating and maintaining dynamic objects, synchronizing different processes, and coordinating searches. However, such agents may limit the system throughput when managing synchronization. Our related work reports various problems about such scheduling [Noble 88, Muliawan 89]. Schedulers can be overloaded, however, there are no clear-cut rules for the decomposition of such objects. ConClass uses no scheduler objects and handles no physical synchronization between objects.

In ConClass, variation of a problem object is propagated on the classifier network and instance networks by classifying and reclassifying the problem. The creation and elimination of instance networks are achieved by means of the propagation of creation and discard requests of classified-instances between objects, respectively. These propagation schemes do not require synchronization. However, such propagations may occur simultaneously and cause state conflicts in an object. For example, a classifier may receive a request for a classified-instance creation from its super-classifier while its sub-classifier is sending a message of disproving the classification. Each object of ConClass manages various requests efficiently considering the state transitions of instantiation and use no synchronization which may make other objects idle.

Classifiers and interface-objects are represented by means of LAMINA objects as described above. Classifiers and interface-objects communicate with each other using the message

passing facilities of CARE and LAMINA. Messages between objects in ConClass are not guaranteed to arrive in the order of origination because the message passing on CARE is asynchronous. For example: An object may receive stale data later than brand-new data. When a classified-instance is discarded shortly after being created, its sub-classifier may receive a discard request earlier than a creation one for its related classified-instances. ConClass adopts embedded features to properly manipulate all messages that are in the wrong order.

Classified-instances and problem objects are created dynamically and those references are propagated to other objects. A dynamic object is typically created on a different processing element than that of the creator according to the object allocation scheme described below. Although the creator does not receive a created object's reference until a later time, it keeps indirect reference to the new object, which can be used to send messages. The creator sends the indirect reference to other objects if the new object is being created so that the messages from those objects to the created one are sent via the creator indirectly. The direct reference is later propagated to those objects that hold an indirect reference automatically.

These features were useful for implementing the ConClass system.

3.4. Load Balancing

It is one of the goals of parallel distributed processing to allocate objects over processing elements such that the work they do is balanced as evenly as possible. We adopted the same modified random load balancing used by our related work [Nakano 88] to allocate classified-instances. This scheme involves random selection for dynamic objects from the set of all processing elements excluding those used by static objects if there are fewer static objects than processing elements. Otherwise, the dynamic object is allocated randomly from the set of all processing elements. The random allocation of classified-instances is reasonable because it is difficult to predict that any given classified-instance will be busier than another and because it is not suitable to allocate on the basis of statistics concerning non-permanent objects. In fact, empirical evidence suggests that in the absence of such load knowledge, random allocation is optimal [Nakano 88].

We allocated another sort of dynamic object, problem object, evenly on the processing elements dedicated to dynamic objects. Because problem objects, for example, aircraft platforms, exist more permanently, processing elements are assigned using a round-robin method.

Static objects, classifiers and interface-objects, can be replicated as much as desired as described above. These objects are allocated to the processing elements dedicated to static objects in advance according to domain knowledge, statistics, and user definition³.

4. Performance Evaluation

We implemented an experimental application system in ConClass and confirmed the efficiency of the ConClass system.

³ We defined the allocation of static objects in implementing the experimental application system because of the absence of the ConClass compiler.

4.1. Experimental Problem

We have been developing an aircraft radar signal interpretation system called AirTrac for tracking and classifying aircraft. The AirTrac system is composed of three major modules: Data Association, Path Association, and Platform Interpretation. Data Association accepts aircraft signal reports of multiple radar systems at regular time intervals and periodically abstracts the radar signal reports into observation records for individual signal tracks [Nakano 88]. Path Association reports hypothesized platforms to which the periodic observation records are associated to form tracks for the same aircraft [Noble 88, Muliawan 89]. Platform Interpretation analyzes and interprets information contained in platforms and provides continuous real-time assessments about the observed aircraft.

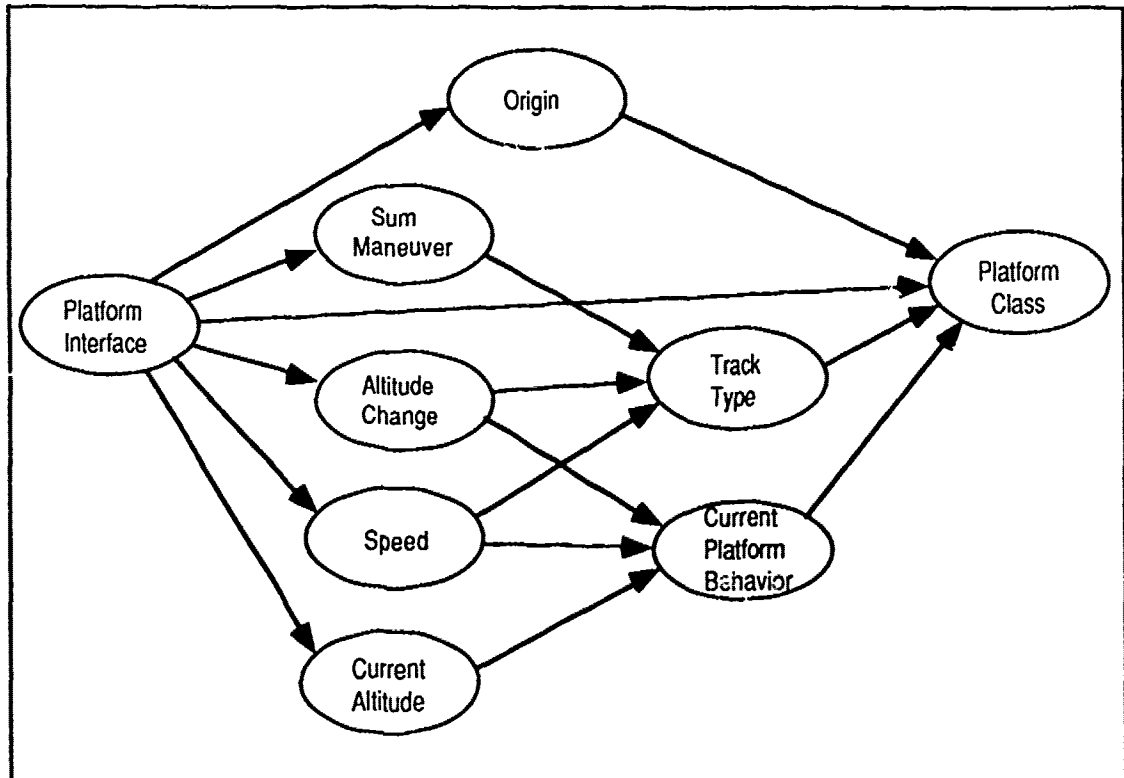


Figure 4.1. Experimental Classification System

The problem selected for our experiments is a simplified experimental implementation of AirTrac's Platform Interpretation module. The configuration of the experimental system is shown in Figure 4.1. Each sub-problem solver is implemented by means of a classifier and its classification facilities. This system consists of eight classifiers which have between two and ten classification-categories. The system input is a series of simplified emulated aircraft platforms which have aircraft position and sizes information.

The experimental application requires the following:

- The classification system is a hierarchy of classifiers which have multiple fan-in and fan-out.
- The classifier network has cut-through connections.
- The problem data is continuous and problem solving in each classifier is potentially dynamic.

These requirements are in order to evaluate the experimental system in an environment where configuration and computation are uneven between problem solvers. The experimental system meets these requirements.

4.2. Experimental Results and Analysis

We experimented with the data of 50 aircraft platforms which appeared in real-time successively and were classified and reclassified typically three times in the classifications lower in the hierarchy. The experiment has two parameters: the number of processing elements and the data rate. The numbers of processing elements used were 8, 16, 32, 64, and 256. The data rate is the frequency at which problem data is fed into the application system. We can change the data rate by altering the sampling frequency of observed problem data. This scheme, however, will change the frequency of classification in relation to the data rate. In order to maintain the classification quality between the data rates, we changed the data rate by altering the time interval of feeding the same set of data. Thus, the experimental application system was performed using the time of the emulated data while we evaluated the performance of ConClass system using the simulation time of the CARE system.

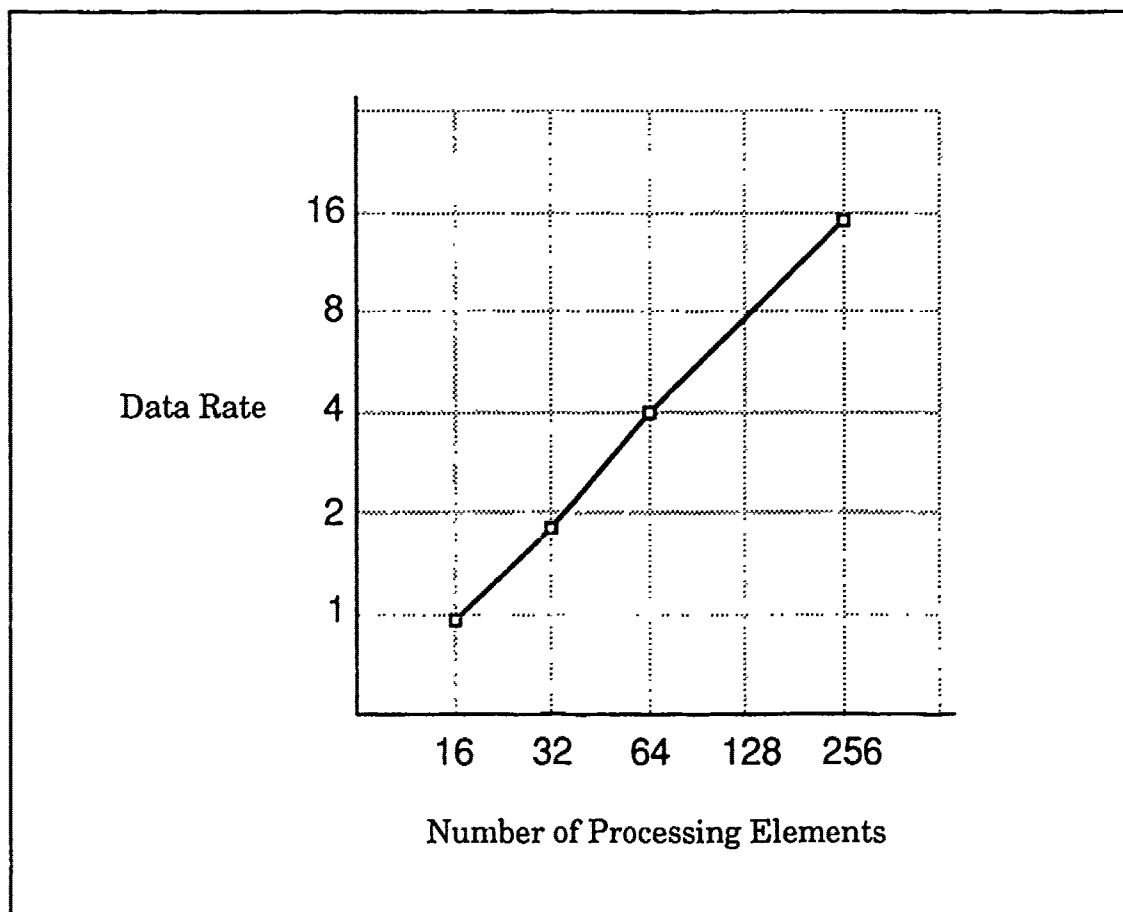


Figure 4.2. Speedup Curve (based on sustainable data rate)

The experimental system achieved a linear speedup against the number of processing elements as shown in Figure 4.2. The speedup was based on the sustainable data rate, the maximum data rate for which all measured latencies stabilize and do not increase over time. See appendices for the observed latencies from which the speedup was evaluated. The load

balancing in ConClass uses two methods: replication and allocation of static objects, and the assignment of processing elements for dynamic objects. We assumed that we could optimize these factors using domain knowledge and statistics. Therefore, we fine-tuned the factors in the experiment so as to optimize the results. Another reason for the optimization was to evaluate the processing speed with respect to achieving an efficient concurrency. In addition, we implemented the ConClass system paying attention to even the execution efficiency of Lisp functions. This was to more precisely evaluate the system overhead for parallel processing.

The CARE simulation system has an user interface where we can observe a variety of statistics and latencies of CARE components. Figure 4.4 shows the processor utilization graph whose upper half specifies utilization of evaluators which execute actual data computation. The lower half specifies that of operators which manage the communication between processing elements. In a typical classification situation, for example, ConClass was able to use 28 to 30 processing elements at a time out of a possible 32. Including the initialization of ConClass, which brought about considerable computation, the overall average of concurrent utilization was 21 processing elements. Because the classification computation in ConClass is coarse-grained, the operators are not busy.

In ConClass, the concurrency designed by an application developer can correspond effectively to actual computational hardware components. We were able to implement the experimental application system efficiently using this scheme. It was easy to estimate replication and allocation of objects and assign processing elements. The ConClass development environment for describing and decomposing classification problems was useful. ConClass execution facilities excluding schedulers and various synchronization schemes improved the efficiency of parallel distributed processing.

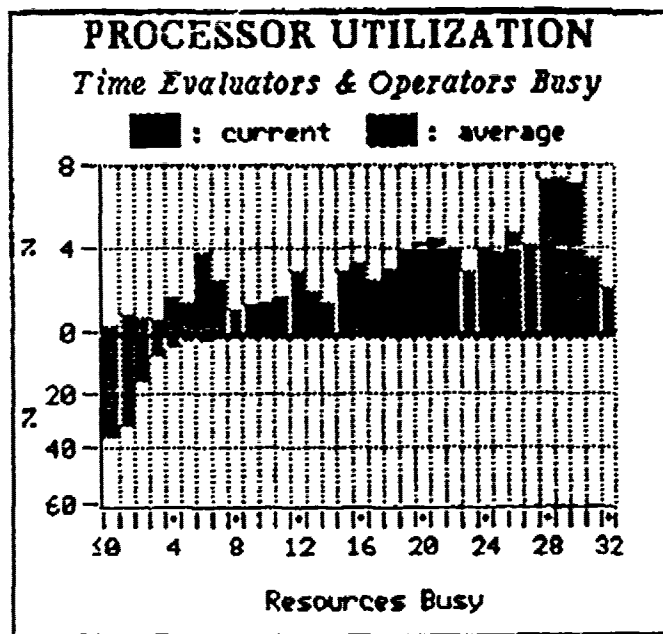


Figure 4.3. Processor Utilization

Table 4.1 shows the frequencies of messages sent between objects to solve the experimental problem. These frequencies correspond to sustainable data flows. Most messages are propagated between classifiers and between classified-instances. Messages can be sent between the classifier network and instance networks while creating and

discarding classified-instances. Data propagation messages can be sent from classified-instances to their related but uninstantiated classification-categories. The former messages are not frequent because classified-instances are long-lived according to the instantiation scheme of ConClass as described above. The latter situation is fairly rare due to the classification problem's structure. The ratio of number of messages to classified-instances decreases as the number of processing elements increases. This is because the instantiation time becomes longer compared to the experimental simulation length. However, this is not a factor which can affect the independence between the classifier network and instance networks. Although more dynamic problems may increase the interactions between the two kinds of networks, the experimental results show the efficiency of a dynamic load distribution of ConClass.

Table 4.1. Message Frequencies

Messages	Number of Messages (Percentage)				
	Processing Elements	16	32	64	256
To Classifiers		5448(51.0)	5445(51.2)	5527(56.1)	5409(79.3)
From Instances		313 (2.9)	268 (2.5)	389 (4.0)	131 (1.9)
To Classified-instances		5248(49.0)	5181(48.8)	4317(43.9)	1409(20.7)
From Classifiers		167 (1.6)	180 (1.7)	273 (2.8)	206 (3.0)

5. Conclusions

In this paper we have described the parallel solution of classification problems. The developed framework, ConClass, is capable of classifying continuous real-time problems dynamically and concurrently.

ConClass provides a high-level structure for describing and decomposing classification problems. The ConClass classification system can handle multiple sources of problem inputs as well as dynamic global information. A ConClass application can use static knowledge to solve problems in the system. Such a high-level framework was useful in implementing the experimental application in ConClass.

Classification problem solving can be structured hierarchically by means of decomposing problem and synthesizing solutions. We implemented the ConClass framework based on this characteristic so that decomposed problem solving modules were directly represented as distributed processing components. Therefore, the concurrency designed by developers is effectively reflected in the actual parallel computation and this scheme makes it easier to anticipate and evaluate the system behavior. Moreover, a decomposed classification problem solver, consisting of a classifier and its classified-instances, is very uniform in

terms of its basic structure and execution mechanism. These features are useful in the design of concurrency and the implementation of efficient distributed processing. The classification execution in ConClass is intrinsically parallel, in contrast to our previous problem solving frameworks [Brown 86, Nii 89, Saraiya 89] which report various problems of parallel processing.

We implemented the replication features of static objects for preventing a particular object from being overloaded. The dynamic creation of problem objects may cause the system load to increase. We incorporated the load distribution scheme by means of dynamically creating instance networks which maintain hypotheses as solutions of problem objects. We implemented an efficient execution mechanism for ConClass without using schedulers or synchronization schemes which are liable to be bottlenecks. We confirmed the efficiency of the parallel processing and the load balancing of ConClass by an experiment.

ConClass is a concurrent problem solving framework using a structural hierarchy of classification problem and continuity of problem data. Real-time problem solving systems are increasing in importance and we realize the advantage of the ConClass framework. Furthermore, ConClass suggests a construct for dynamic information fusion and multiple assessments. AirTrac, a part of which we selected as an experimental application, is an example: AirTrac fuses information such as radar signal, flight plans, ground information, aircraft knowledge, and geography. AirTrac reports real-time assessments such as aircraft classifications and predictions of flight courses and aircraft actions. The hierarchical structure of decomposing a problem and synthesizing solutions are useful and effective for implementing these functions.

Acknowledgments

Many of these ideas were improved by discussions with Harold Brown. I would like to thank him for providing valuable suggestions throughout this project. I would also like to thank Nakul Saraiya for his tireless support for using the CARE/LAMINA system. I am indebted to the members of the Advanced Architectures Project for nurturing a rich research environment. The Symbolic Systems Resources Group of the Knowledge Systems Laboratory provided excellent support of our computing environment. Special thanks are due to Edward Feigenbaum for his continued leadership and support of the Knowledge Systems Laboratory which made this research possible. George Fukuda provided an opportunity to do the reported research.

This research was supported by DARPA Contract F30602-85-0012 and NASA Ames Contract NCC 2-220-S1.

References

- [Bennett 78] Bennett, J., Creary, L., Engelmores, R. and Melosh, R., SACON: A Knowledge-Based Consultant for Structural Analysis, *Technical Report HPP-78-23*, Stanford Univ. (1978).
- [Brown 86] Brown, H. D., Schoen, E. and Delagi, B. A., An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures, *Technical Report KSL-86-69*, Stanford Univ. (1986) and *Proceedings of DARPA Expert Systems Workshop* (1986) 93-105.

- [Brown 82] Brown, J. S., Burton, R. R. and De Kleer, J., Pedagogical, Natural Language, and Knowledge Engineering Techniques in *SOPHIE I, II, and III*, in Sleeman, D. and Brown, J. S. (Ed.), *Intelligent Tutoring Systems*, Academic Press (1982) 227-282.
- [Buchanan 84] Buchanan, B. G. and Shortliffe, E. H., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing (1984).
- [Bylander 86] Bylander, T. and Mittal, S., CRSL: A Language for Classificatory Problem Solving and Uncertainty Handling, *AI Magazine* 7(3) (1986) 66-77.
- [Clancey 84] Clancey, W. J., Classification Problem Solving, *Technical Report STAN-CS-84-1018*, Stanford Univ. (1984) and *Proceedings of the AAAI-84* (1984) 49-55.
- [Clancey 85] Clancey, W. J., Heuristic Classification, *Technical Report KSL-85-5*, Stanford Univ. (1985) and *Artificial Intelligence* 27 (1985) 289-350.
- [Cohen 85] Cohen, P., Davis, A., Day, D., Greenberg, M. Kjeldsen, R., Lander, S., and Loiselle, C., Representativeness and Uncertainty in Classification Systems, *AI Magazine* 6(4) (1985) 136-149.
- [Delagi 87a] Delagi, B. A., Saraiya, N. P., Nishimura, S. and Byrd, G. T., An Instrumented Architectural Simulation System, *Technical Report KSL-86-36*, Stanford Univ. (1987) and *Proceedings of DARPA Expert Systems Workshop* (1986) 106-118.
- [Delagi 87b] Delagi, B. A., Saraiya, N. P. and Byrd, G. T., LAMINA: CARE Applications Interface, *Technical Report KSL-86-67*, Stanford Univ. (1987) and *Proceedings of the Third International Conference on Supercomputing* (1988) 12-21.
- [Laffey 88] Laffey, T. J., Cox, P. A., Schmidt, J. L., Kao, S. M. and Read, J. Y., Real-Time Knowledge-Based Systems, *AI Magazine* 9(1) (1988) 27-45.
- [Muliawan 89] Muliawan, D., Performance Evaluation of a Parallel Knowledge-Based System, *Technical Report KSL-89-51*, Stanford Univ. (1989).
- [Nakano 88] Nakano, R., Minami, M. and Delaney, J., Experiments with a Knowledge-Based System on a Multiprocessor, *Technical Report KSL-89-16*, Stanford Univ. (1989) and *Third International Supercomputing Conference*, Information Sciences Institute (1988).
- [Nii 89] Nii, H. P., Aiello, N. and Rice, J., Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems, *Technical Report KSL-88-66*, Stanford Univ. (1988) and in Gasser, L. and Huhns, M. N. (Ed.), *Distributed Artificial Intelligence II*, Morgan Kaufmann (1989).

- [Noble 88] Noble, A. C. and Rogers, E. C., AIRTAC Path Association: Development of a Knowledge-Based System for a Multiprocessor, *Technical Report KSL-88-41*, Stanford Univ. (1988).
- [Rich 79] Rich, E., User Modeling via Stereotypes, *Cognitive Science* 3 (1979) 355-366.
- [Rice 89] Rice, J., The Advanced Architectures Project, *Technical Report KSL-88-71*, Stanford Univ. (1989) and *AI Magazine* 10(4) (1989) 26-39.
- [Saraiya 89] Saraiya, N. P., Delagi, B. A. and Nishimura, S., Design and Performance Evaluation of a Parallel Report Integration System, *Technical Report KSL-89-16*, Stanford Univ. (1989).
- [Steele 84] Steele Jr., G. L., *Common Lisp: The Language*, Digital Press (1984).
- [Weinreb 80] Weinreb, D. and Moon, D., Flavors: Message-passing in the Lisp Machine, *AI memo 602*, MIT AI Lab. (1980).

Appendices

A1. Problem Data Profile

Figure A1.1 specifies the data used in the experimental classification system. This figure shows the numbers of total and new problem objects at every data input.

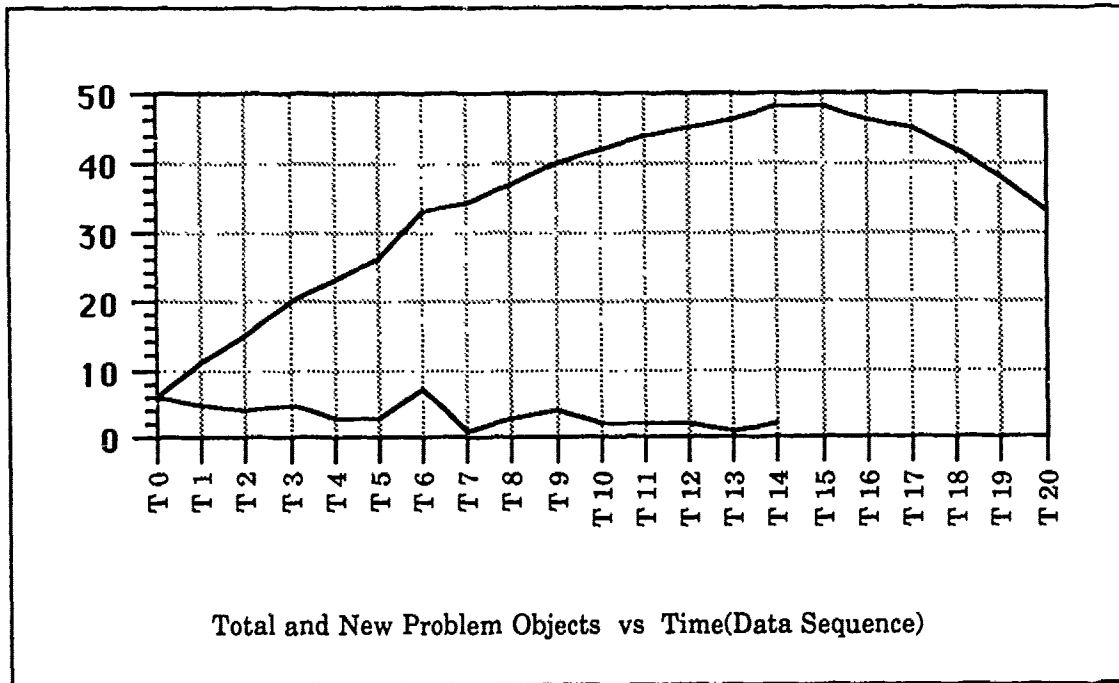


Figure A1.1. Problem Data Profile

A2. Latencies

The following are observed latencies from which we evaluated the sustainable data rates. We observed the latencies of forming initial hypotheses, making other hypotheses, and disproving those hypotheses. We compared the latencies with the same object allocation on each set of processing elements. We denote a processing element a PE and use the legend as shown in Figure A2.1 for the latency figures in this section.

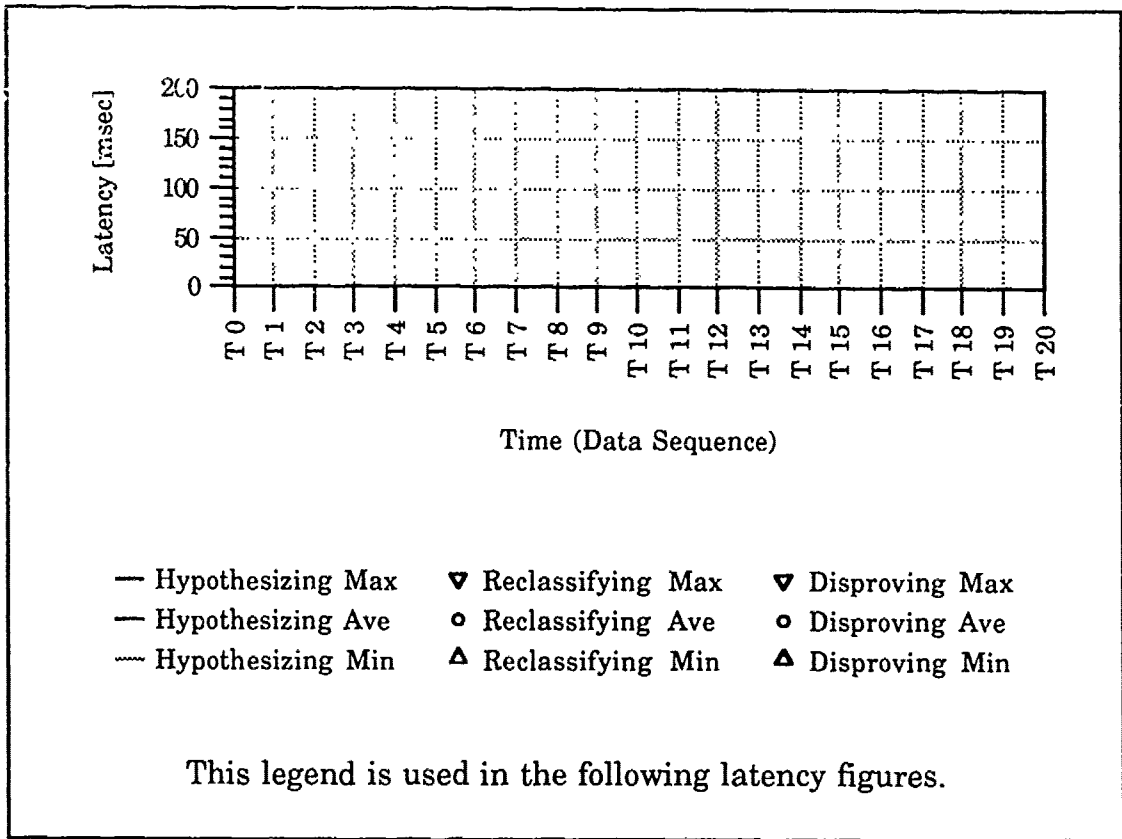


Figure A2.1. Legend for Latency Figures

A2.1. Latencies at Sustainable Data Rates

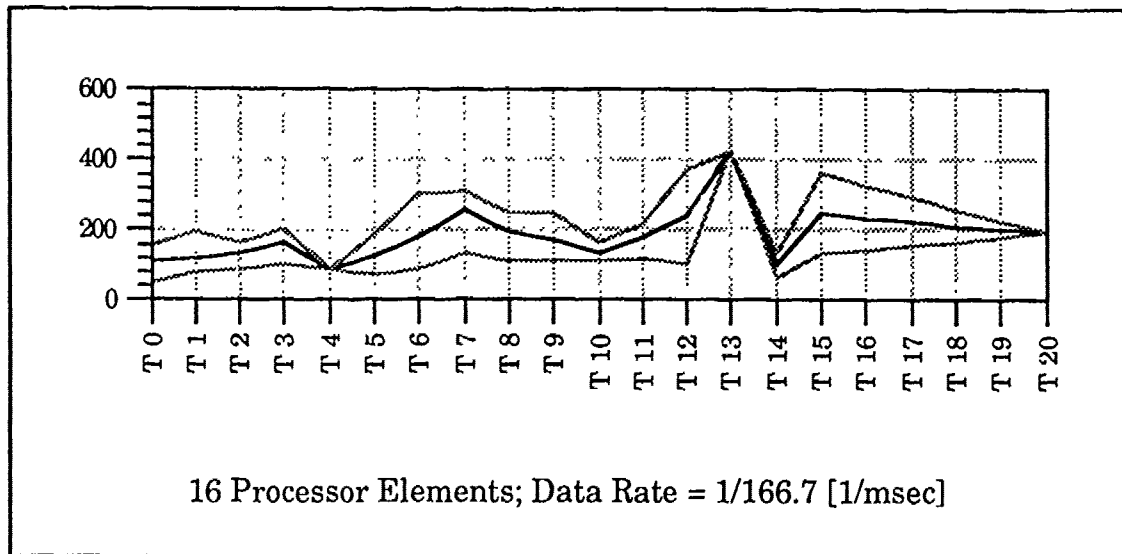


Figure A2.1.1. Hypothesizing Latency on 16 PEs

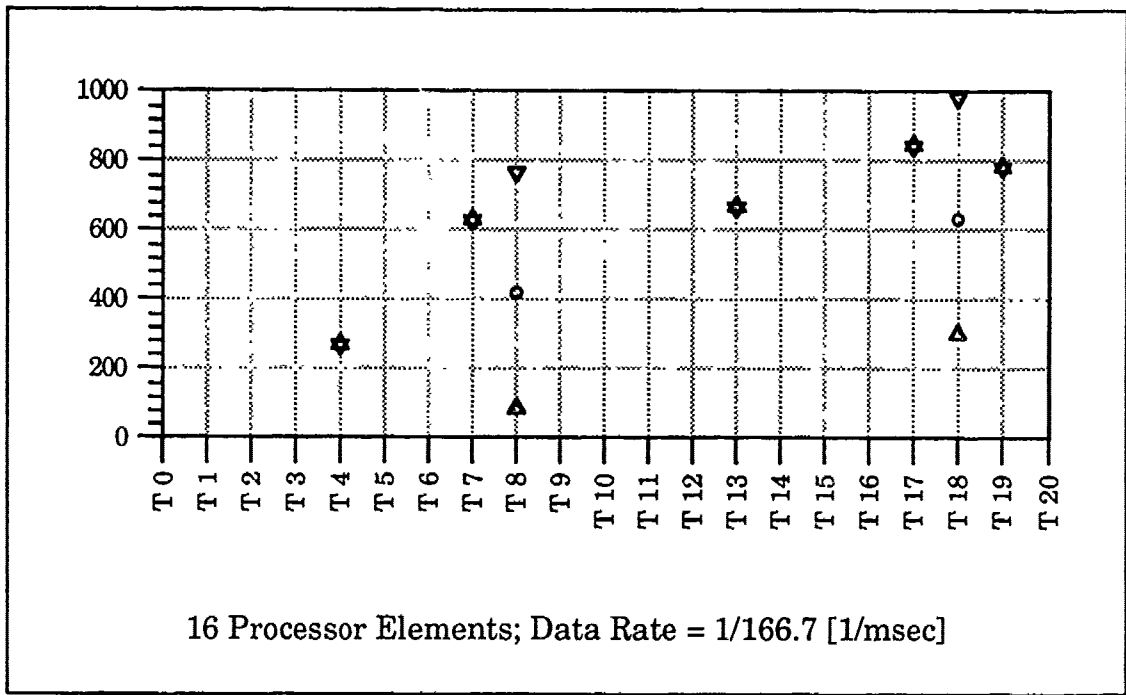


Figure A2.1.2. *Reclassifying Latency on 16 PEs*

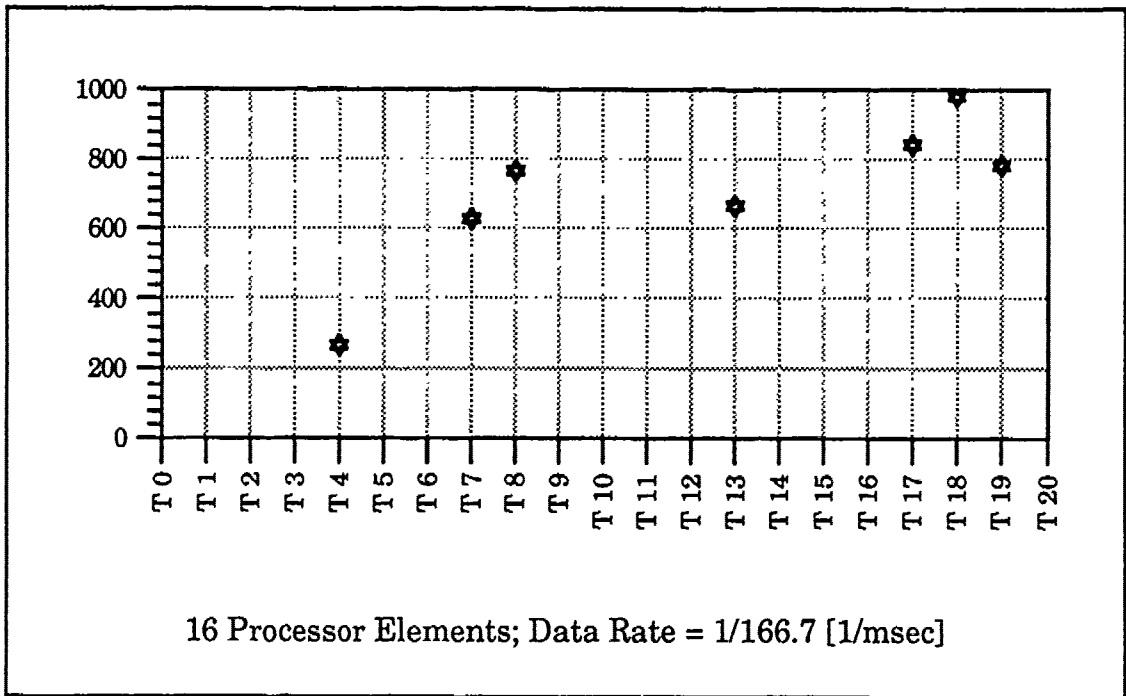


Figure A2.1.3. *Disproving Latency on 16 PEs*

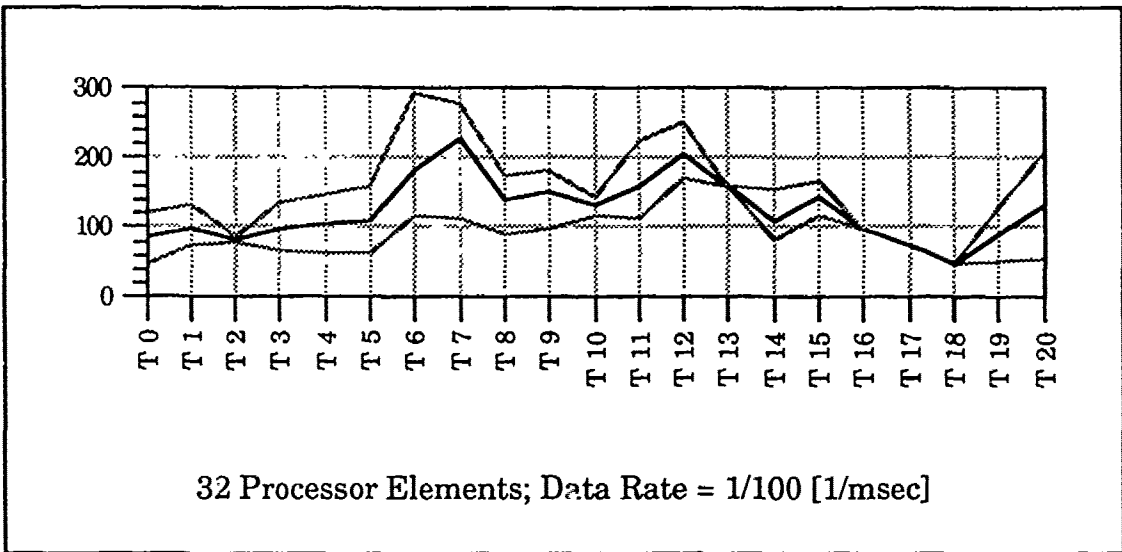


Figure A2.1.4. Hypothesizing Latency on 32 PE's

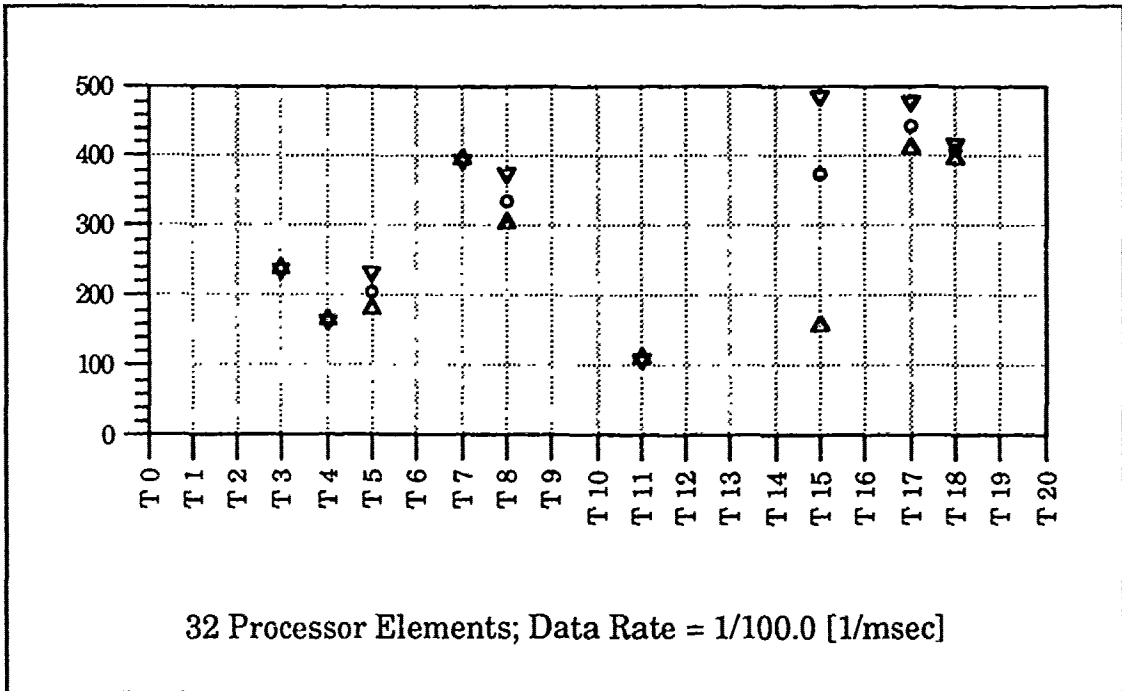


Figure A2.1.5. Reclassifying Latency on 32 PE's

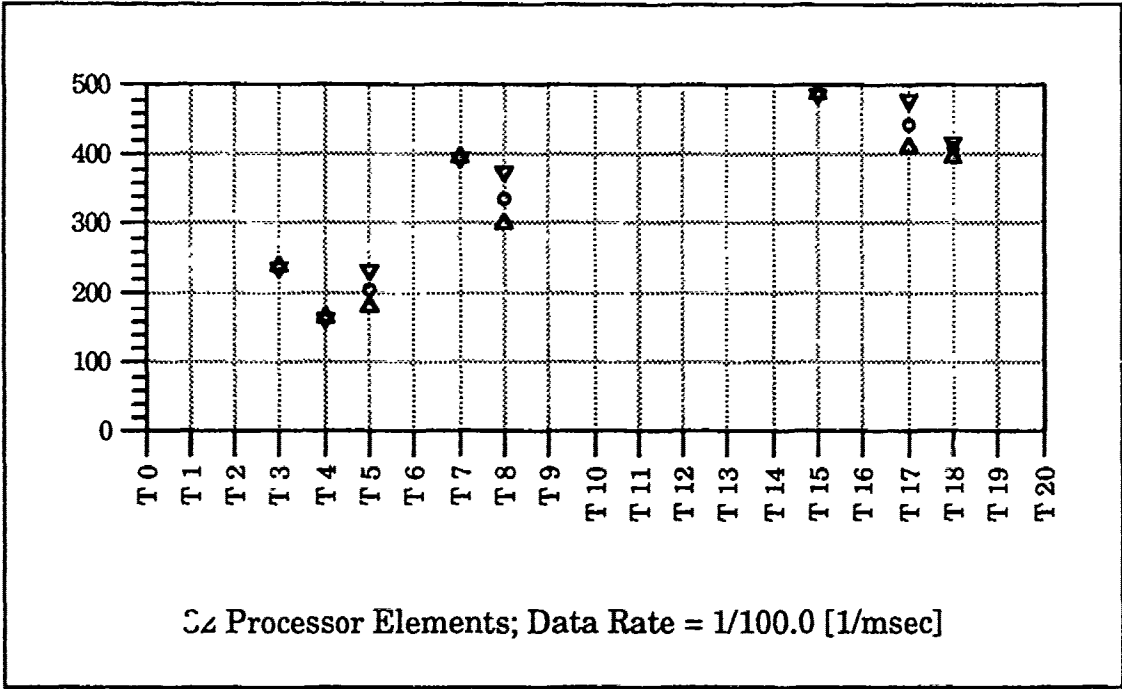


Figure A2.1.6. Disproving Latency on 32 PEs

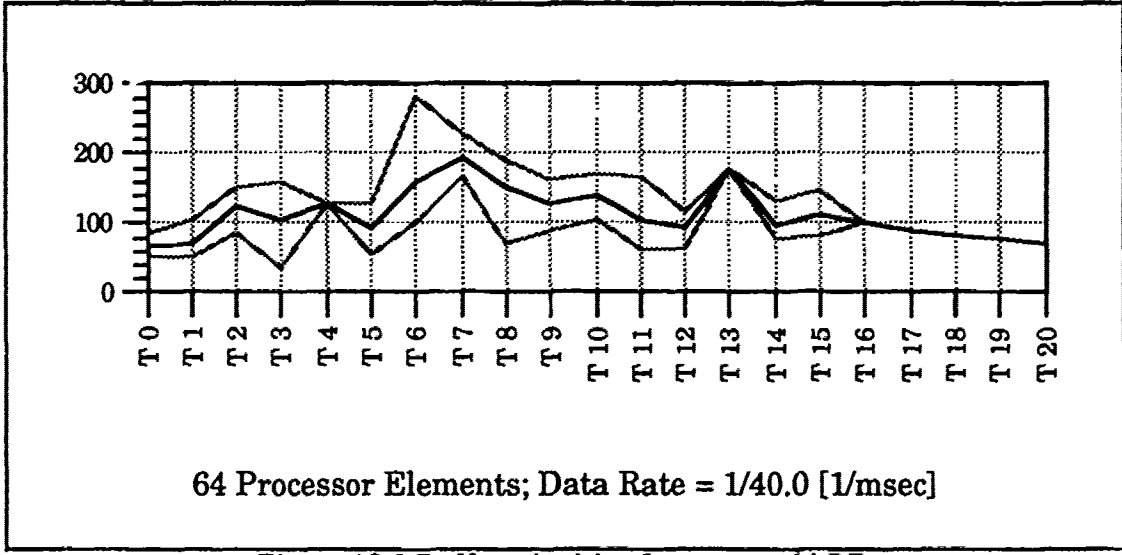


Figure A2.1.7. Hypothesizing Latency on 64 PEs

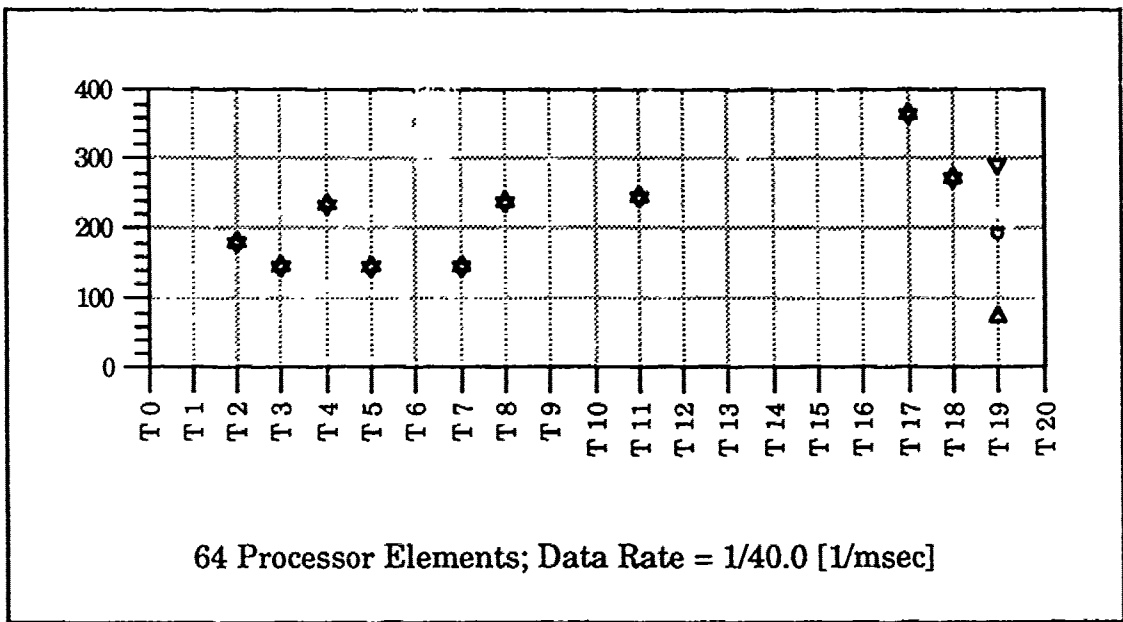


Figure A2.1.8. Reclassifying Latency on 64 PEs

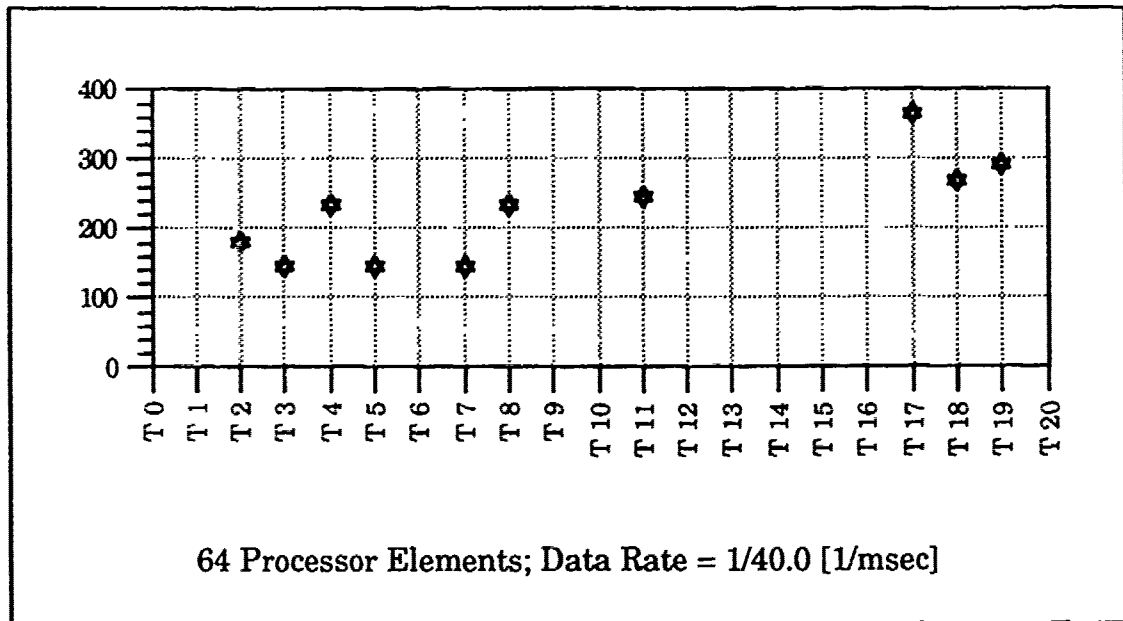


Figure A2.1.9. Disproving Latency on 64 PEs

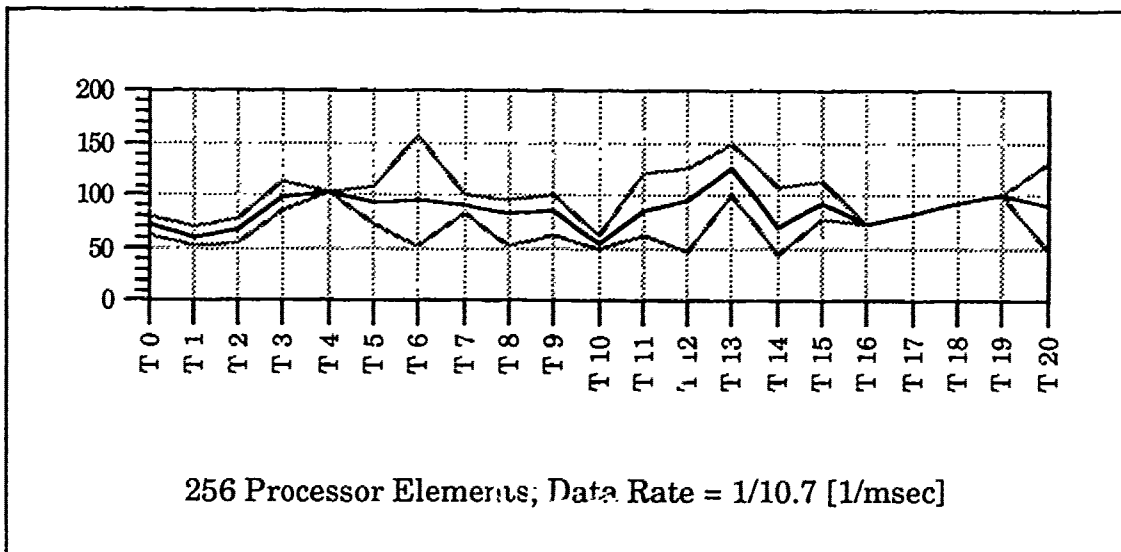


Figure A2.1.10. Hypothesizing Latency on 256 PEs

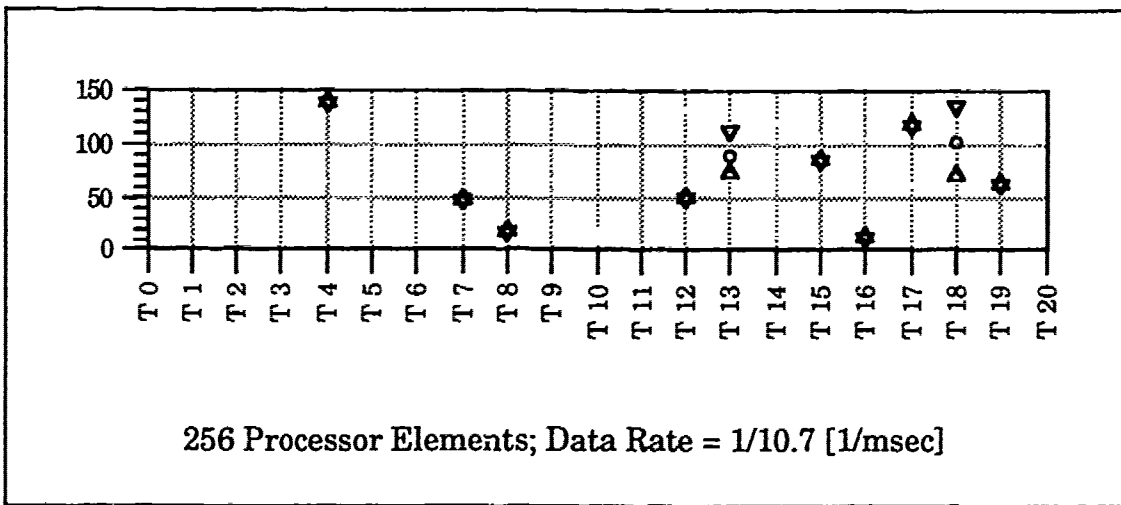


Figure A2.1.11. Reclassifying Latency on 256 PEs

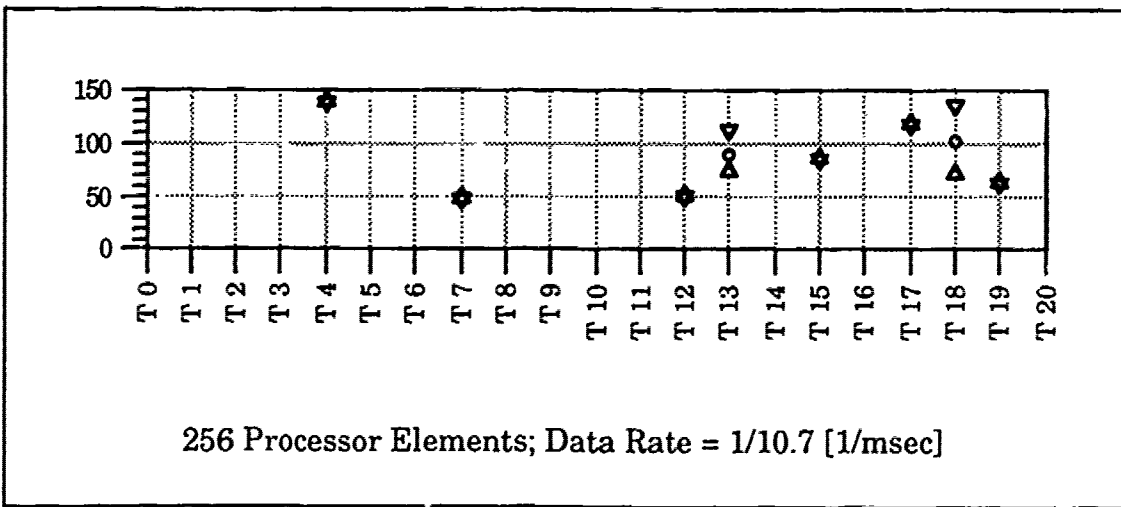


Figure A2.1.12. Disproving Latency on 256 PEs

A2.2. Sample Latencies at Overloaded Data Rates

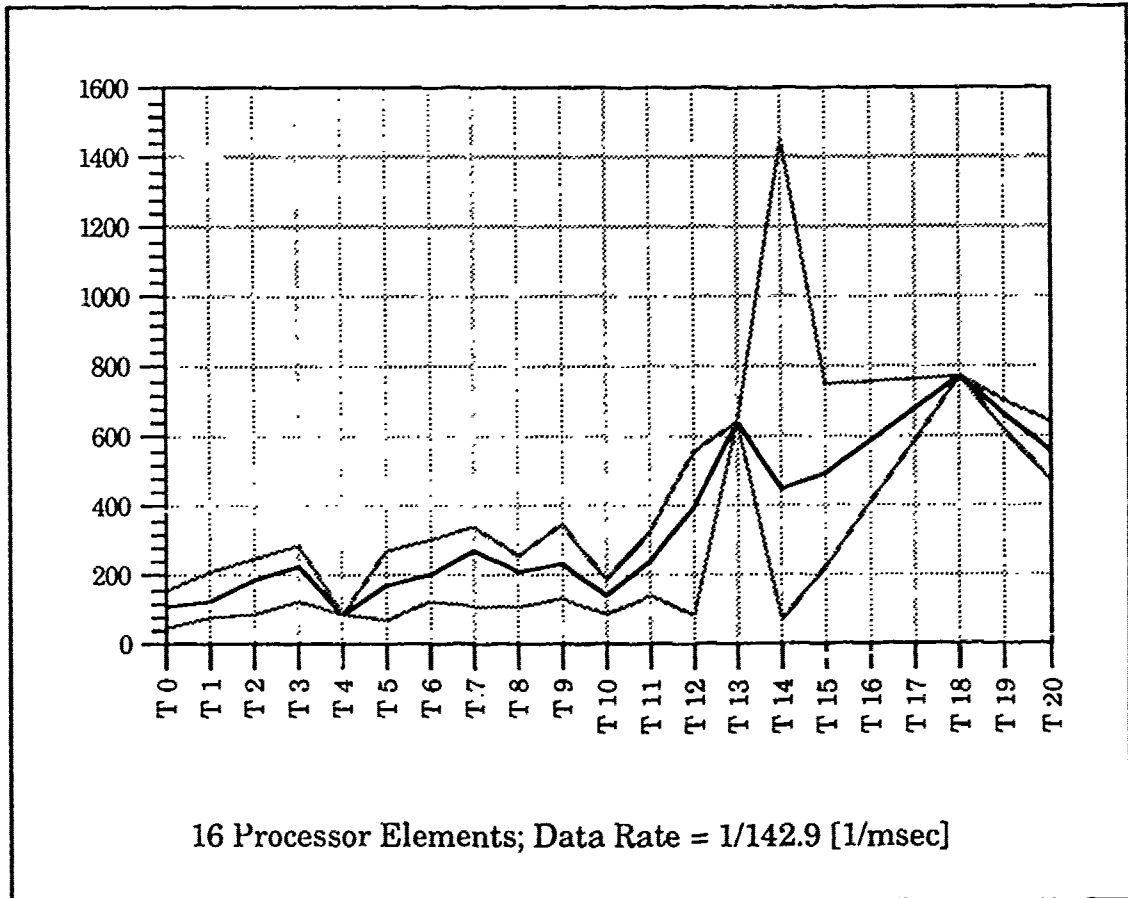


Figure A2.2.1. Hypothesizing Latency on 16 PEs

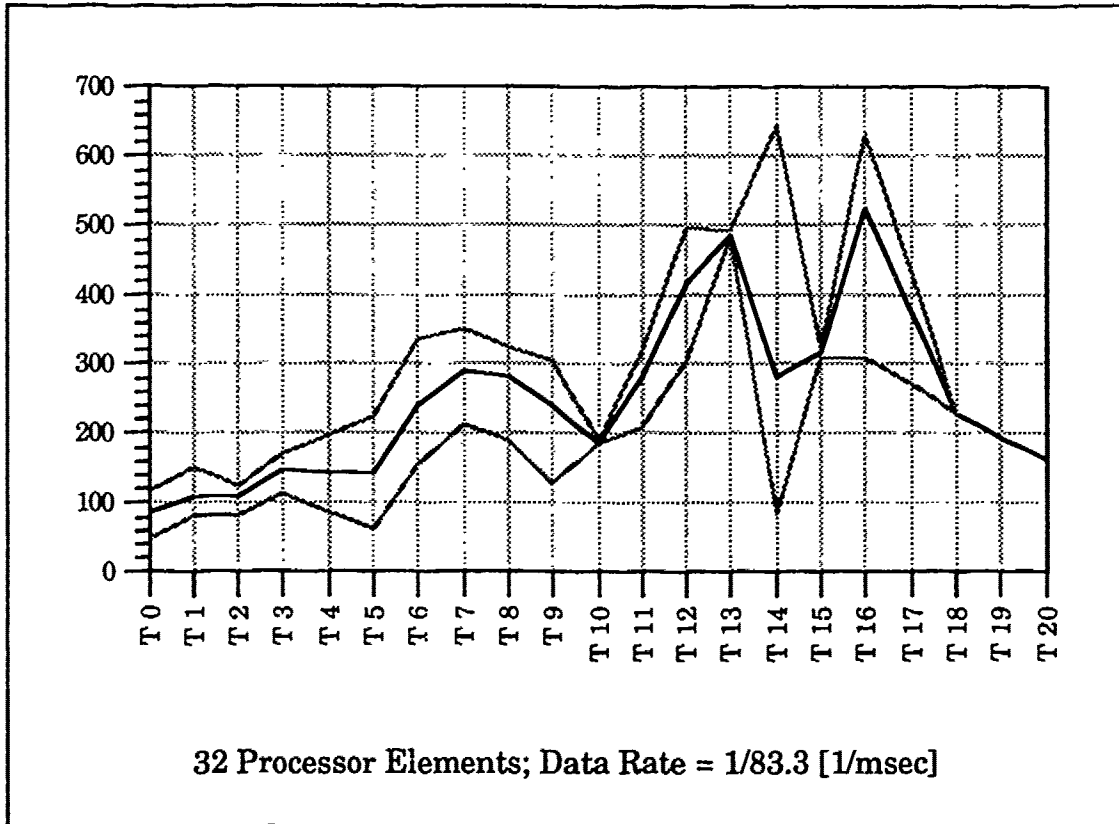


Figure A2.2.2. Hypothesizing Latency on 32 PEs

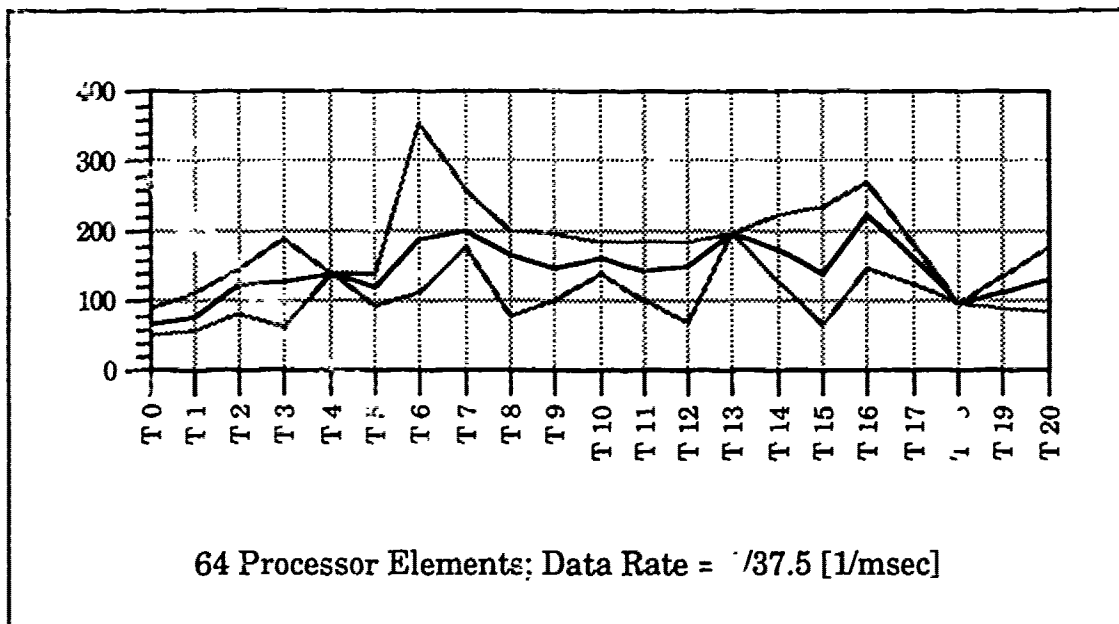
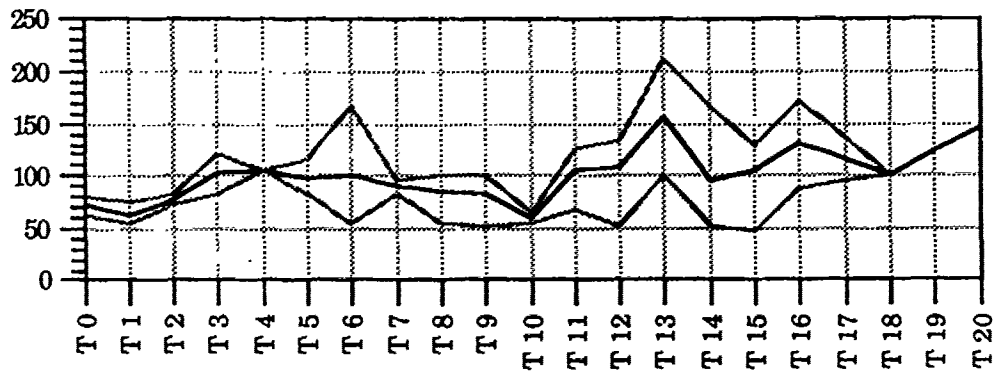


Figure A2.2.3. Hypothesizing Latency on 64 PEs



256 Processor Elements; Data Rate = 1/9.38 [1/msec]

Figure A2.2.4. Hypothesizing Latency on 256 PEs

Performance Evaluation of a Parallel Knowledge-Based System

by

Djuki Muliawan

**KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305**

*This work was supported by DARPA Contract F30602-85-C-0012
and NASA Ames Contract NCC 2-220-S1*

Abstract

This paper discusses the quantitative and qualitative performance of a module of a parallel knowledge-based system for tracking and classifying aircraft, called Airtrac. The Airtrac system is built to gain some understanding of the potential speedup through concurrency of reasonably large and complex continuous signal understanding systems. Airtrac runs on CARE, a simulated distributed-memory, asynchronous message-passing multicomputer.

Evaluating the performance of a continuous parallel knowledge-based system such as Airtrac is difficult. The simple approach of timing its execution would not work, since the system is continuous. Furthermore, the performance is usually multidimensional and cannot easily be expressed into a single number. The notions of *latency*, *excess ratio*, *sustainable data rate*, and *capacity* are instead used to rate the performance of the system.

The paper reports the effects of important high-level control strategies on the system performance, the effects of varying the frequency and width of the input data across different numbers of processors, and some possible speedup curves of the overall system performance as a function of the number of processors.

Finally conclusions are presented in the relationship between the quantitative and qualitative performance of Airtrac, and in the potential speedup of large and complex parallel knowledge-based systems.

1. Introduction

This report documents the enhancement and performance evaluation of a module of Airtrac, a parallel knowledge-based system for tracking and classifying aircraft. The system is being developed within the Advanced Architectures Project (AAP) at the Knowledge Systems Laboratory at Stanford University [5, 7]. John Delaney of the MIT's Lincoln Laboratory created the system's concepts in 1985-87 while he was a visiting scholar at the KSL.

The primary goal of the AAP is to realize parallel software and hardware architectures to achieve significant speedup in the performance of knowledge-based systems. For some such systems, projected performance limits of uniprocessors fall short of the speed required by as much as a couple orders of magnitude. Multiprocessor parallel computing must be attempted to attain the necessary levels of performance.

The approach taken by the AAP to achieve this goal is to study all levels of the computational hierarchy, from hardware to programming languages to problem-solving frameworks to applications. The AAP is performing many experiments to understand the computational characteristics and potential speedup of reasonably large and complex parallel knowledge-based systems. Each experiment represents a narrow, vertical slice through the space of design alternatives.¹

Airtrac represents the application level of the vertical design slice. The experiment has the following two goals:

- To investigate the potential speedup via concurrent processing of reasonably large and complex knowledge-based systems.
- To understand useful software constructs for parallel signal understanding systems.

The remainder of this paper is organized as follows. Section 2 provides an overview of the Airtrac application as a whole. Section 3 gives a detailed description of the Airtrac Path Association module (or PA for short). Section 4 discusses the experiment design. Section 5

¹ A summary of systems and experiments done within the AAP can be found in [8].

describes and explains the performance of PA. Section 6 presents some high-level conclusions, and Section 7 describes future work.

2. Overview of the Airtrac Application

This section provides a general description of the Airtrac application. Its motivation, functionality, architecture, and implementation will be discussed. Much of the material in this section can originally be found in [7].

2.1. Motivation

The underlying motivation is to gain some understanding of the potential speedup through concurrency of relatively large and complex continuous signal data interpretation systems. A similar application to Airtrac called ELINT [1] was built earlier within the AAP. That experiment demonstrated a speedup via concurrency of up to two orders of magnitude. However, ELINT is simple and unrealistic in its reasoning. This motivates the attempt to build a more complex and realistic signal understanding system, and investigate its potential speedup through concurrent processing. Airtrac is the result of this attempt.

2.2. Airtrac's Functionality

Given continuous track data from one or more radar and signal processing systems in a particular region of airspace, the Airtrac system monitors and classifies, in real time, the flight of aircraft within the region, and interprets and predicts their behavior. Airtrac processes data as soon as they are available. It is essentially a knowledge-based information fusion system. Data are collected from multiple sources at different times to create a consistent and comprehensive picture of aircraft in the given region of airspace. Airtrac uses heuristics to deal with incomplete and errorful information.

2.2.1. Airtrac's Input

The inputs to Airtrac are (simulated) output data from one or more active radar and signal processing systems tracking aircraft in a given region of airspace. Each piece of input data, called a *Radar Track Report (RTR)*, represents the observation of an aircraft from one radar site during a periodic time interval (*scantime*¹). Each radar observation is processed by the radar and signal processing system so that an RTR provides the information listed in Table 2.1.

Table 2.1 Airtrac's Input

Each Radar Track Report contains the following:

<i>observation scantime</i>	the time of the observation
<i>radar ID</i>	the identifier of the radar site observing the track
<i>track ID</i>	an integer (unique to the radar site) assigned by the radar to the track to which the observation belongs
<i>aircraft type</i>	the type of the aircraft under observation, indicated by signal characteristics
<i>position</i>	the location (x, y) of the aircraft at the time of the observation
<i>position covariance</i>	the (E_x E_y) estimate of the error associated with the reported position
<i>velocity</i>	the velocity (V_x V_y) of the aircraft calculated at the time of the observation
<i>velocity covariance</i>	the (E_x E_y) estimate of the error associated with the reported velocity

A radar assigns a unique *track ID* to an aircraft track when the radar observes the track for the very first time. The radar continues to assign the same *track ID* to subsequent

¹ In this experiment 1 scantime = 10 data time units, which are usually interpreted as seconds.

observations if it determines, usually by a simple track extension algorithm, that those observations correspond to the previously-detected track.¹ Otherwise, the track is considered finished and its *track ID* is dropped. Consequently, when a radar fails to continuously keep track an aircraft's full flight path, the path may be represented by several tracks. The radar is assumed to be capable of determining the *type* of each aircraft under observation from the particular characteristics of signals it receives. The algorithm employed by the radar also calculates *covariance* figures for the position coordinates and velocity vectors that it reports, providing a measure of the probability of error associated with each of these values. This error information is based on factors such as the strength of the signals and the distance between the aircraft and the radar site. All of this information is passed along as input to Airtrac in the form of an RTR.

2.2.2. Airtrac's Output

Airtrac is intended to provide continuous information about all aircraft in the given region of airspace. The information may include: a complete track history of all aircraft in the monitored region based on fused data from all radar sites, a classification of all aircraft based on their behavior (e.g., commercial, military, private, smuggler), and an intelligent prediction of the future flight paths and actions of observed aircraft. The current stage of Airtrac provides the first type of information.

2.3. Airtrac's Organization

The Airtrac system is composed of three major modules: Data Association (completed), Path Association (the main focus this paper), and Path Interpretation (yet to be developed). As shown in Figure 2.1, each module takes as input the output from the previous module. Airtrac, then, can be viewed as an application that employs several distinct levels of abstraction and reasoning leading to its final output.

¹ Basically, this algorithm predicts, based on a simple linear projection of observation points already received for that track, an area where the next observation for a track should appear.

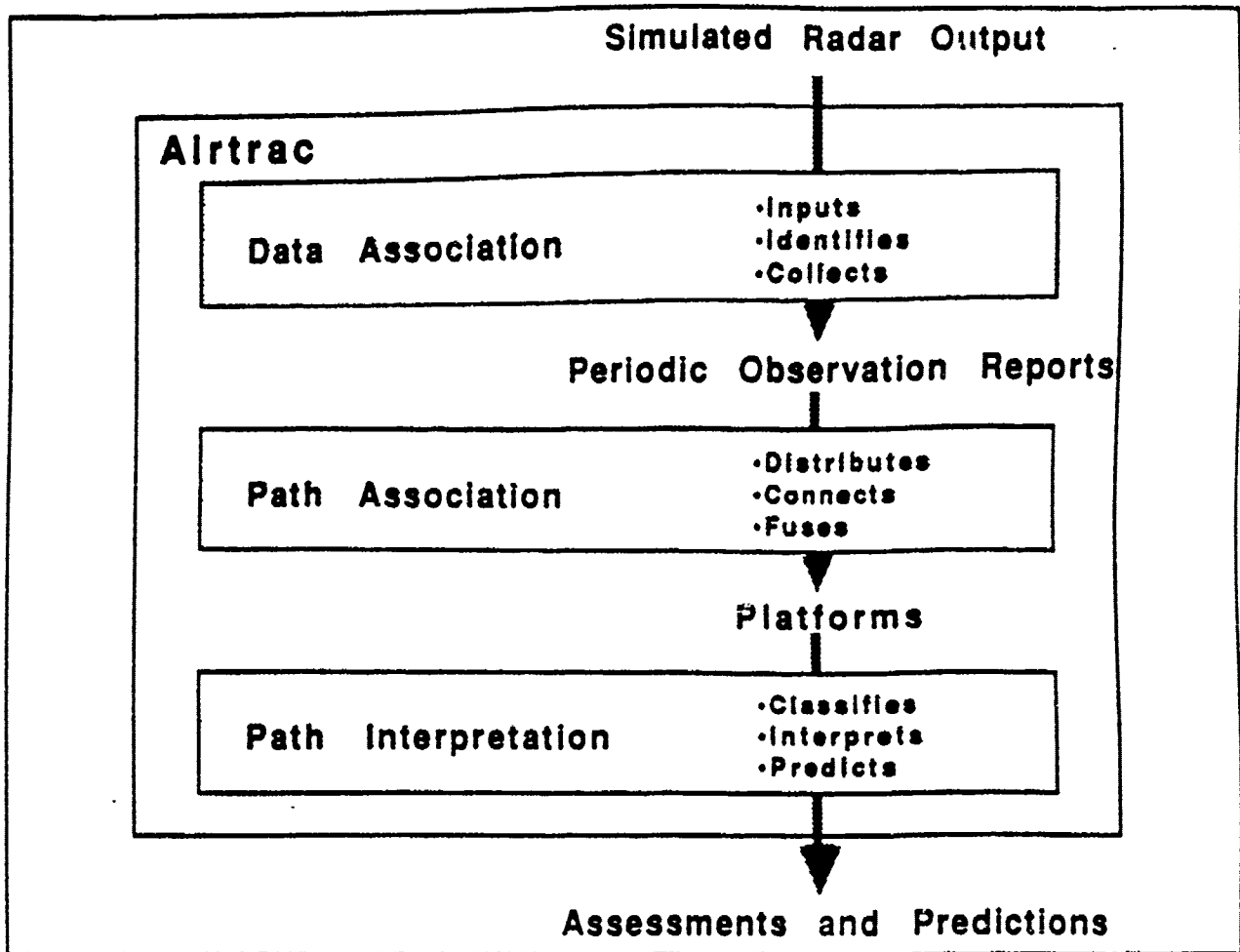


Figure 2.1 Airtrac's modules' interaction and functionality

2.3.1. Data Association

The Data Association (DA) module was completed in June of 1987 by Russell Nakano and Masafumi Minami [5].

The primary function of DA is to accept and process, at scantimes, RTR's from all radars in the given region. It identifies and collects together, in time-ordered sequence, all RTR's which belong to the same aircraft track, i.e. all RTR's which have the same *track ID*. Periodically, DA abstracts the individual RTR's it has gathered for a particular aircraft track into a *Periodic Observation Record* (POR), and forwards it to the Path Association module

for further processing. A track may consist of several POR's. A POR represents a regular portion of an aircraft's flight path as seen from a single radar.

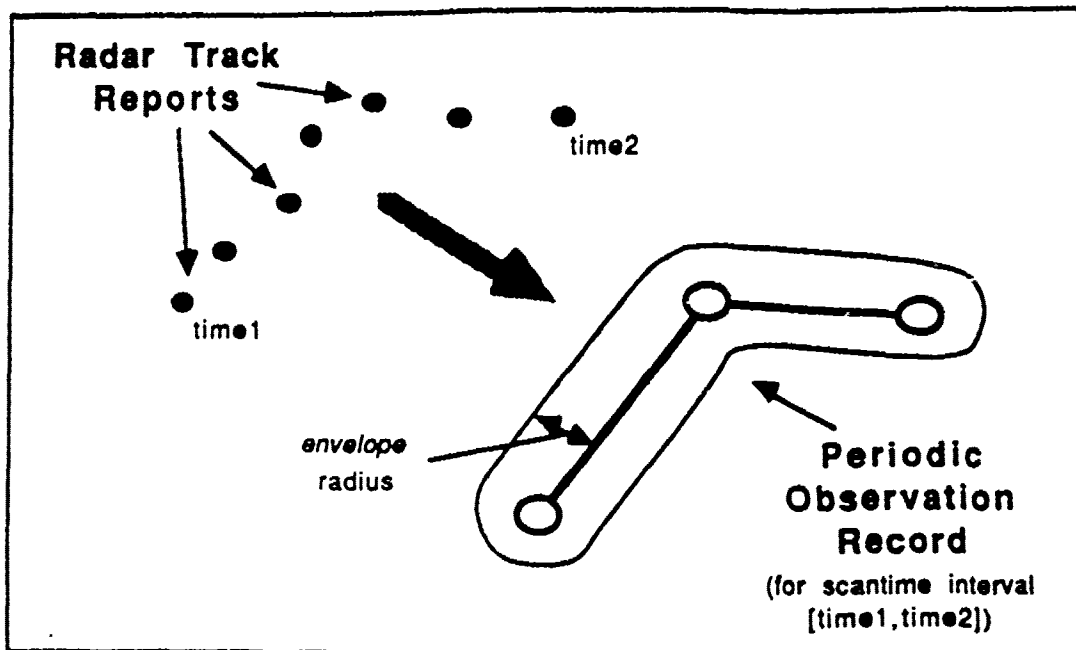


Figure 2.2 How Data Association creates a Periodic Observation Record

The abstraction process in DA is depicted in Figure 2.2. DA creates a POR by fitting one or more line segments, or *line estimates*, through the RTR coordinates.¹ More than one line segment in a POR reflect large changes in course executed by the aircraft. The error *envelope* radius of a POR is conservatively calculated so as to completely contain the *position covariances* of each of the RTR's. The complete information contained in a POR is listed in Table 2.2.

One notable attribute of a POR is its *status*, a keyword that DA assigns to a POR indicating its temporal position within an aircraft track. A status of *create* means that the POR is the first (in data time) of a track. Similarly, an *inactivate* status signals the last POR of a track. A POR with a status of *update* is part of a continuing track, neither first nor last. A status of *create-*

¹ A line estimate is represented as a sequence of its endpoints and corresponding error radius. However, to make it more intuitive, a line estimate in this paper is depicted as a full line with an error envelope or a full line with error radii at its endpoints.

Table 2.2 POR Information

A Periodic Observation Record contains the following:

<i>track ID</i>	the identifier of the aircraft track assigned by the radar to the RTR's in this POR
<i>radar ID</i>	the identifier of the radar site
<i>status</i>	the observation status of this POR; one of <i>create</i> , <i>update</i> , <i>inactivate</i> , or <i>create-and-inactivate</i>
<i>aircraft type</i>	the type of the aircraft
<i>report start time</i>	the beginning scantime of the POR period covered by this POR
<i>report finish time</i>	the end scantime of the POR period covered by this POR
<i>actual start time</i>	scantime of the earliest RTR of the track within this POR
<i>actual finish time</i>	scantime of the latest RTR of the track within this POR
<i>line estimates</i>	the error envelope radius and the sequence of line segments fitted through the RTR's for this POR
<i>velocity vectors</i>	velocities (V_x V_y) for the aircraft at the beginning and end of the POR

and-inactivate means that the POR represents a whole aircraft track.¹

The POR *period* is the length, in scantime units, of a POR generated by DA, i.e., the interval of time in which RTR's for a track are processed and abstracted into a POR.² The POR period is uniform across all POR's in the system. Furthermore, the beginning and ending times of POR's, *report start times* and *report finish times*, are synchronized. If the first RTR of a track is received *during* a POR period (the POR's status is *create*), then the *actual start time* of

¹ From here on, a *:create* POR means a POR whose status is either *create* or *create-and-inactivate*. Likewise, an *:inactivate* POR means one with status *inactivate* or *create-and-inactivate*.

² In this experiment the POR period is 5 scantimes or 50 data time units.

the POR may be different than the *report start time*. The same applies to the *actual finish time* and the *report finish time* of an *inactivate* POR. For *update* POR's the actual and report begin and end times are the same.

Nakano and Minami showed that the quantitative performance of DA improves significantly with additional (up to 100) processors. The quality of the results can be maintained in spite of a high degree of problem decomposition and highly overloaded input data conditions.

2.3.1.1. Data Association Simulator

Due to system and input-output incompatibilities, the real Data Association module was not used together with the Path Association module [7]. The POR's are generated, instead, by an object called the *Data Association Simulator (DAS)*.

2.3.2. Path Association

Most of the functionality of the Path Association (PA) module was developed by Alan Noble and Chris Rogers during the 1987-1988 academic year [7].

The functionality of PA can be described in three stages: Distribution, Connection, and Fusion, as depicted in Figure 2.3. In *Distribution*, POR's of the same aircraft track and reported from the same radar site are grouped together into objects called *Flight Path Segments (FPS's)*. In *Connection*, all FPS's reported from the same radar site that seem to belong to the same aircraft's flight path are associated together into objects called *Observed Flight Paths (FPO's)*. Finally in *Fusion*, FPO's reported from different radar sites that appear to be equivalent representations of a single aircraft's flight path are fused into objects called *Platforms (P's)*.

It is important to note that as input POR's come to the system, FPS's, FPO's, and P's *grow together* over time, i.e. the system does not wait for all data to be available before performing any of the three stages. For example, Distribution immediately creates an FPS once a POR of a new track is available. Distribution does not wait for all POR's of the track. Similarly for Connection and Fusion.

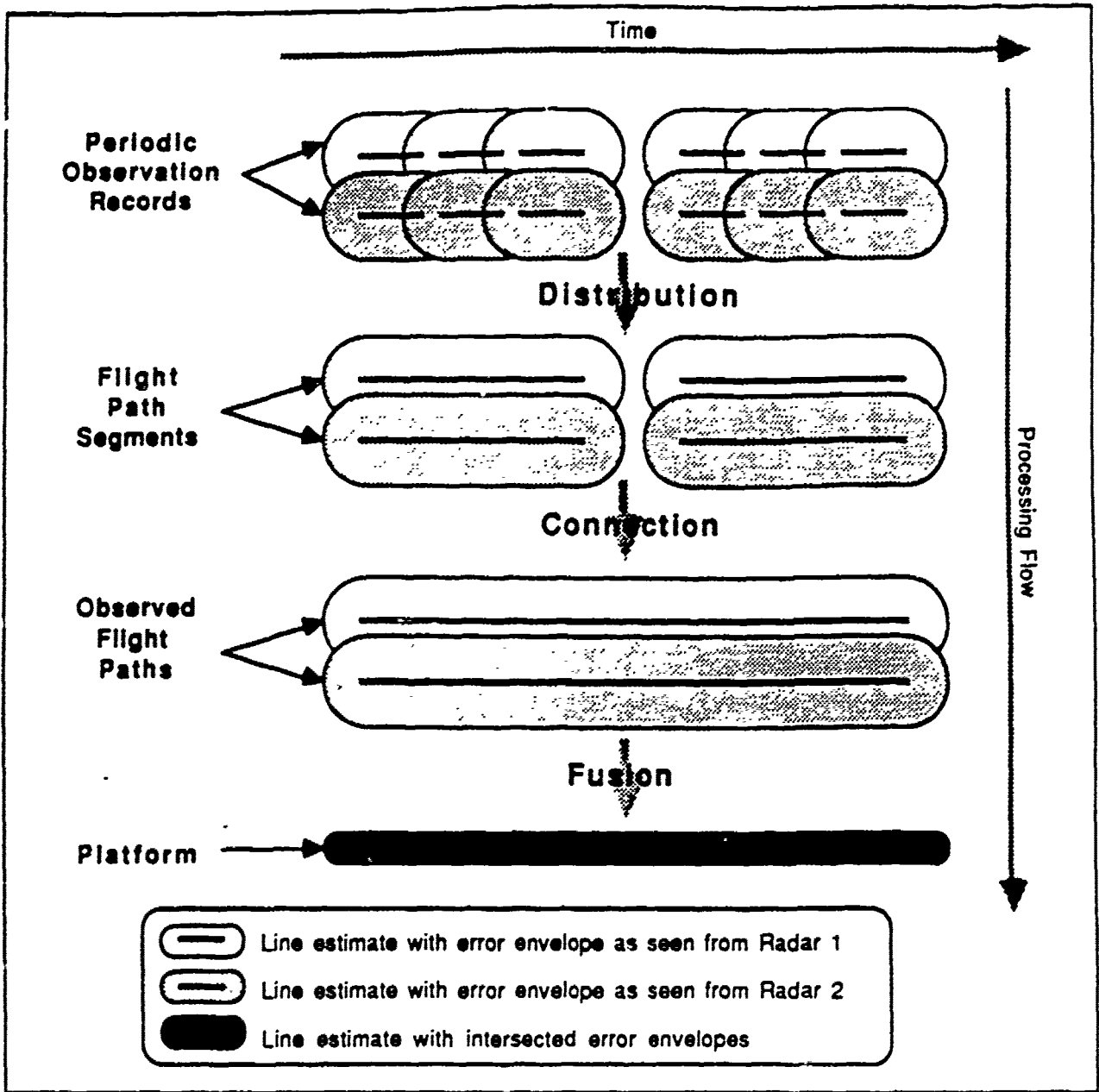


Figure 2.3 Distribution, Connection, and Fusion in Path Association

Platforms represent hypothesized aircraft in the real world as seen from one or more radar sources. As shown in Table 2.3, a Platform incorporates all information about a single aircraft available through its individual FPO's. Platforms are dynamic entities that are continuously being created, updated, and terminated to reflect the rapidly changing state of the monitored region, in which aircraft are constantly appearing and leaving.

Table 2.3 Platform

The information in a Platform includes the following:

<i>aircraft type</i>	the type of the aircraft
<i>FPO attributes</i>	list of names and attributes of all Observed Flight Paths that make up the P
<i>composite flight path</i>	the composite flight path computed from a best-fit of all line estimates of individual FPO's of the P

A detailed description of the functionality and architecture of PA appears in Section 3.

2.3.3. Path Interpretation

The final module of Airtrac, which is not yet implemented, is Path Interpretation (PI). PI is intended to analyze and interpret information contained in Platforms generated by the PA module. PI will provide continuous and real-time assessments and predictions about the observed aircraft represented by these Platforms. These assessments and predictions will include: a classification of all aircraft based on their behavior (e.g., commercial, military, private, smuggler), and a prediction of the future flight paths and actions of the observed aircraft.

2.4. Airtrac's Implementation

The Airtrac application is implemented on a simulated parallel architecture called CARE [3]. CARE (Concurrent ARchitecture Emulator) is a distributed-memory, asynchronous message-passing architecture. CARE models 1 to 1000 processor-memory pairs, or *sites*¹, communicating via a high-speed network. Each site operates on its own instruction stream.

¹ Throughout this paper the term *site* is used to describe a CARE processing element.

asynchronously with respect to other sites. Message delivery between sites is reliable (never lost), but messages are not guaranteed to arrive in the order of origination.

CARE is simulated using a general, event-driven, highly-instrumented system called SIMPLE [4]. SIMPLE is written in Zetalisp and runs on a Texas Instruments Explorer Lisp machine.

Airtrac is written in LAMINA with ELMA. LAMINA is the basic language interface to CARE and consists of Zetalisp with extensions that provide primitive mechanisms and language syntax for expressing and managing concurrency and locality [2]. Three computational styles are supported: functional, shared-variable, and object-oriented. All three are based on the notion of a stream, a data type which represents the promise of a potentially infinite sequence of values.¹

As in other object systems, objects in object-oriented LAMINA (hereafter referred to simply as LAMINA) encapsulate state (instance variables) and behavior (methods). Methods are invoked by message sending. But unlike the case of sequential systems, this involves transmitting a packet containing the message from one LAMINA object to another, typically on a different site. Message sending is non-blocking and the time required for communication is thus visible to the LAMINA programmer. Methods run atomically within processes which are restartable but not resumable.² An object and its methods can be considered a non-nested monitor; exclusion is guaranteed by the fact that only one method is ever scheduled to run at a time, and then runs to completion. The time required to create a LAMINA object is also visible to the programmer.

ELMA (Extended Lamina for Memory-management Applications) is a high-level parallel programming interface to CARE based on LAMINA [6]. ELMA is a specialized interface for applications which involve extensive dynamic object creation and deallocation and require some

¹ LAMINA's predecessor, CAOS [1], was based on the notion of a future, the promise of a single value resulting from a computation. It was observed, however, that communication between objects was fairly regular; a given object, having communicated with another, invariably communicated with that same object again. The stream notion captures this behavior much more naturally, and was thus chosen as the basic data type for LAMINA. In LAMINA, a future is the special case of a stream with only one value.

² There is also a more expensive resumable cousin.

form of memory management. Its syntax and constructs facilitate programming in the object-oriented style at a higher level than LAMINA.

3. Path Association

Path Association (PA) is the second of the three-module Airtrac system. PA takes as input *Periodic Observation Records* (POR's), which are periodic abstractions of radar signals generated by the Data Association (DA) module. PA processes these POR's and generates *Platforms* (P's), the hypotheses of real-world aircraft passing through the given region of airspace. The following subsections describe the functionality and architecture of PA.

3.1. PA's Functionality

The functionality of PA can be described in three stages: Distribution, Connection, and Fusion.

3.1.1. Distribution

This is the simplest of the three stages. This stage involves the distribution of POR's of the same, unbroken aircraft track and reported from the same radar site into dynamic objects called *Flight Path Segments* (FPS's) by manager objects¹ called *Flight Path Managers* (FPM's).

An FPM handles only POR's of a particular aircraft type reported from a particular radar site. Using these two invariants, the functionality of an FPM is *replicated* into the number of distinguishable aircraft type times the number of radar sites. So, for example, in a *scenario* (simulated input data) with three aircraft types and three radar sites, up to nine distributions can be performed in parallel.

¹ Managers in Airtrac are objects allocated at initialization time, and are typically responsible for tasks involving dynamic objects. Those tasks include maintaining *free pools* of dynamic objects, creating from and deallocating dynamic objects into the free pools, synchronizing different processes, and coordinating searches. For more details, see [7].

When an FPM receives a POR from DA, the FPM creates an FPS for the POR if the POR is part of a new track. Otherwise, the FPM forwards the POR to its associated FPS. The FPM does not wait for all POR's of the track to create an FPS for it. The FPM registers all FPS's it has created for the input POR's.¹ If the POR represents the beginning or the end of a track, i.e. a *create* or *inactivate* POR, the FPM tells the FPS to report to its associated Connection manager, called a *Flight Path Connector* (FPC), to start the Connection stage.²

The information contained in an FPS is listed in Table 3.1. An FPS is *active* if it continues to receive POR's, *inactive* otherwise.³ The *FPO parent* is the name of the Observed Flight Path this FPS and other FPS's are associated with to represent the complete flight path of an aircraft as seen from one radar site.

3.1.2. Connection

The Connection stage is needed because an aircraft's complete flight path may be *broken* into several tracks due to a radar's tracking errors, radar shadows⁴, or the aircraft's sharp maneuvers. In any case, the radar fails to recognize the continuation of an ongoing track, and instead treats the continuation as part of a new track. The goal of this stage is to connect, using some heuristics, those broken tracks that are actually parts of the same flight paths. That is, to connect FPS's into *Observed Flight Paths* (FPO's).

¹ In a real, instead of simulated, continuous system, there must be a mechanism to get rid of old dynamic objects. Airtrac never deallocates objects during its execution, since they are needed for post-run analysis. However, Airtrac uses some heuristics to keep its *conscious* knowledge, and the amount of reasoning involved, from growing exponentially. This is not dealt in Distribution, because the dynamic FPS's are not involved in any reasoning. This will be an issue, however, in Connection and Fusion.

² The reason for this double-directed connection search is given in the next section.

³ Normally an FPS stays *active* if it continues to receive *update* POR's, and becomes *inactive* if it receives its *inactivate* POR. However, since some messages may be out of order, this is not always true.

⁴ A radar shadow is part of the monitored region not observable from a radar site due to some obstacles such as mountains.

Table 3.1 Flight Path Segment

A Flight Path Segment contains the following information:

<i>track ID</i>	the identifier assigned to this track by the radar
<i>radar ID</i>	the identifier of the radar site from which the track is being observed
<i>status</i>	track status; one of <i>active</i> or <i>inactive</i>
<i>aircraft type</i>	the type of the aircraft
<i>line estimates</i>	the sequence of PCR line estimates
<i>initial velocity</i>	velocity (V_x V_y) of track at time of creation
<i>final velocity</i>	velocity (V_x V_y) of track at time of termination
<i>FPC</i>	name of Flight Path Connector this FPS reports to or connection
<i>FPO parent</i>	name of Observed Flight Path parent

Connection is coordinated by manager objects called *Flight Path Connectors* (FPC's). The same two invariants used in Distribution, aircraft type and radar site, are used to replicate the functionality of an FPC. This means that an FPC only handles the connections of aircraft of the same type reported from a particular radar site. So there is a one-to-one correspondence between FPM's and FPC's. Although there are up to the number of distinguishable aircraft types times the number of radar sites Connection processes run in parallel, there is only one Connection process at a time per FPC.

An FPC maintains a list of all FPO's it has created. It also keeps a list of FPS's waiting for connections. But since Airtrac is a continuous system, it can only store a limited amount of history. So Airtrac has a *history ring buffer* mechanism to store those FPS's requesting for connections that are created in the latest n scantimes, where n is the length of the buffer. So the number of FPS's in the buffer is limited to the actual number of tracks in any n consecutive scantimes. The buffer is typically full after some initialization time. In that case, if an FPS of a new scantime is added to the buffer, the first set of FPS's in the buffer (the earliest ones), although still connectable with later FPS's, have to be discarded, and the new FPS is added at the

end of the buffer.¹ Otherwise, the FPS is just inserted to the appropriate scantime slot already in the buffer. The length of the ring buffer is set prior to a simulation run. The ring buffer mechanism is not used for the FPO's created, because they are needed for post-run analysis and are not involved in any reasoning in this level. The waiting FPS's, on the other hand, have to be checked one by one for connections. It would be undesirable in terms of quantitative performance to keep a growing and unlimited number of them.

Two FPS's are connected to each other if the later FPS's creation time is within some *connection search interval* from the earlier FPS's termination time, and if the later FPS's creation location lies within the earlier FPS's termination *continuation region*. Connection search interval is a heuristic, set before a simulation run, that limits the time gap between connected FPS's to a reasonable length. The continuation region, as shown in Figure 3.1, is determined by the termination location and velocity of the earlier FPS, the time gap between the two FPS's (Δt), and some performance figures of the aircraft involved.

The process starts when an FPS, a representation of a single track, reports for connection to its corresponding FPC. An FPS requests for a connection when it receives the beginning (*creation*) or end (*termination*) of its track.² Usually only the *creation connection*, i.e. the connection process in which an FPS looks for a connection with a logically earlier FPS, is needed. However, since messages can and do get out of order in CARE, when an FPS, call it FPS-2, searches for a connection with a logically earlier FPS, call it FPS-1, the latter FPS may not be ready for a connection yet. FPS-1 may have yet to receive the end of its track, or in the worst case, it may not yet exist. So the creation connection fails and a *termination connection*, i.e. a connection process in which an FPS looks for a connection with a logically later FPS, has to be performed when FPS-1 receives the end of its track. Consequently every FPS has to check both creation and termination connections.

¹ This assumes that the new scantime is the one after the latest scantime in the ring buffer. If not, more than one set of FPS's have to be discarded to accommodate the time gap.

² The terms *creation* and *termination* are used throughout this paper to denote the beginning and end, respectively, of a track or flight path. When used in conjunction with Airtrac's representations of tracks and flight paths, the terms always denote the properties of the real objects. For example, an FPS's creation time means the tho scantime associated with the beginning of the track represented by the FPS.

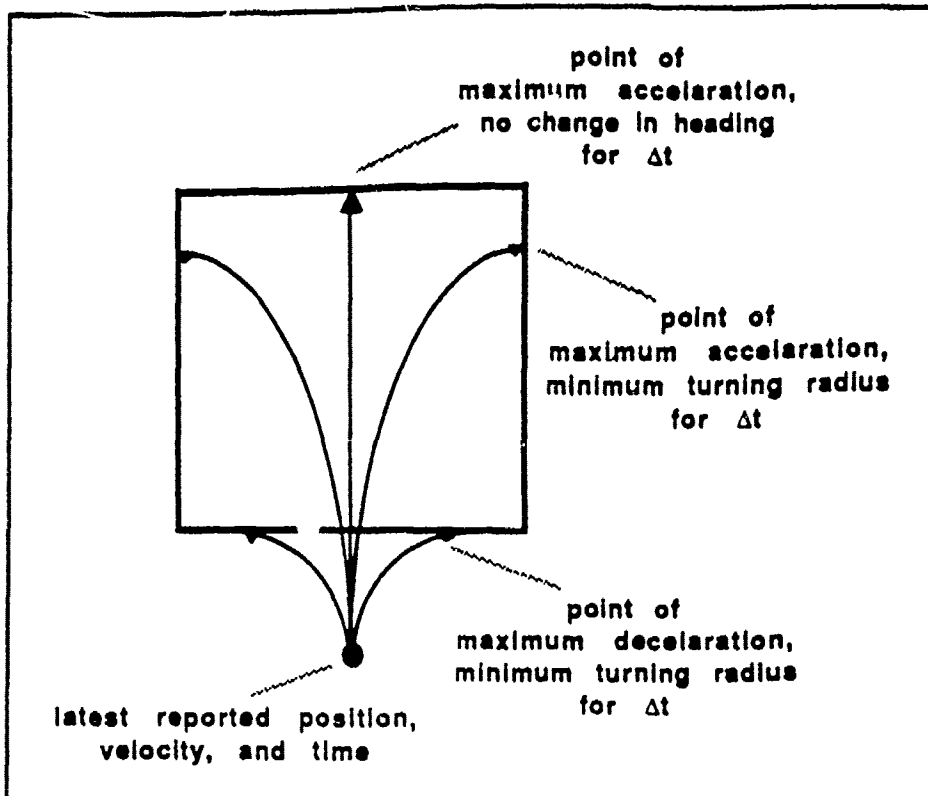


Figure 3.1 Continuation region

Both connection processes are similar. It will be described as a single process unless noted otherwise. After a request for connection from an FPS is received, the FPC adds the FPS to its ring buffer of FPS's reporting for connections, and searches the buffer for possible connection. There are three possible results:

- If no connection is found, a new FPO is created for the FPS.¹ This is usually not necessary in termination connection, since the FPS should have tried creation connection and been associated with an FPO (unless some message disordering happens).
- If there is exactly one connecting FPS, the FPS is associated with the FPO of the connecting FPS.

¹ Right after the creation, the FPO reports for fusion to its Fusion manager, called *Chief Platform Manager* (CPM).

- If there are more than one possible connecting FPS, a new FPO is created for the FPS (if not yet associated with one), and the connection is performed at the FPO, instead of FPS, level. The FPO's of the possibly connected FPS's are linked to each other as to indicate possible connections, and called *connected FPO's*. PA relies on Fusion to resolve this kind of ambiguity.

In Distribution, POR's of continuing tracks are forwarded by the FPM's to the FPS's. As soon as an FPS is associated with an FPO, the FPS also starts forwarding this new information about the track to the FPO. The FPO, in turn, forwards all update information to its associated higher-level object in the Fusion stage.

The information contained in an FPO is listed in Table 3.2. An FPO becomes *inactive* if it is no longer involved in the Connection process, i.e. it fails to make new connections with newly created FPS's after some period (the length of the ring buffer). This usually means that the FPO does indeed represent a complete flight path. *UFP* (Unfused Flight Path) *parent* and *P* (Platform) *parents* are the names of higher-level objects associated with this FPO in the Fusion stage.

3.1.3. Fusion

Fusion is the most complex of the three stages of PA. Fusion, unlike Distribution and Connection, collects information from multiple radar sites to produce a consistent and comprehensive picture of aircraft activities in the monitored region. Fusion is used to resolve some missing data (breaks in a flight path) and ambiguities (multiple connections to the same track) among tracks of a flight path that Connection fails to resolve with information from one radar source only. It is also used to track flight paths that fly across different areas covered by different radars. In this stage, FPC's of different radar sites are fused into *Platforms* (P's).

The Fusion process is coordinated by two layers of managers, *Chief Platform Managers* (CPM's) and *Platform Managers* (PM's). A CPM only coordinates the fusion of aircraft of a particular type. So the number of CPM's is equal to the number of aircraft types. Each CPM is helped by a same number of PM's. The number of PM's per CPM is set at initialization. Unlike Connection, in which there is only one Connection process per manager at a time, in Fusion there are potentially unlimited number of Fusion processes per manager at a time. Fusion takes place in a distributed search method coordinated by the CPM's.

Table 3.2 Observed Flight Path

The information in an Observed Flight Path includes the following:

<i>radar ID</i>	the identifier of the radar site from which the track is being observed
<i>status</i>	flight path status; one of <i>active</i> or <i>inactive</i>
<i>aircraft type</i>	the type of the aircraft
<i>FPS children</i>	list of names of FPS's that make up this FPO
<i>creation connections</i>	list of names of earlier FPO's connected to this one
<i>termination connections</i>	list of names of later FPO's connected to this one
<i>UFP parent</i>	name of associated Unfused Flight Path (if any)
<i>P parents</i>	list of names of Platforms of which this OFP is a part
<i>CPM</i>	name of Chief Platform Manager that handles fusion searches for aircraft of this type

An earlier version of PA had only one layer of managers for Fusion. Preliminary performance evaluation of the system showed that the managers were overloaded. Their task queues were growing and unacceptably long. So an additional layer of managers were introduced to help the overloaded managers with their tasks.

A CPM maintains a list of P's it has created. On the other hand, a PM is totally dependent on its CPM for its knowledge of P's in the system. This is to ensure data consistency among all PM's of a CPM. The CPM does all the creation and registration of P's. The two operations are atomic to make sure that the list of P's is always consistent.

Although *all* created P's are kept in CPM's for post-run analysis, not all of them are involved in the Fusion process itself. This is because Airtrac is a continuous input system, and it must have a mechanism to cope with history. So PA classifies P's into three classes with respect to their history: *active P's* are those P's whose supporting FPO's are still involved in

Connection, *inactive P's* are those P's whose supporting FPO's are not involved in Connection, and *finished P's* are those P's that have been inactive for some *waiting period* set at initialization.¹ The Fusion process only includes the active and inactive P's. This way Fusion is limited only to recent P's. Aircraft come and go within the monitored region, and PA must disregard those aircraft that the system is confident have left the region.

Fusion takes place between an FPO and a P. The line estimates of the FPS's of the supporting FPO's of a P are fused to form a *composite flight path*. When a P is created, its composite flight path consists of the line estimates of the FPS's of the FPO that triggers its creation. An FPO fuses with a P if all of the following conditions are satisfied:

1. The FPO and P intersect in time for at least one scantime.
2. The FPO and P intersect in space for every point in which they intersect in time. The error envelopes of the line estimates and composite flight path are used to check this spatial intersection.
3. The FPO and P do *not* intersect in time at any point for which the P has already incorporated another FPO reported from the same radar. This indicates that the two FPO's represent two distinct aircraft.

An FPO cannot possibly fuse with a P if the second or third condition above is violated. However, the result is uncertain if only the first condition, temporal intersection, is violated. As the FPO and P receive new information about their continuing flight path, they may eventually be able to fuse.

Figure 3.2 gives a before-and-after look at the successful fusion of corresponding portions of an FPO and a P. The resulting new composite point for each pair is a weighted average of the locations of the two points. The weight of a composite point is the number of contributing radars whose reports are already represented in that point. The error radius for each new composite point is computed from the location of the new point and the two intersecting radii. Note that as each new FPO is fused with a P, the composite flight path for that P becomes more defined as the error radii of its constituent points become smaller.

¹ An inactive P can become active again if it is fused with a new FPO that is still involved in Connection. In that case the waiting period is reset. But once it is marked finished, it stays that way.

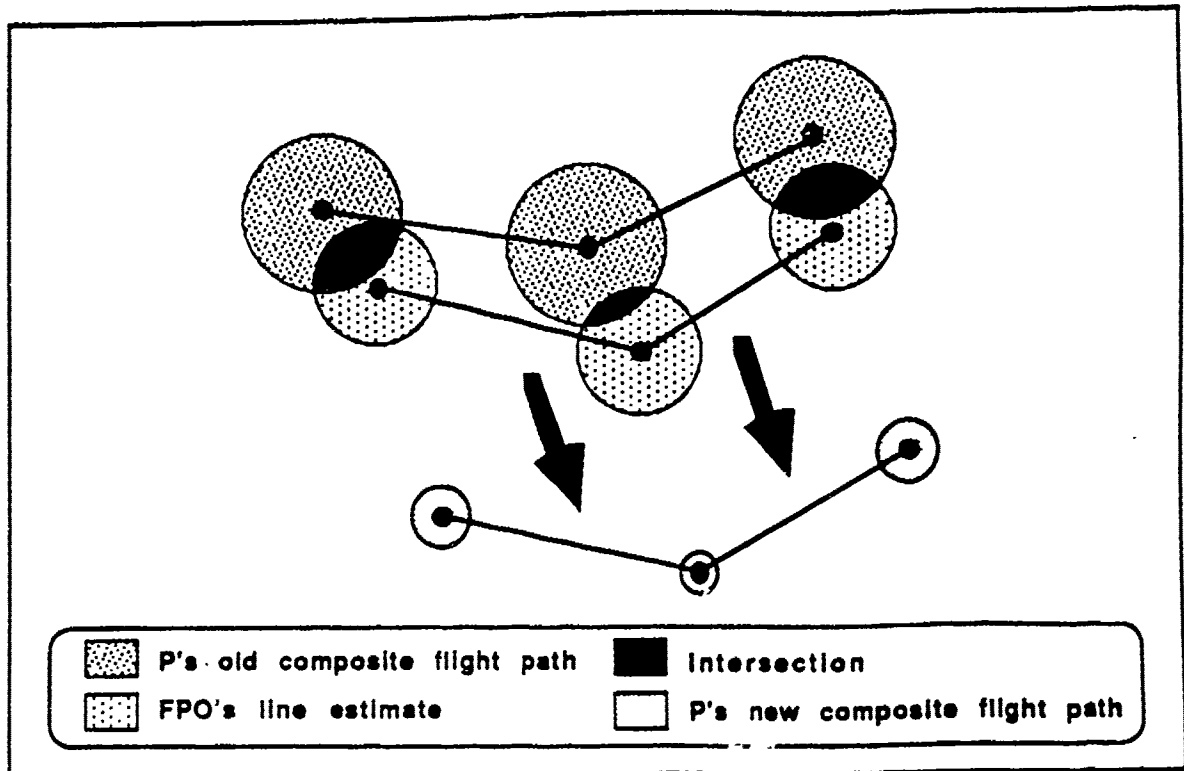


Figure 3.2 Fusing portions of an FPO and a P

The Fusion process starts right after an FPO, a representation of a full flight path, is created. Since the FPO does not duplicate the information contained in its supporting FPS's, the FPO multicasts a message to the FPS's requesting their line estimates for fusion. Instead of waiting for the replies, the FPO spawns a continuation process¹ to collect the replies from the FPS's. Once all FPS replies are received, the FPO continuation notifies its CPM that it is ready for fusion. The CPM forwards the fusion request along with its list of active and inactive P's to one of its PM's in a round-robin fashion. The PM then broadcasts a message to the active and inactive P's asking them to try to fuse with the FPO. To collect the P replies, the PM spawns a continuation. Each P checks whether the FPO can fuse with its composite flight path, and relays the result to the PM continuation. There are three cases:

¹ A continuation of a method occurs in the context of the object executing the method. The method which spawns the continuation finishes normally. The continuation executes each time values are received on specified input streams. For more details on the continuation mechanism in LAMINA, see [2].

- If one or more P's can fuse with the FPO, each of the P's merges the FPO into its composite flight path. If more than one P fuse with the FPO, this is an ambiguous case. Fusion relies on the future data of the continuing flight paths to resolve this kind of ambiguity.
- If no P can possibly fuse with the FPO, the PM continuation forwards the result to the CPM. The CPM has to check the FPO for fusion with P's created *during* the current fusion process. Since the CPM (usually) coordinates more than one fusion process at a time, P's may be created as results of the of the *other* fusion processes. The FPO may very well fuse with one of these recently created P's. The maximum number of this *FPO fusion retries* is set at initialization. If after so many retries no P can fuse with the FPO, a new P is finally created for it.¹
- If no P can fuse with the FPO at the moment but some P's are uncertain because the FPO and the P's do not intersect in time, the PM continuation tells the PM² to create an *Unfused Flight Path (UFP)* for the FPO. It is uncertain at this point if the UFP can fuse with an existing P or if it needs its own P. Fusion once again relies on the future data of the continuing flight paths to resolve this ambiguity. Every time the UFP receives a new piece of information about the flight path, it tells the CPM to check for fusion again with the uncertain P's from the previous fusion process, and with new P's created since then. The limit of this *UFP fusion retries* is also set at initialization. The result of this UFP fusion retry can be one of the following:
 - If one or more P's can finally fuse with the UFP, the UFP is no longer needed and deallocated.

¹ A probably more powerful mechanism involves *variable* FPO fusion retries. The maximum number of retries is allowed to vary within a certain range. This solves a probable case in which two FPO's that should fuse into each other are involved in the same *n*th retry at the same time. Without variable maximum retries, the two FPO's will likely end up as two different Ps.

² Since no manager registers the UFP's created, a PM, instead of a CPM, can create them without jeopardizing any data consistency among the CPM and PM's.

- If no P can fuse with the UFP but some P's are still uncertain, the UFP waits for new data for its next fusion retry.
- If no P can possibly fuse with the UFP, or if the limit of the UFP fusion retries is exceeded, the UFP tells the CPM to create a P for its FPO, and deallocates itself.

As soon as an FPO is associated with a P, the FPO starts forwarding new information about the continuing flight path to the P. (Similarly if the FPO is associated with a UFP as described in the previous paragraph.) The P immediately tries to fuse the new information, in the form of line estimates, to its composite flight path through the same fusion process described above. The P has to make sure that all its supporting FPO's remain consistent with one another. If the FPO's new line estimate fails to fuse with the P's composite flight path, the FPO is *split* from the P. A platform split happens when two or more airplanes fly across the monitored region so close that their early flight paths are fused into one P. But when they no longer fly so close to each other, their later flight paths are no longer fusable, causing the P to split. As a result, all of the P's supporting FPO's have to be disassociated from the P, and the P is deallocated. If the FPO's are still associated with one or more other P's, they do not have to do anything else. In fact, the platform split has resolved (part of) the ambiguity mentioned earlier when an FPO is associated with more than one P. (See the first case of the fusion process.) However, if the FPO's are not associated with any other P's, they have to go through the fusion process one more time. This time, the greater amount of line estimates information the FPO's possess usually allows them to unequivocally fuse with the right P's.

P's are the final output of the PA module. They contain the information listed in Table 3.3. The output of PA may still contain some ambiguities. This is not necessarily the fault of poor reasoning on the part of PA; rather, it often has more to do with the incompleteness of the input data. One of the tasks of the Path Interpretation module, the module after PA, would be to resolve the remaining ambiguities present in the P's.

3.2. PA's Architecture

Figure 3.3 presents the manager architecture in the PA system. As mentioned above, the number of managers in PA is determined by the number of distinguishable aircraft types, the number of radars reporting in the monitored region, and the number of PM's per CPM in the Fusion stage. Figure 3.3 shows the relationships among managers in a scenario with 3 aircraft

Table 3.3 Platform

The information in a Platform includes the following:

<i>status</i>	status of the hypothesized aircraft represented by the P; one of <i>active</i> , <i>inactive</i> , or <i>finished</i> .
<i>aircraft type</i>	the type of the aircraft
<i>FPO attributes</i>	list of names and attributes of all FPO's that make up the P
<i>composite flight path</i>	the composite flight path computed from a best-fit of all line estimates of individual FPO's of the P
<i>CPM</i>	the name of the CPM in charge of P's of this aircraft type

types, 3 radars, and 3 PM's per CPM.¹ There is only 1 DAS in any scenario, since the manager is only simulating the functionality of the DA module. There are 9 FPM's and 9 FPC's in this scenario because there are 3 aircraft types and (multiplied by) 3 radars. There are 3 CPM's because of the 3 aircraft types. Finally each CPM is helped by 3 PM's, set at initialization, in a round robin fashion.

Note that the diagram only shows how a manager is related conceptually with another. It does not necessarily imply any control or data flow. For example, the left most FPM in the diagram above handles a particular aircraft type, say Type A, observed from a particular radar, say Radar 1. The FPM is linked to the left most FPC which also handles aircraft of Type A seen from Radar 1. This FPC is then linked to the left most CPM which handles aircraft of Type A observed from *all* radars in the monitored area. The CPM is helped by a number of PM's in handling aircraft of Type A.

Figure 3.3 also shows graphically how parallelism is achieved in PA via pipelining and replication. Each row of managers is essentially a stage in the pipeline. The output of one stage

¹ In fact, all scenarios in this experiment have these characteristics. See Section 4.3.

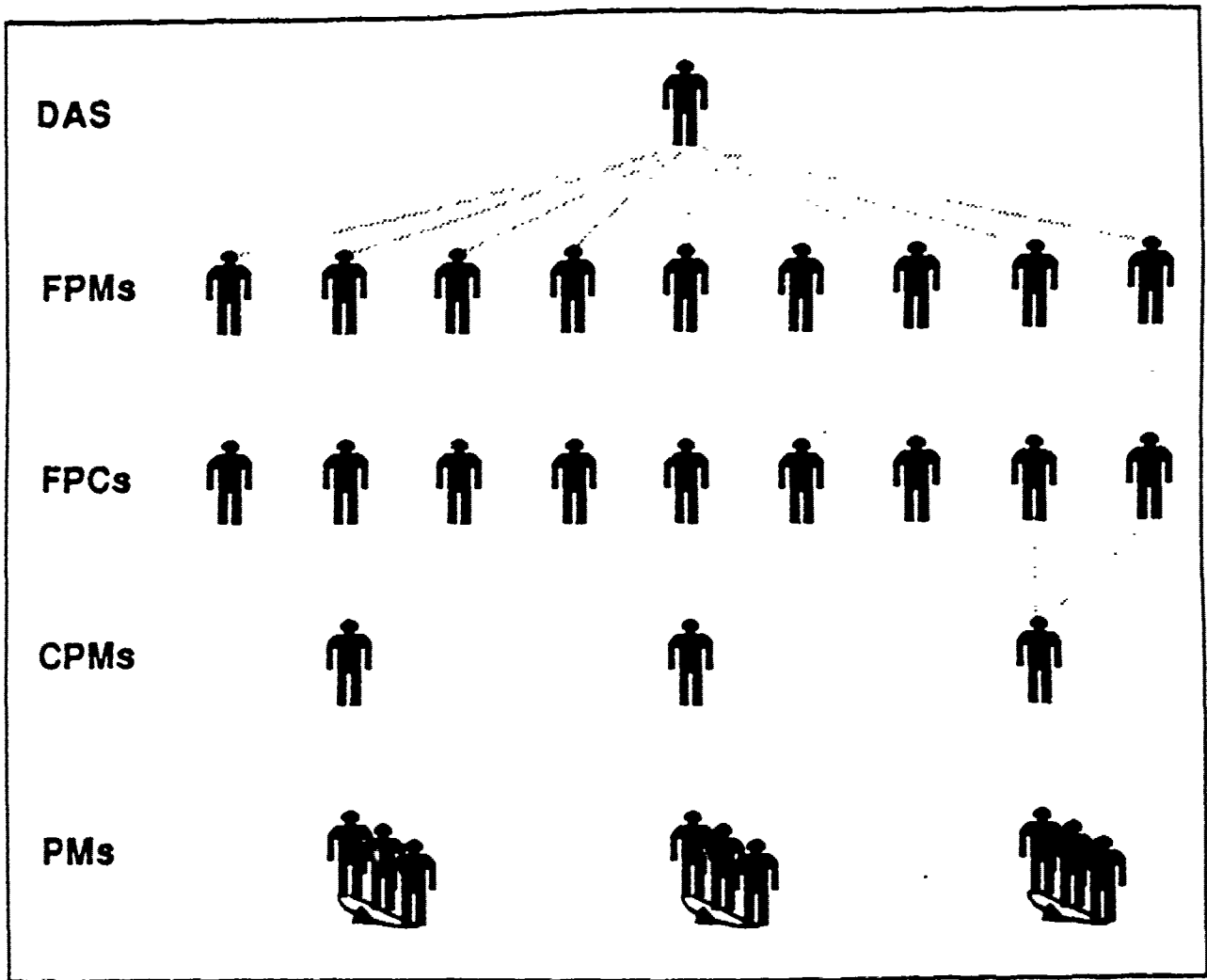


Figure 3.3 Manager architecture involving 3 aircraft types, 3 radars, and 3 PM's per CPM

is the input of the next stage. The number of managers in each row shows how many times the functionality of that stage has been replicated. Managers in a row have the same functionality but deal with distinct parts of the data.

The system architecture of the PA module is shown in Figure 3.4. Pictured are all of the manager and dynamic objects of the system along with the paths of message-passing communication among them, and how they fit into the three stages of PA. Each circle above, with the exception of DAS, represents a *class* of objects, not a single entity. Continuations objects are omitted from the picture and collapsed into the objects that spawn them.

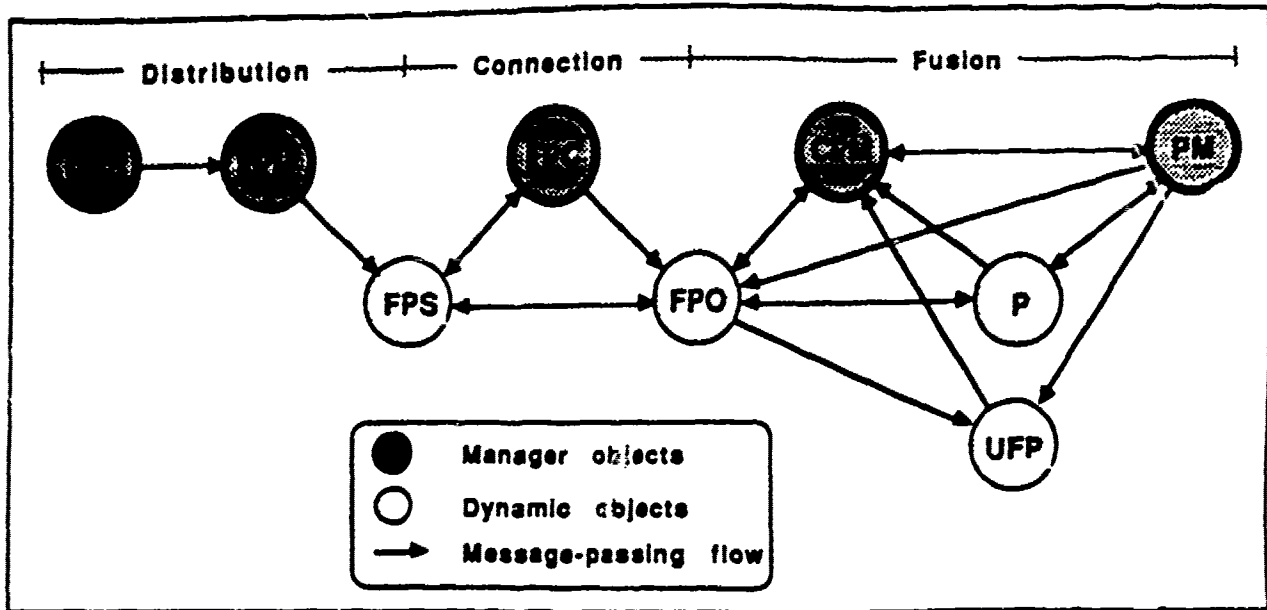


Figure 3.4 System architecture

4. Experiment Design

This section discusses the PA experiment design. The following sub-sections describes the goals of the experiment, the criteria used to evaluate the performance of PA, and the experiment plan. Section 5 will present the results of the experiment.

4.1. Goals and motivation

This experiment is conducted with the following goals:

- To understand how the performance of PA is affected by its parameters. The PA parameters studied in this experiment are:
 - *Free pool lengths* [7], which are the initial and threshold lengths¹ of free pools maintained by the FPM's, FPC's, CPM's, and PM's. An

¹ A threshold length indicates when a free pool needs to be replenished.

FPM maintains a free pool of FPS's, an FPC FPO's, a CPM P's, and a PM UFP's.

- *PM multiplier*, which is the number of PM's per CPM.
 - *FPS history ring buffer length*, which is the length of ring buffer containing FPS's waiting for connections.
 - *Connection search interval*, which is the time gap allowed between connectable FPS's.
 - *Maximum FPO fusion retry*, which is the maximum fusion retries an FPO has to do with P's created during the previous fusion search.
 - *P waiting period*, which is the duration inactive P's remain in the system before being deallocated.
- To understand how the performance of PA across different *grid sizes*, i.e. different numbers of processors, is affected by the frequency and width of input data.
 - To generate possible speedup curves of the performance of PA.¹

4.2. How to Evaluate the Performance of PA

Evaluating the performance of a continuous parallel knowledge-based system such as Airtrac is difficult. The simple approach of timing its execution would not work, since the system is continuous. Furthermore, the performance is usually multidimensional and cannot easily be expressed into a single number. So other forms of measurement are needed to know how the system *keeps up* with its input both quantitatively and qualitatively.

¹ Several speed-up curves are possible due to different performance criteria.

4.2.1. Quantitative Performance

The performance of PA is measured quantitatively in terms of *latency*. Latency is defined as follows:

Latency is the duration between the time when the system receives a datum and the time when it actually uses that datum to assert some fact.

For example, if a POR enters PA at time t , and triggers the creation of an FPS at time $t+x$, then the FPS creation latency at time t is x .

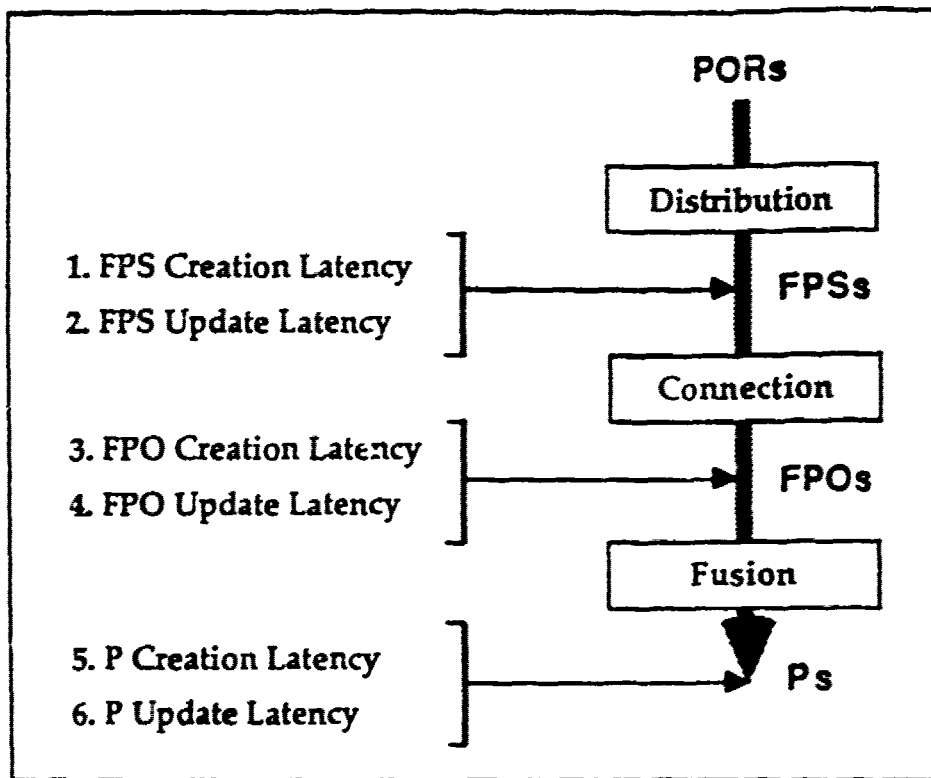


Figure 4.1 The measured latencies in PA

Six types of latencies, as shown in Figure 4.1, are measured in PA; two at each of the three stages of PA. Those latencies are:

1. *FPS Creation Latency*, which is the time between a POR for a new FPS entering the system and being incorporated into an FPS.

2. *FPS Update Latency*, which is the time between a POR for an existing FPS entering the system and being incorporated into that FPS.
3. *FPO Creation Latency*, which is the time between a POR for a new FPO entering the system and being incorporated into an FPO for the very first time.
4. *FPO Update Latency*, which is the time between a POR for an existing FPO entering the system and being incorporated into that FPO.
5. *P Creation Latency*, which is the time between a POR for a new P entering the system and being incorporated into a P for the very first time.
6. *P Update Latency*, which is the time between a POR for an existing P entering the system and being incorporated into that P.

The latencies are recorded in intervals of 5 scantimes, which is the length of POR's. In each interval, the minimum, average, maximum, and the number of reported latencies in that interval are recorded. A reported latency of x at time interval t means that a POR entering the system at time interval t has a latency of x simulation time units.

4.2.2. Qualitative Performance

Measuring the qualitative performance of PA is a harder problem. Ideally, the quality of PA should be measured by the quality of its final output, the P's. That is, by comparing the P's with "ground truth." But this would require reimplementing the PA system serially, and comparing the parallel results with the serial results. Given the size of the system, this would be very time consuming. So *excess ratio* is instead used as a qualitative measurement.

Excess ratio is the number of excess output objects created over the number of the actual objects they represent in the real world.¹

The output objects are the FPS's, FPO's, and P's. The FPS, FPO, and P excess ratios measure the numbers of distribution, connection, and fusion, respectively, that PA misses. For

¹ The number of the actual objects is known since simulated data are used.

example, in a scenario which has 100 FPO's, and each FPO consists of two FPS's, there are 100 connections to do. If PA misses 3 out of the 100 connections, there would be 103 FPO's instead of 100. So the FPO excess ratio is $\frac{103}{100} = 0.03$.

The notion of excess ratio is based on the assumption that PA will not undercreate FPS's, FPO's, and P's, but may overcreate them. In other words, PA will not do distributions, connections, and fusions it should not do, but PA may miss to do some of the distributions, connections, and fusions it should do. This is because PA is very conservative in its reasoning. It is much more likely for PA to create, for example, a new FPO for an FPS rather than connecting that FPS with a wrong FPO.

4.2.3. Performance Evaluation

Now that latencies and excess ratios are available as forms of performance measurement, the question is how to use them so that performance comparison and evaluation can be made.

4.2.3.1. Sustainable Data Rate

Subsequent work refining the ELINT application has led to the notion of *sustainable data rate*:

Sustainable data rate is the maximum data rate for which designated latencies do not increase over time [9].

Speedup is determined by plotting sustainable data rate, rather than latency, versus grid size. This way the different latencies can now be expressed as a single number. However, the definition has some disadvantages:

- The definition does not take into account any latency upper bounds. A sustainable input data rate of, say, 25 Hz, may have a latency of 100 or 500 ms. Under this definition, both cases are considered to have the same level of performance, but clearly they do not.

- The definition fails to include the qualitative aspect of the performance. Unless the system is always and absolutely correct, it is necessary to include the qualitative performance as part of the overall performance evaluation.

Partly to remedy the first disadvantage, Alan Noble and Chris Rogers redefine sustainable data rate as follows:

Sustainable data rate $SDR_{a,b}$ is the input data rate for which absolute latencies are below a threshold a at least b percent of the time [7].

Although this definition introduces a latency upper bound, it also has several important disadvantages:

- The definition may not allow the system to settle down in the beginning of the simulation run, in which the latencies tend to be high. The duration of high latencies in the beginning may exceed the tolerance ($100-b$ percent of the time), but the system may actually sustain the input data rate. As a result, either the result is classified as unsustainable or the a and/or b are compromised to suit to the result.
- The definition fails to detect any increase in latencies over time. The system may actually fail to sustain the input data rate if a longer scenario is simulated.
- Again the qualitative aspect of the performance is not directly tied to the overall performance evaluation.

To remedy the above disadvantages, the following definition of sustainable data rate is instead used:

Sustainable data rate $SDF_{qn,ql}$ is the maximum data rate for which all measured latencies stabilize at or below qn and do not increase over time, and for which the excess ratios of output objects are less than or equal to ql .¹

¹ qn and ql are vectors or matrices.

This definition permits the latencies to be higher than qn in the beginning. It allows the system to settle down or stabilize.¹ But the latencies are not permitted to increase over time. The definition requires all reported latencies, rather than just the average of reported latencies in each time interval, to stabilize at or below qn . If the system keeps up with the input data rate, the deviation of reported latencies in each time interval shouldn't be significant. In addition, some quality attribute ql is attached to the definition and tied to the overall performance evaluation.

4.2.3.2. Capacity

Another way of evaluating the performance of PA is through its *capacity*², which is defined as follows:

Capacity $C_{qn,ql}$ is the maximum number of input data per data time unit, i.e. POR's per scantime, for which all measured latencies stabilize at or below qn and do not increase over time, and for which the excess ratios of output objects are less than or equal to ql .

While sustainable data rate is clearly a simulation-oriented evaluation form, capacity is more a real-world-oriented one. Sustainable data rate evaluates how *frequent* input data can be fed into the system without overloading it. Capacity evaluates how *wide* input data per data time unit can be fed into the system without overloading it.

4.2.3.3. Performance Requirement

Real world systems are built to performance specifications. In PA, qn and ql are vectors, and should be viewed as its quantitative and qualitative performance requirements, respectively. For the purposes of this experiment, qn and ql will have the following forms:

¹ The latencies are allowed to be higher than qn for the first 20% of the scenarios described in Section 4.3.

² This idea originated from Max Hailperin [7].

$$qn = \begin{bmatrix} \text{FPS-CL} \\ \text{FPS-UL} \\ \text{FPO-CL} \\ \text{FPO-UL} \\ \text{P-CL} \\ \text{P-UL} \end{bmatrix}$$

where *object-CL* means *object* Creation Latency and *object-UL* means *object* Update Latency, and

$$ql = \begin{bmatrix} \text{FPS-ER} \\ \text{FPO-ER} \\ \text{P-ER} \end{bmatrix}$$

where *object-ER* means *object* Excess Ratio.

The values for *qn* and *ql* vary with different performance requirements. The variation in values will affect the sustainable data rate and capacity of the system. This in turn may generate different speedup curves of the system.

4.3. Scenarios

A *scenario* is simulated input data fed to the PA system. A scenario must be long enough, in data time units, to enable the system to settle down into steady-state behavior. It must also be wide enough, i.e. contain enough simultaneously observed aircraft as manifested in POR's per scantime, to provide sufficient data parallelism and thus opportunities for parallel computation.

Seven different scenarios are used in this experiment. They basically have the same characteristics, except their widths, which are expressed in the numbers of aircraft, or POR's per scantime, in the scenarios. The common characteristics are:

- The size of the monitored region is 300,000 by 300,000 data area units.
- There are 3 radar sites in the region, known as Radars 1, 2, and 3.
- The radars can distinguish 3 aircraft types, known as Type: A, B, and C. A scenario contains the same number of aircraft of each type.
- The length of each scenario is approximately 4,000 data time units.

- The average length of full flight paths is 750 data time units. Each flight path consists of, on the average, 2.65 tracks.
- The radars fail to detect approximately 1% of all tracks.
- The radar scan period, i.e. 1 scantime, is 10 data time units.
- The POR period is 5 scantimes.

The seven scenarios and their specific characteristics are:

- C-40 has 40 aircraft in 2,039 POR's (5.08 POR's/scantime).
- C-65 has 65 aircraft in 3,277 POR's (8.11 POR's/scantime).
- C-90 has 90 aircraft in 4,574 POR's (11.76 POR's/scantime).
- C-110 has 110 aircraft in 5,558 POR's (14.00 POR's/scantime).
- C-130 has 130 aircraft in 6,391 POR's (15.74 POR's/scantime).
- C-150 has 150 aircraft in 7,539 POR's (18.61 POR's/scantime).
- C-170 has 170 aircraft in 8,518 POR's (20.93 POR's/scantime).

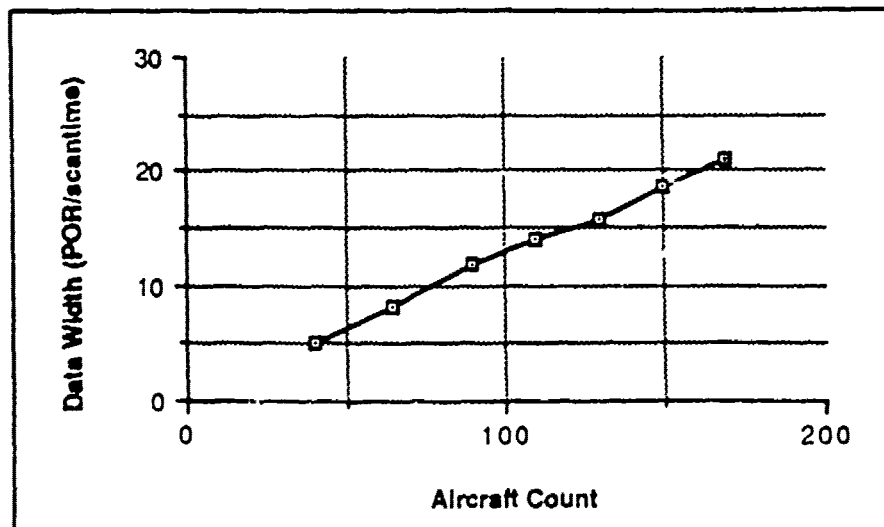


Figure 4.2 Scenarios and their widths

The scenarios and their widths are shown in Figure 4.2. The C-130 scenario is the main scenario in this experiment. More details on the scenarios can be found in Appendix A1.

4.4. Experiment Plan

The following sub-sections discuss the steps to achieve the experiment goals described in Section 4.1.

4.4.1. Generating a Standard

In order to investigate the effects of the parameters of PA, a standard, with which other experiment results are compared, is needed. The C-130 scenario is used for this purpose. It will be simulated on grid size 128 with an input data rate of 20 Hz, i.e. data come every 50 ms into the system. The values assigned to the parameters of PA are:

- Free pool lengths: 5, 4; 5, 4; 5, 4; 2, 1¹
- PM multiplier: 3
- FPS history ring buffer length: 10
- Connection search interval: 5
- Maximum FPO fusion retry: 2
- P waiting period: 120 ms

These values may later be changed if it turns out that other values can yield better performance. Section 4.4.3 discusses this subject.

¹ The first pair is the initial length and threshold of the FPS free pool maintained by an FPM. The second pair is of the FPO free pool maintained by an FPC, the third pair is of the P free pool maintained by a CPM, and the last pair is of the UFP free pool maintained by a PM.

4.4.2. Varying the Parameters of PA

To investigate the effects of the parameters of PA, the value of each parameter will be varied individually, while the other parameters are kept constant. Each result is then compared with the standard generated above. Changes in the quantitative and qualitative performance will be observed. As in the previous step, the C-130 scenario will be used and simulated on grid size 128 with an input data rate of 20 Hz (unless otherwise noted). The values to be used for each parameter are as follow:¹

- Free pool lengths: 0, 0; 0, 0; 0, 0; 0, 0, **5, 4; 5, 4; 5, 4; 2, 1, 10, 8; 10, 8; 10, 8; 2, 1, 15, 12; 15, 12; 15, 12; 2, 1²**
- PM multiplier: 1, 2, 3, 4
- FPS history ring buffer length: 1, 5, **10**, 15, 20, 50
- Connection search interval: 1, **5**, 10, 20, 50
- Maximum FPO fusion retry: 0, 1, **2**, 3, 4
- P waiting period: 0, 80, 100, **120**, 140, 1,000, 6,000 ms

4.4.3. Generating a Revised Standard

The results from the previous step may suggest that some parameter values yield better performance than the standard. So a revised standard will be generated with these new values. These values will be used in the next steps of the experiment. This step assumes the following:

- The combined effects of the new parameter values on performance are at least as good as the individual effect of each of the new parameter values.
- The new parameter values also yield better performance when used with different scenarios on different grid sizes with different input data rates.

¹ The standard values of the parameters are in italic bold.

² The length of the UFP free pools is kept low since UFPs are rarely created.

4.4.4. Varying the Frequency and Width of the Input Data

Four grid sizes, 36¹, 64, 128, and 256, will be used to study the effects of the frequency and width of the input data across different grid sizes. The C-130 scenario will be used to investigate the effects of the input data rate on the performance. Five different input data rates will be used for each grid size to generate latency and excess ratio vs. input data rate curves. The input data rates used on each grid size should find the knees of the speedup curves.

To study the effects of the width of the input data, different scenarios with different numbers of PCHs per scantime will be used. Four to six out of the seven scenarios will be simulated on each grid size with an input data rate of 20 Hz. The results would be latency and excess ratio vs. width of input data curves. The different scenarios used on each grid size should also find the knees of the speedup curves.

4.4.5. Generating Possible Speedup Curves

An SDR-based or Capacity-based speedup curve is generated by the following procedure:

1. Specify a quantitative and qualitative performance requirement,
2. Find the corresponding sustainable data rates or capacities for all grid sizes from the curves generated in the previous step, and
3. Plot the grid size (x-axis) vs. sustainable data rate or capacity (y-axis) curves.

Different speedup curves may be generated by the two different evaluation approaches, the SDR and Capacity. Furthermore, tightening or relaxing the performance requirement will very likely produce different speedup curves. So multiple, instead of a single, speedup curves can be generated.

¹ Since managers have dedicated sites (Section 4.5), grid sizes 32 and smaller are too small. The total number of managers is $22 + (3 * n)$, where $n \geq 1$ is the PM multiplier.

4.5. Load balancing

The goal of load balancing is to distribute objects over CARE sites such that the work they do is as evenly balanced as possible. A preliminary PA experiment shows that the performance of PA improves significantly when all of its manager objects have dedicated sites [7]. So, in this experiment, every manager has its own site and does not share it with any other objects in the system. The same modified random load balancing used by Data Association [5] is used to place the dynamic objects in PA. The scheme essentially involves random selection from the set of all sites excluding those used by managers (providing the multiprocessor is large enough). This is reasonable since there is no way of knowing a priori whether any given dynamic object will be busier than another. In fact empirical evidence suggests that in the absence of such load knowledge, random allocation is optimal. Excluding dynamic objects from manager sites works well, although it is likely that some managers do not actually require dedicated sites.

4.6. Experiment Repeatability

Simulation results on CARE are not always repeatable due to its non-deterministic nature. This is especially true when some random load balancing is involved. Some of the simulation runs will be repeated to verify the results, in which case the mean of the results would be reported. However, time constraint prohibits the extensive verification of all results.¹

5. Experimental Results

This section presents the results of the experiment done to understand the effects of the PA parameters, and of the frequency and width of input data on the performance of the system. Some possible speedup curves of the PA system are presented at the end.

¹ The simulations done in this experiment took anywhere between 3 to 11 hours.

5.1. The Effects of the Parameters of PA

The following sub-sections describe the effects on performance of each of the six PA parameters studied in this experiment. For each parameter, two main graphs are presented: one contains the latency curves, i.e. the quantitative performance curves, and the other one contains the excess ratio curves, i.e. the qualitative performance curves. The latency curves represent the highest latency reported. The FPS Excess Ratio is not included in the graphs because it is always 0, i.e. PA never misses any distribution tasks.

5.1.1. Free Pool Lengths

One of the goals of implementing a free pool mechanism in PA is to reduce dynamic object creation latency [7]. The hypothesis is that by creating a dynamic object before it is needed, only an initialization, instead of a more expensive object creation, is required when the object is needed. Thus, by cutting the cost of creating objects, the free pool mechanism should reduce the object creation latency.

Figure 5.1 shows how the latencies and excess ratios are affected by the length of free pools. The x-axis shows the initial length of the free pools, except for the UFP free pools which always have the initial length of 2, unless when they are initialized empty. The threshold length is 80% of the initial length, except for the UFP free pools which always have the threshold length of 1, unless when they are always kept empty.

Contrary to the hypothesis, the potential reduction in object creation latencies never materializes. Empty free pools are just as good as long ones. Empty free pools do not cause the FPS, FPO, and P creation latencies to be higher than usual. This is because the free pool mechanism as is used now is not fully exploited. One main advantage of free pool is its ability to recycle objects. Recycling saves time and space. But there is no real recycling in the PA system. Dynamic objects are deallocated only when they are the results of a wrong chain of reasoning; this is very rare. Otherwise they are never deallocated since they are needed for post-run analysis. As a result, all the free pools are busy creating dynamic objects all the time. Available dynamic objects are requested as soon as created. Or even worse, managers have to wait for the free pools to create the dynamic objects. Consequently, it does not matter how long the free pools are.

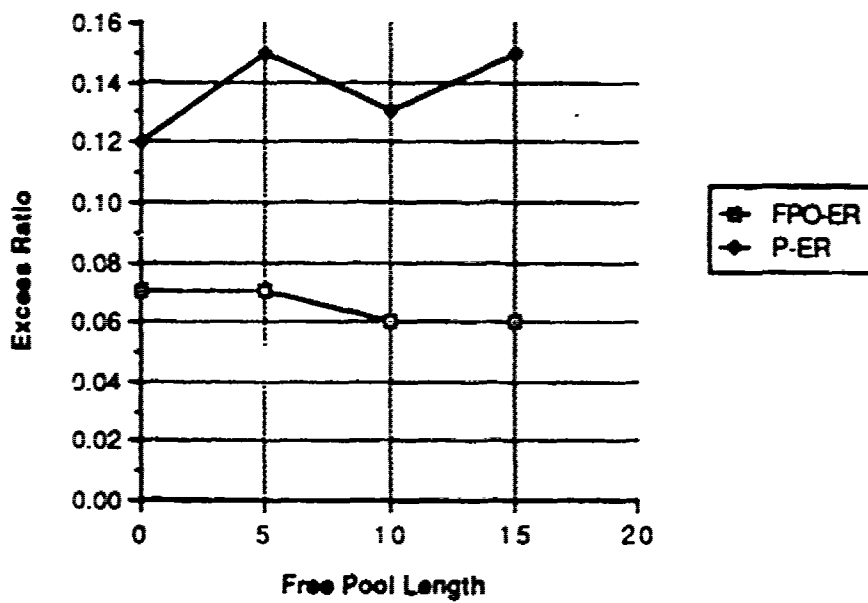
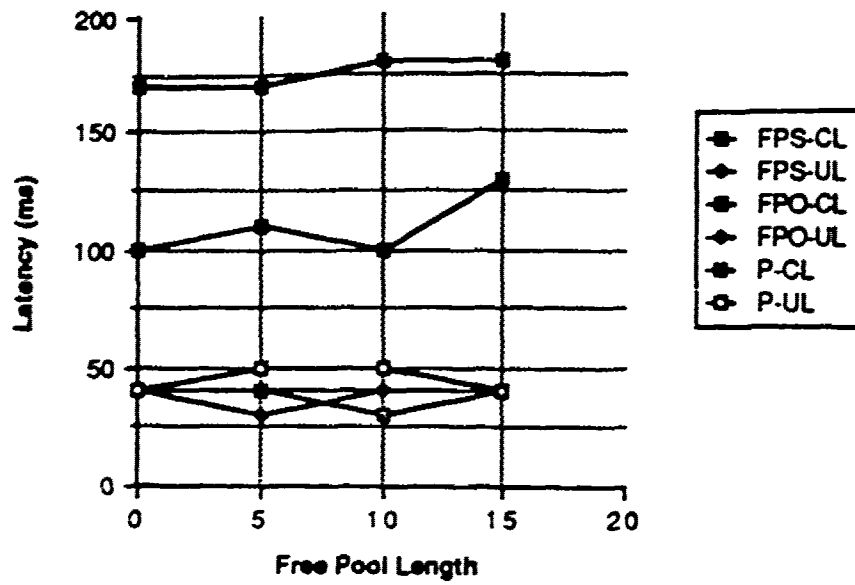


Figure 5.1 The effects of the length of free pools

Moreover, the cost of creating objects is insignificant in comparison with the object creation latencies. As shown in Figure 5.1, the object creation latencies are between 40 and

180 ms. But the object creation cost, exemplified by the cost to create P's shown in Figure 5.2, is only around 5 ms.

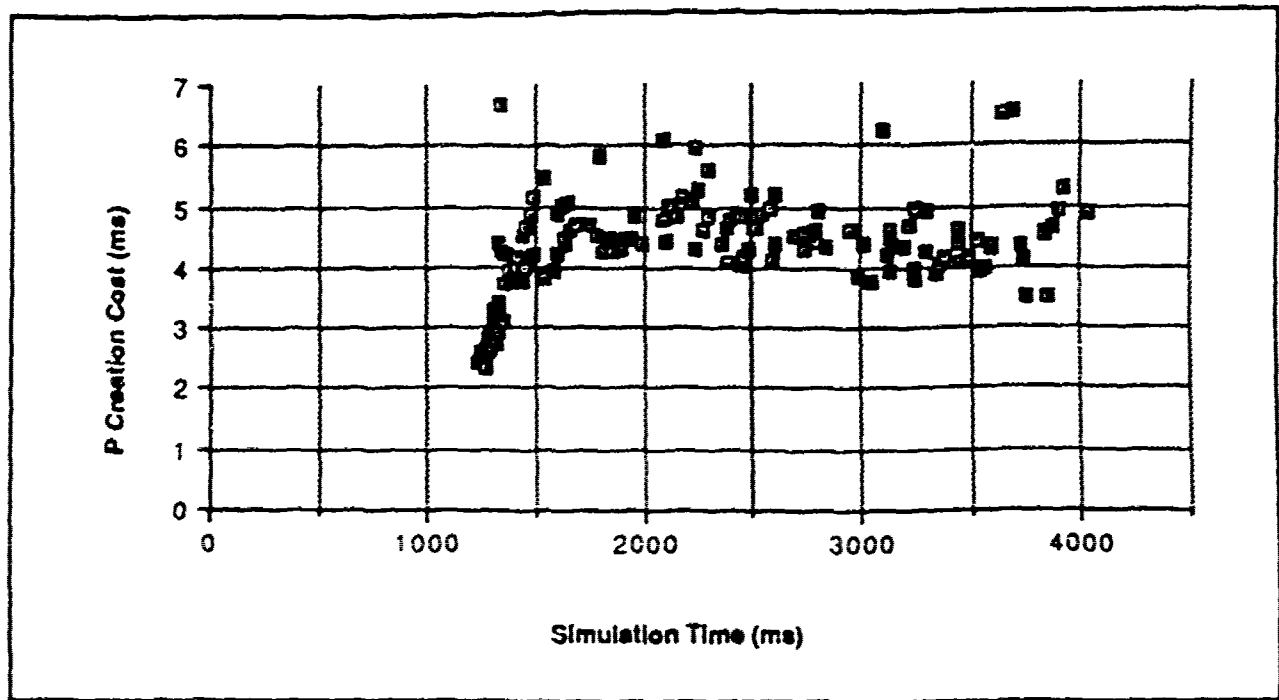


Figure 5.2 The cost to create P's

5.1.2. PM Multiplier

PM's are introduced to the manager architecture to help CPM's coordinate Fusion. The main goal is to remove the bottleneck present at the Fusion level as demonstrated in the earlier experiment [7]. Without the PM's, the CPM's was severely overloaded. Figure 5.3 shows how additional PM's help the CPM's with their tasks at the input data rate of 28.57 Hz, i.e. POR's come every 35 ms into the system. The y-axis represents the peak of the longest CPM task queue. The PM's significantly reduce the CPM task queue length. The reduction is, however, less dramatic at slower data rate.

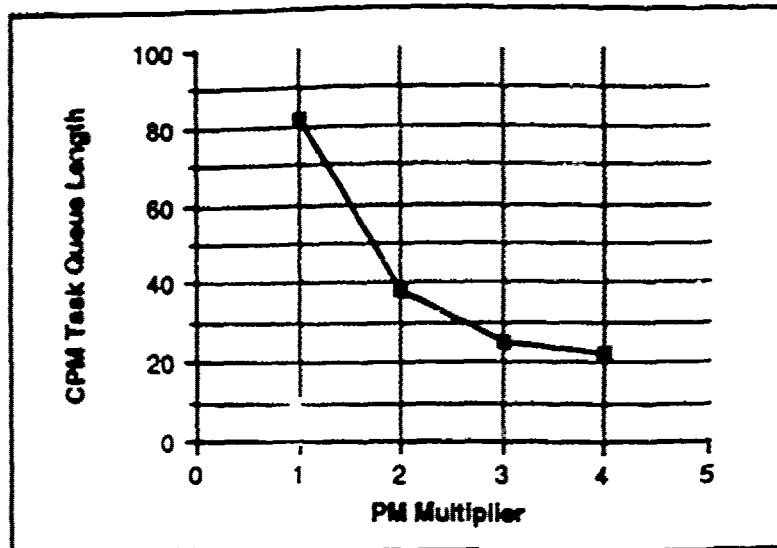


Figure 5.3 The CPM task queue length is reduced with additional PM's

The advantage of PM's is also demonstrated by their effects on *fusion cost*. Fusion cost is the time elapsed between a fusion request and its result. Figure 5.4 shows that the maximum fusion cost is dramatically reduced with additional PM's. The input data rate used is the same as above, 28.57 Hz. Fusion is quicker with more PM's because there are more managers to coordinate the process.

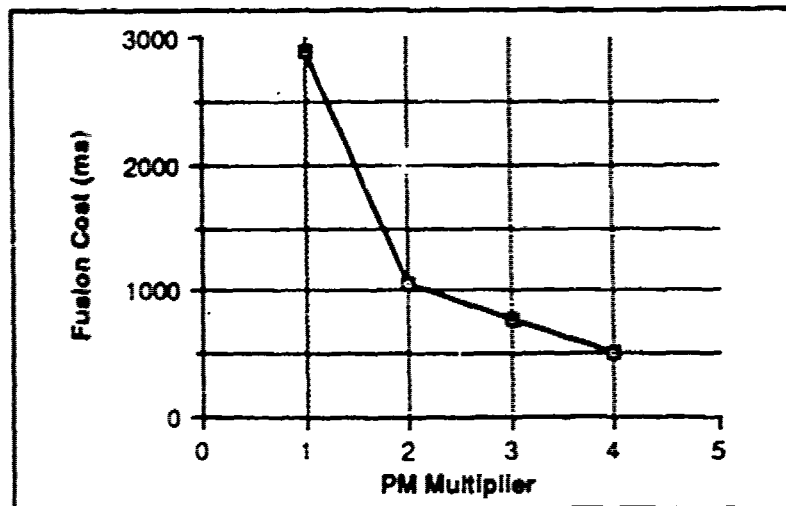


Figure 5.4 Fusion cost is reduced with additional PM's

Figures 5.5 and 5.6 show how additional PM's affect the performance of PA at the input data rate of 20 Hz and 28.57 Hz, respectively. Both figures show that the ρ Creation Latency is reduced with additional PM's. The reduction is again more significant at the higher input data rate. Since fusion cost is reduced with additional PM's, and since a new P is created only after a

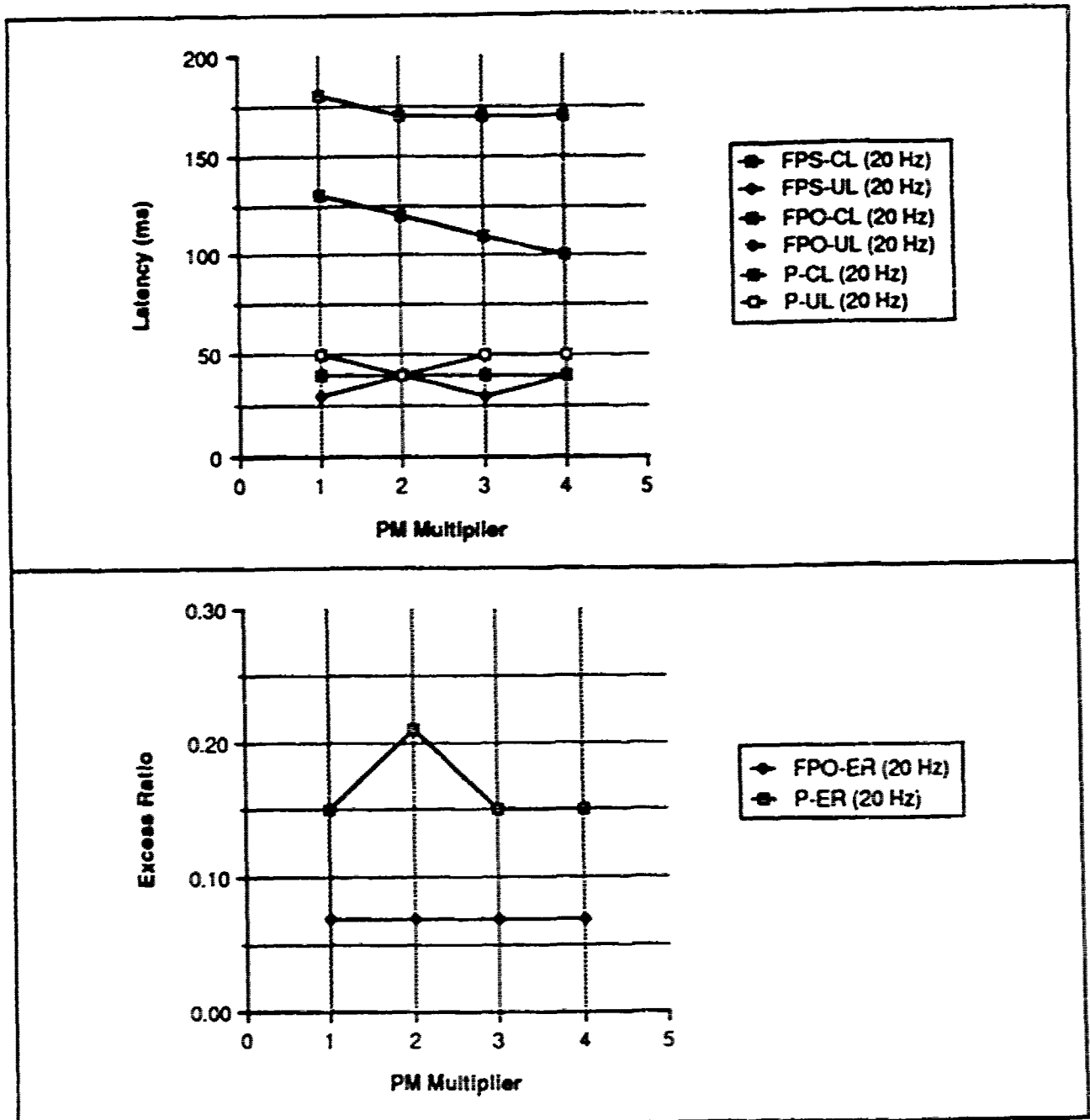


Figure 5.5 The effects of PM multiplier with a 20-Hz input data rate

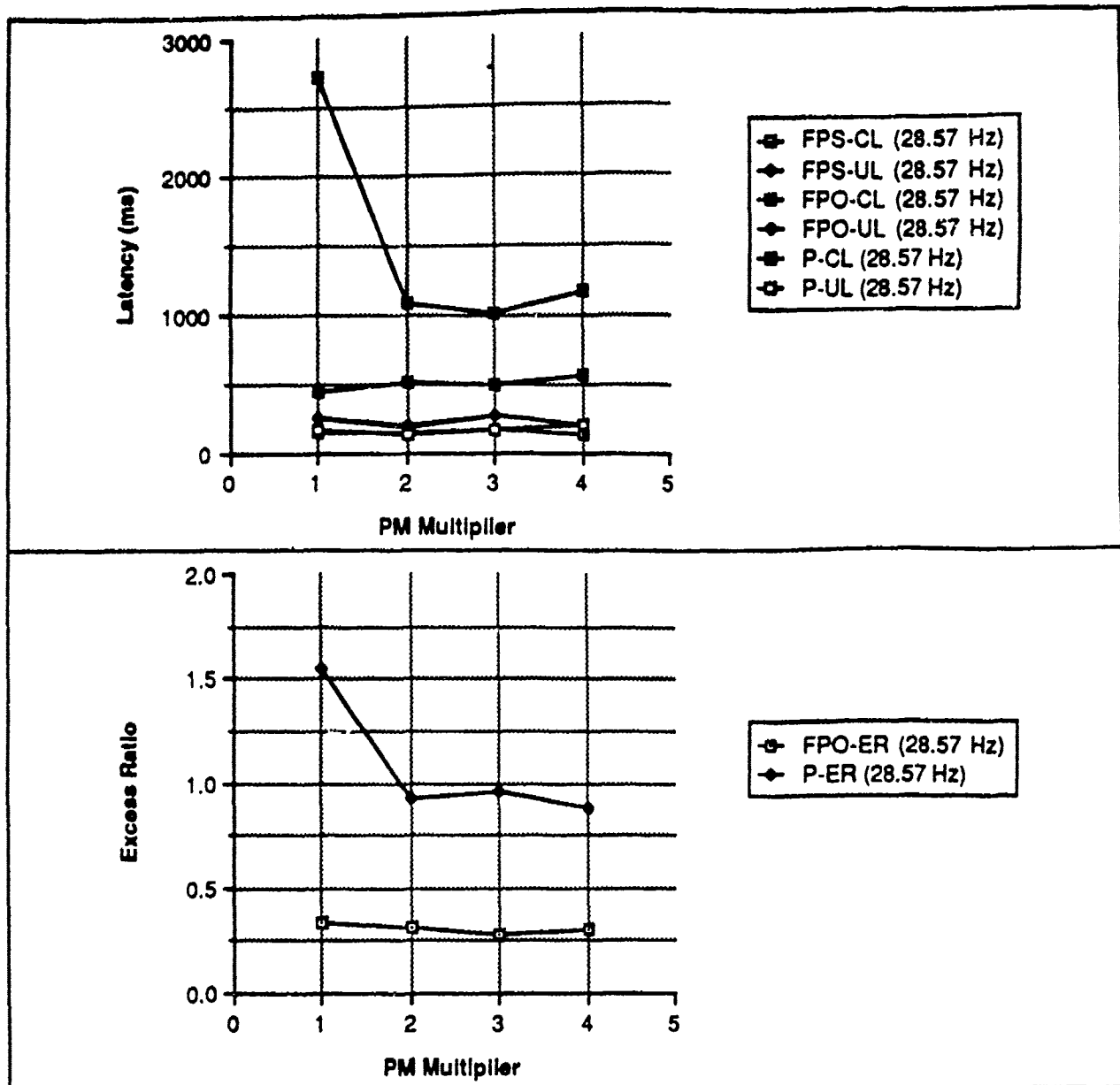


Figure 5.6 The effects of PM Multiplier with a 28.57-Hz input data rate

fusion process fails, the P Creation Latency is also reduced proportionally with additional PM's.¹

¹ The cost figures presented in this paper may be higher than their corresponding latency figures because the latter are the maxima of the last 80% of the scenario. See Section 4.2.3.1. The cost figures, on the other hand, are the absolute maxima.

The P Excess Ratio is also reduced with more PM's as shown in Figure 5.6. It is interesting to note that the P Excess Ratio, a qualitative measurement, and the P Creation Latency, a quantitative measurement, are *proportionally*, not *inversely*, related. When fusion cost is high, an FPO frequently comes back from the last fusion retry empty handed because its matching FPO is still involved in its own last fusion search.¹ As a result, different P's are created for both FPO's, although they can actually fuse. Consequently, the P Excess Ratio becomes high. However, when fusion cost is low, an FPO can quickly decide if it needs a new P. So when a matching FPO reports for fusion, the P is very likely ready to be fused with the FPO. Hence, the P Excess Ratio would be relatively lower.

Since managers in this experiment have dedicated sites, the more PM's in the system, the less sites available for dynamic objects. More PM's in the system reduce the fusion cost and improve the performance. But at the same time, more PM's mean less computational resources for the dynamic objects such as P's. So there is a trade-off. When the grid size is relatively large, such as 128 in this case, the disadvantages of additional PM's are relatively insignificant. However, further experiment needs to be done on smaller grid sizes.

5.1.3. FPS History Ring Buffer Length

The history ring buffer is the mechanism used to store history in the Connection stage. The ring buffer keeps the FPS's to be checked for connection. The length of the ring buffer limits the amount of history saved. If the buffer is short, few FPS's need to be checked for connection, and *connection cost*, the time elapsed between a connection request and its result, is low. This is shown in Figure 5.7, whose y-axis shows the maximum connection cost. The connection cost is higher when the ring buffer is longer, because there are more FPS's to be checked for connection.²

¹ This is possible since there are usually more than one fusion search at a time in one set of fusable P's and FPO's, i.e. P's and FPO's of the same aircraft type. In contrast, there is only one connection search at time in one set of connectable FPO's and FPS's, i.e. FPO's and FPS's of a particular aircraft type observed from a particular radar.

² Since the connection search among one set of connectable FPO's and FPS's is performed *serially*, the connection cost is directly linked to the number of FPS's to be checked for connection. In contrast, the fusion search among one set of fusable P's and FPO's is performed *concurrently*. So the fusion cost is not directly affected by the number of P's to be checked for fusion. However, the greater the number of

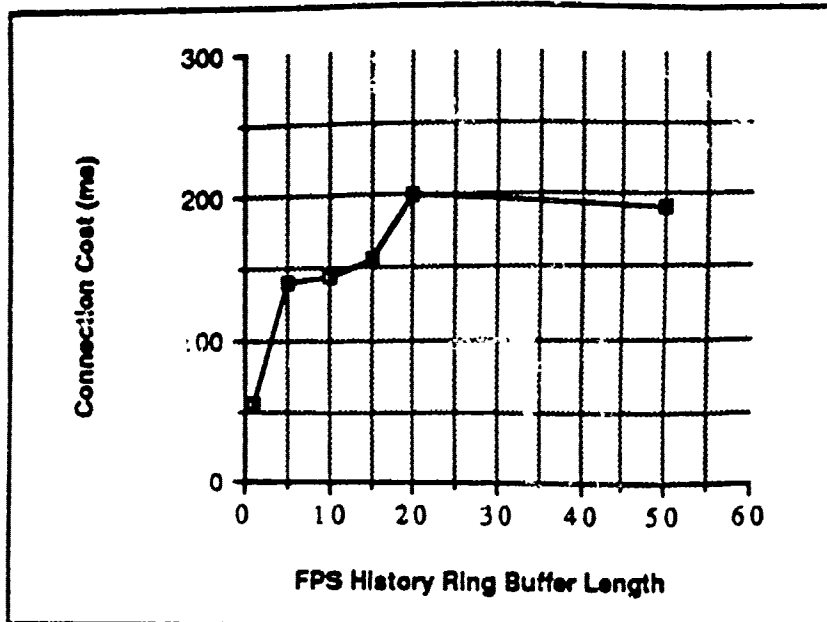


Figure 5.7 The connection cost goes up with longer history

The effect of longer history ring buffer is also reflected on the FPO Creation Latency as shown in Figure 5.8. Since connection cost increases with longer ring buffer, and since a new FPO is created only after a failed connection process, the FPO Creation Latency also increases with longer ring buffer. Furthermore, since Fusion happens after Connection, and since the POR that triggers a P creation in Fusion is generally¹ the same POR that triggers an FPO creation in Connection,² higher FPO Creation Latency means higher P Creation Latency.

The quality of Connection, represented by the FPO Excess Ratio shown in Figure 5.8, improves with longer history in the ring buffer. So the more knowledge stored in the ring buffer, the less connections missed by the system. But the price to pay is the higher creation latencies described above. It is the typical case: the quantitative and qualitative aspects of the performance are *inversely* related.

the P's, the more likely that some of them reside on the same sites, and hence the fusion process takes longer. See Appendix A3 for the kinds of objects on each site.

¹ This would always be true if messages are always in order in CARE.

² Not all POR's that trigger FPO creations will trigger P creations, since a P usually consists of several FPO's.

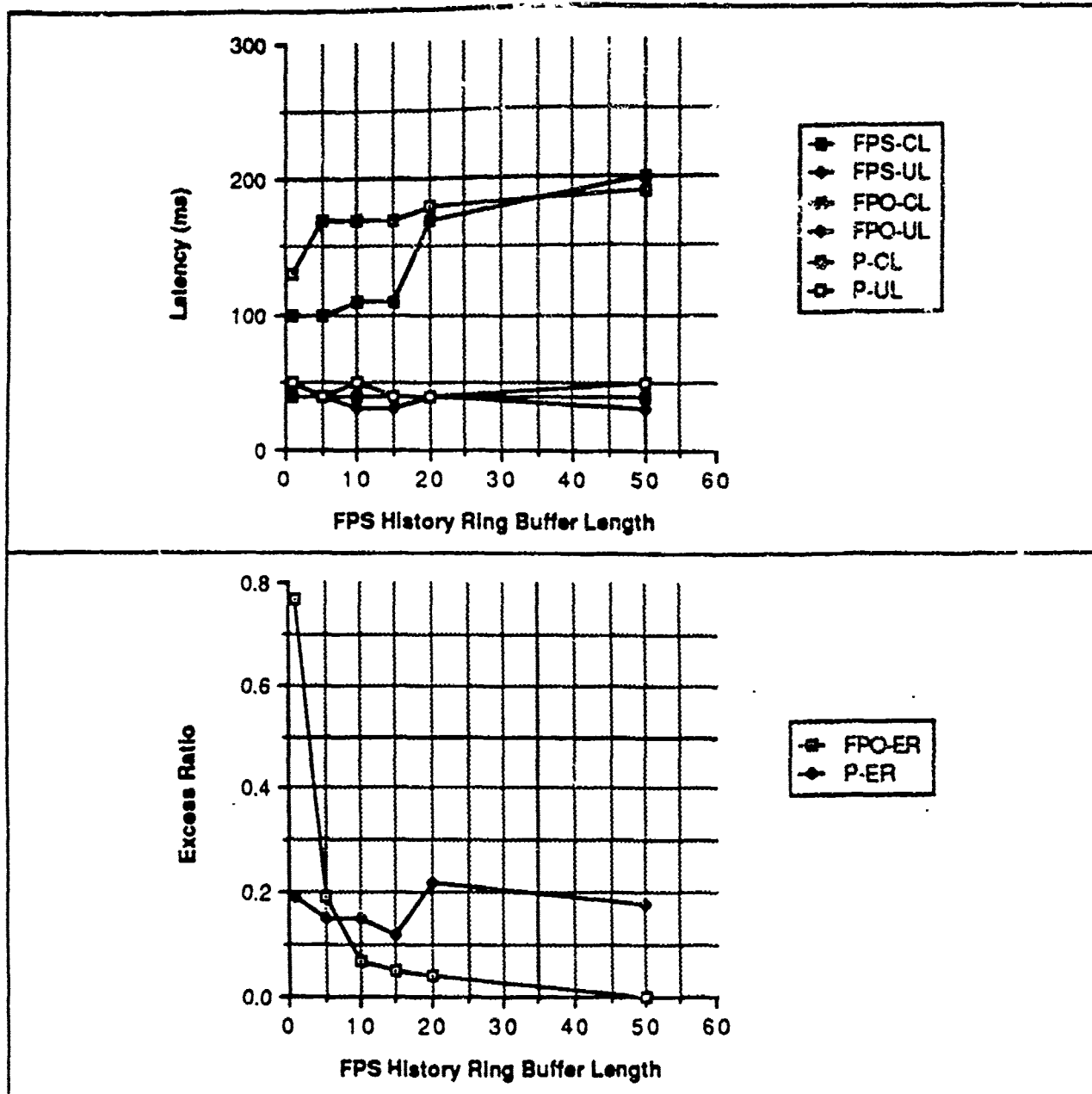


Figure 5.8 The effects of the length of FPS history ring buffer

5.1.4. Connection Search Interval

The connection search interval is the maximum time gap allowed between two connected FPS's. It is a heuristic used to determine temporarily if two tracks, represented by the two FPS's, are part of the same flight path. If no time gap is allowed, no connection needs to be done, and the connection cost is 0, as shown in Figure 5.9. The longer the time gap allowed, the more

connections are possible. The more connections needs to be performed (serially), the higher the connection cost.

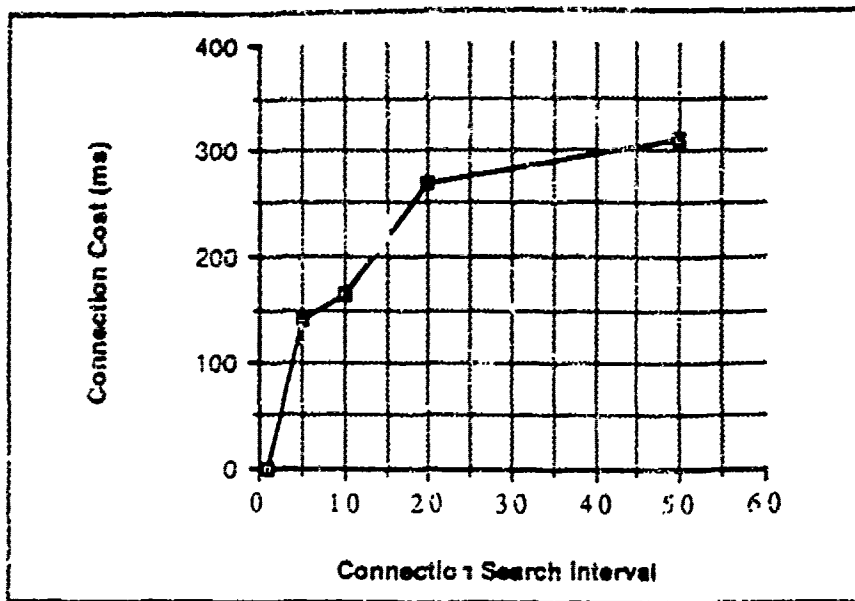


Figure 5.9 The wider the time gap allowed, the higher the connection cost

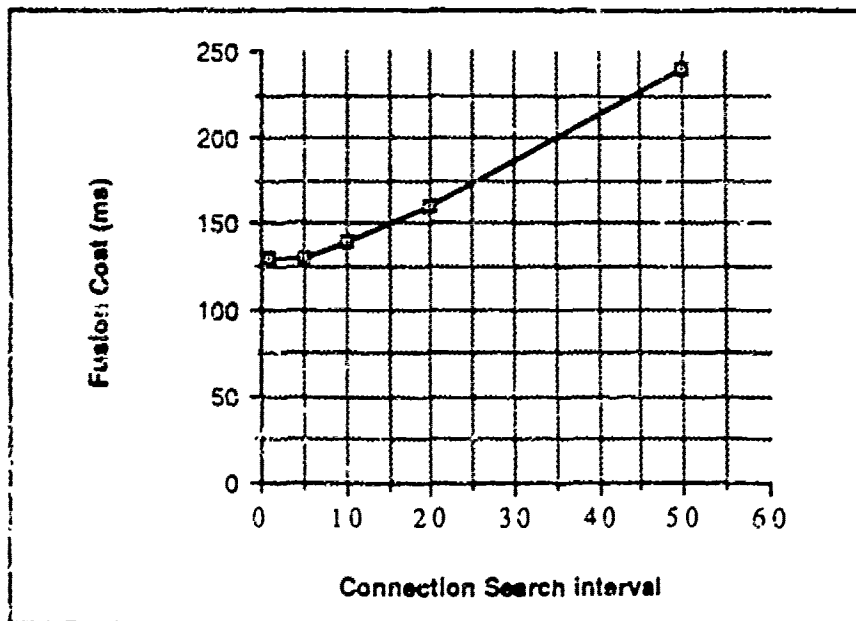


Figure 5.10 Fusion cost is also affected by the connection search interval

The increased possible connections per track introduce unnecessary ambiguities into the system. Fusion has to attempt to resolve these ambiguities. Consequently, as shown in Figure 5.10, the fusion process becomes more expensive as the time gap allowed becomes wider.

The effects of the connection search interval on the latencies and excess ratios are shown in Figure 5.11. The higher connection cost caused by larger connection search interval is directly reflected as higher FPO Creation Latency. This is because an FPO is created only after a failed connection process. Furthermore, since Fusion happens after Connection, and since the POR that triggers a P creation in Fusion is usually the same POR that triggers an FPO creation in Connection, higher FPO Creation Latency means higher P Creation Latency. The higher fusion cost caused by wider connection search interval contributes to even higher P Creation Latency.

If no time gap is allowed, no connection is performed. As a result, each FPS is represented by a different FPO. Hence, the FPO Excess Ratio becomes very high as depicted in Figure 5.11. The FPO Excess Ratio drops dramatically when connections are performed. However, as the connection search interval becomes wider, more connections per FPS are possible. Recall that when an FPS has multiple possible connections, Connection creates a new FPO for the FPS and links them together as connected FPO's. Fusion may fail to resolve ambiguities such as this one, and the excess FPO's remain in the system. Consequently, as the connection search interval becomes wider than necessary, the FPO Excess Ratio becomes higher.

The effects on the P Excess Ratio is similar. If no connection is performed, Fusion may fail to fuse some of the excess FPO's and creates different P's for them instead. As a result, the P Excess Ratio is rather high when no connection is performed, as shown in Figure 5.11. The quality of Fusion improves when connections are performed. However, as more excess FPO's are created due to wider connection search interval, Fusion may again fail to fuse some of them properly. Consequently, the P Excess Ratio rises again.

The FPO Creation Latency and FPO Excess Ratio are inversely related if the connection search interval is low. However, as the interval increases, the relationship becomes proportional. The P Creation Latency and P Excess Ratio show proportional relationship throughout.

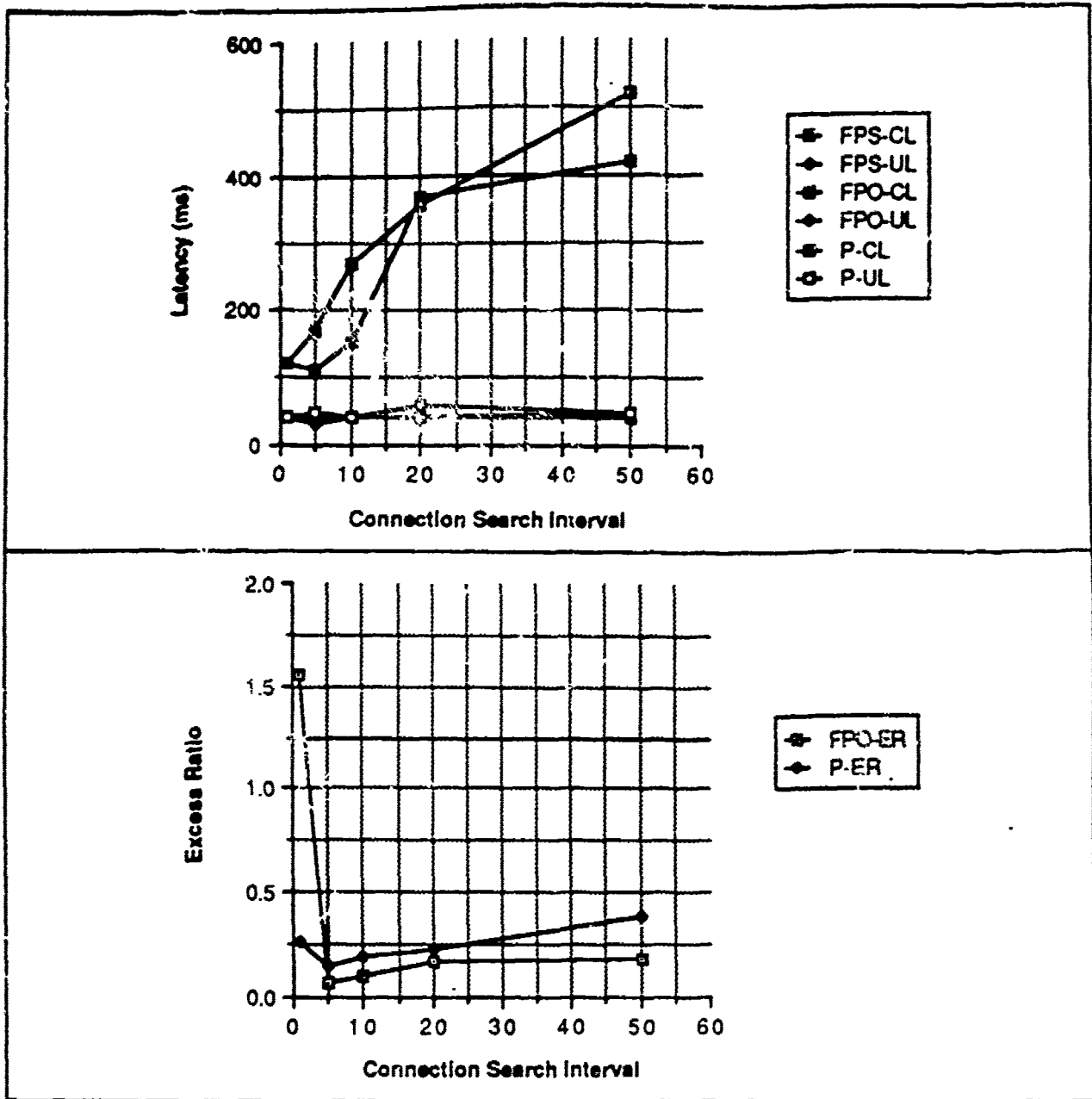


Figure 5.11 The effects of connection search interval

5.1.5. Maximum FPO Fusion Retry

The notion of FPO fusion retry was originally implemented just as a safeguard against the possibility that a fail-to-fuse FPO may fuse with new P's created during the FPO's fusion search. The possibility was first thought to be quite small. However, this is not true. It turns

out that many FPO's that fail to fuse in the first fusion try eventually fuse in subsequent fusion retries. Many of the P's created *during* the FPO's previous fusion searches can actually fuse with the FPO's in the following fusion try. Since a new P is created every time an FPO fail to fuse, the P Excess Ratio is high when no retry is performed. As shown in Figure 5.12, the ratio is reduced significantly when a number of fusion retries are performed.

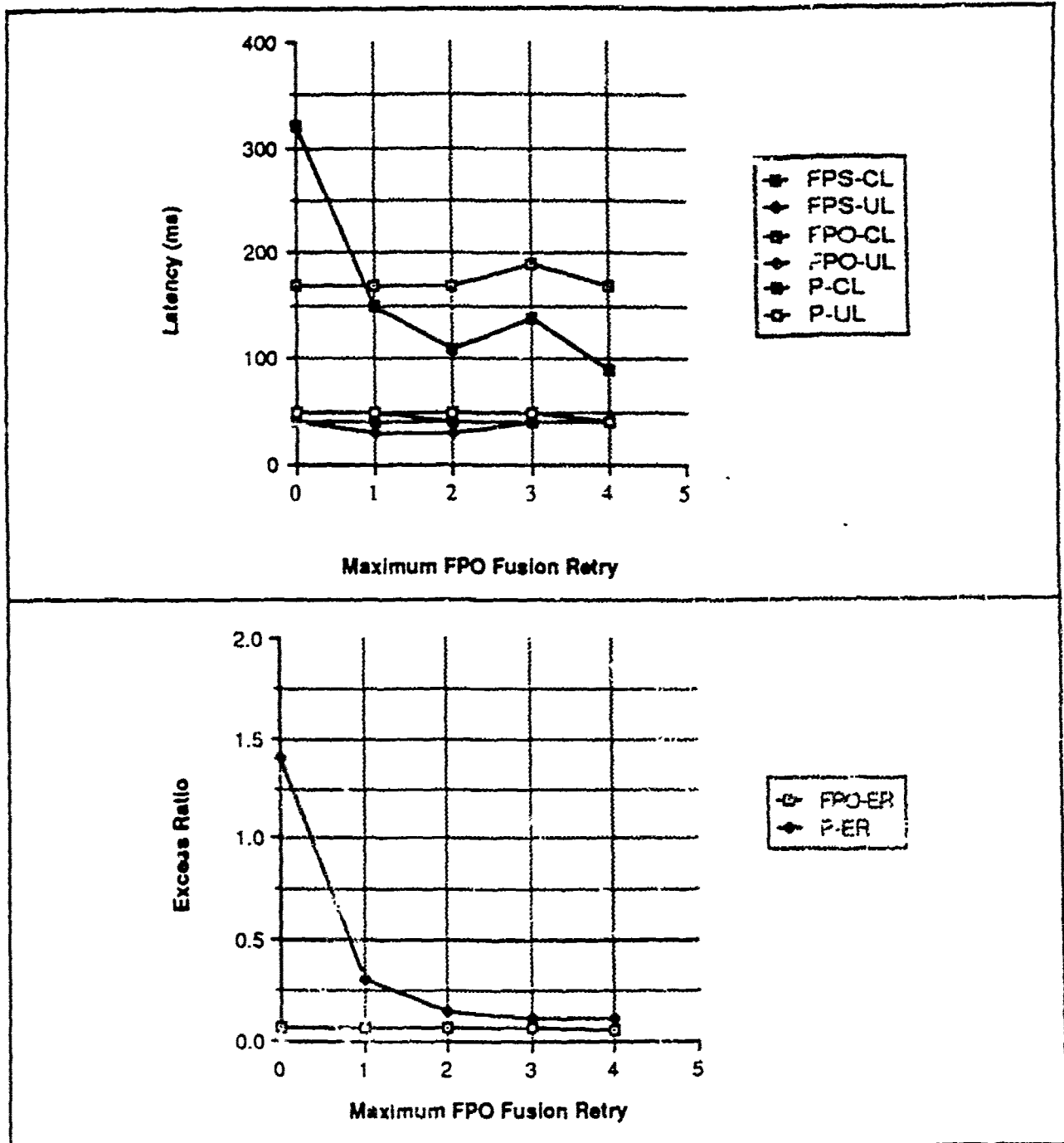


Figure 5.12 The effects of the maximum of FPO fusion retries

Figure 5.12 shows that the P Excess Ratio is once again proportionally related to the P Creation Latency. If no fusion retry is performed, a lot of excess P's are created. As a result, there are also many P's to be checked for fusion. The probability that the P's reside on the same sites becomes high.¹ P's on the same site can only be checked for fusion serially. Consequently, the fusion cost becomes high, as shown in Figure 5.13. This also results in high P Creation Latency. When fusion retries are performed, however, few excess P's are created, therefore the number of P's to be checked is lower, and so are the fusion cost and P Creation Latency.

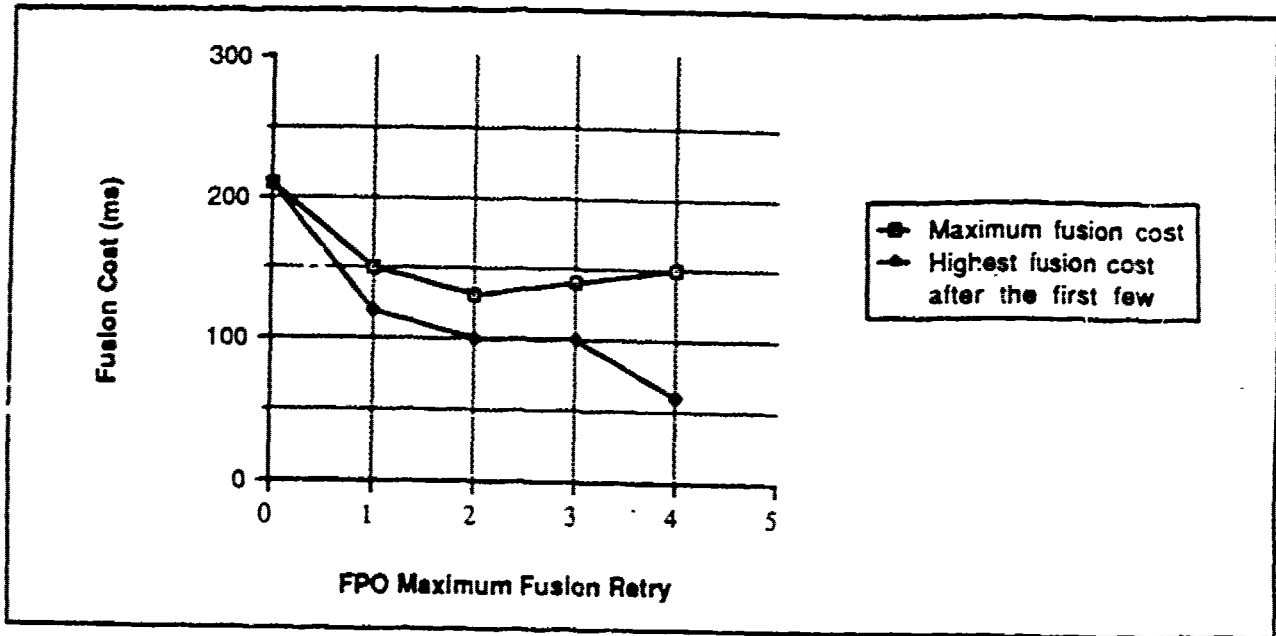


Figure 5.13 The effects of fusion retries on the fusion cost

The maximum fusion cost becomes relatively high when too many fusion retries are performed. But the maximum cost only represents the cost of the first few fusion searches. Since the first few FPO's reporting for fusion cannot be fused with any P's (since there is none), the FPO's are forced to keep trying to fuse before P's are finally created when the retry limit is reached. Consequently, the fusion cost is high. But afterwards, as more and more fusable P's are actually present in the system, the high retry limit would rarely be reached, and the fusion cost decreases. As shown in Figure 5.13, in the case of maximum fusion retry of 4, the

¹ See Appendix A3 for a sample of the kinds of objects on each site.

highest fusion cost after the first few data (in simulation time) is significantly lower than the overall maximum fusion cost. The higher the retry limit, the wider the gap between the two curves.

5.1.6. P Waiting Period

The P waiting period is used to limit the number of P's to be checked for fusion. P's that have been inactive for the duration of the waiting period are retired from the system, and no longer involved in any fusion process. The retired P's represent aircraft that have left the observed region of airspace. So the P waiting period is a mechanism used to discard old and obsolete data.

The effects of the P waiting period on the fusion cost is shown in Figure 5.14. At a waiting period of 6,000 ms, the P's in this simulation are essentially never retired. So every fusion process involves *all* P's the system has seen. Every P, obsolete or recent, has to be checked for fusion. The larger the number of P's to be checked, as explained earlier, the higher the fusion cost. As the waiting period becomes shorter, obsolete P's are retired more quickly, and therefore less P's are involved in any fusion process. Hence, the fusion cost becomes lower.

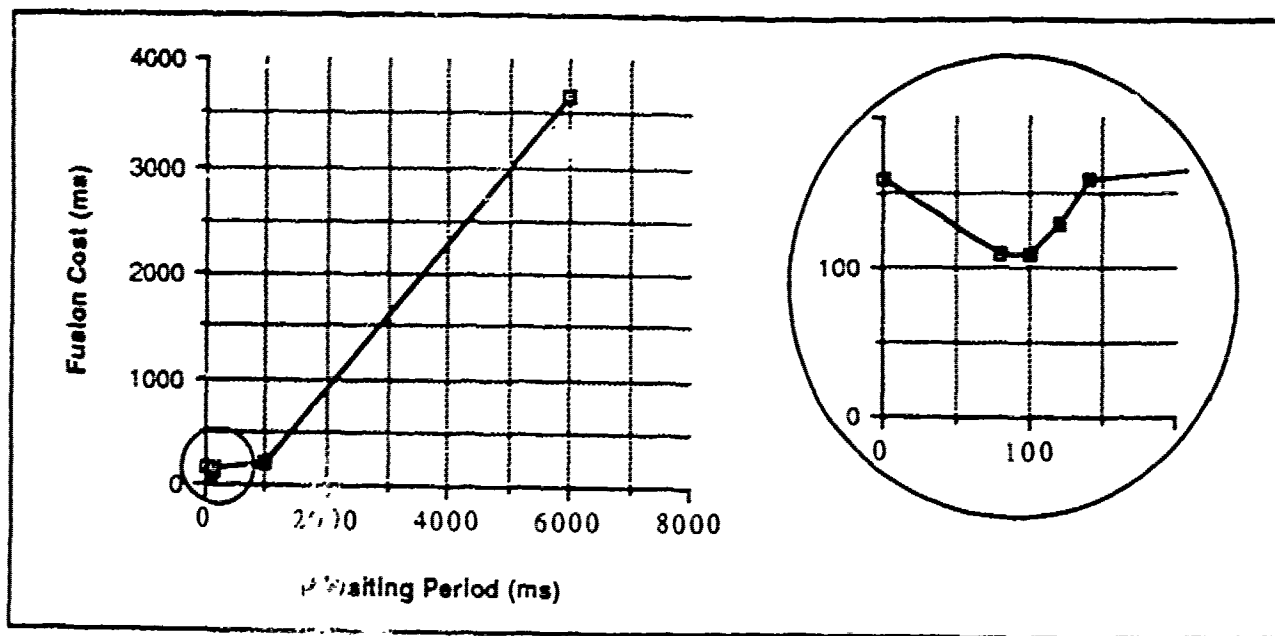


Figure 5.14 The waiting period greatly affects the fusion cost

Since a new P is created only after a failed fusion process, the effects of the P waiting period on the P Creation Latency is essentially the same as its effect on the fusion cost. The P Creation Latency rises as the P waiting period goes up, as shown in Figure 5.15.

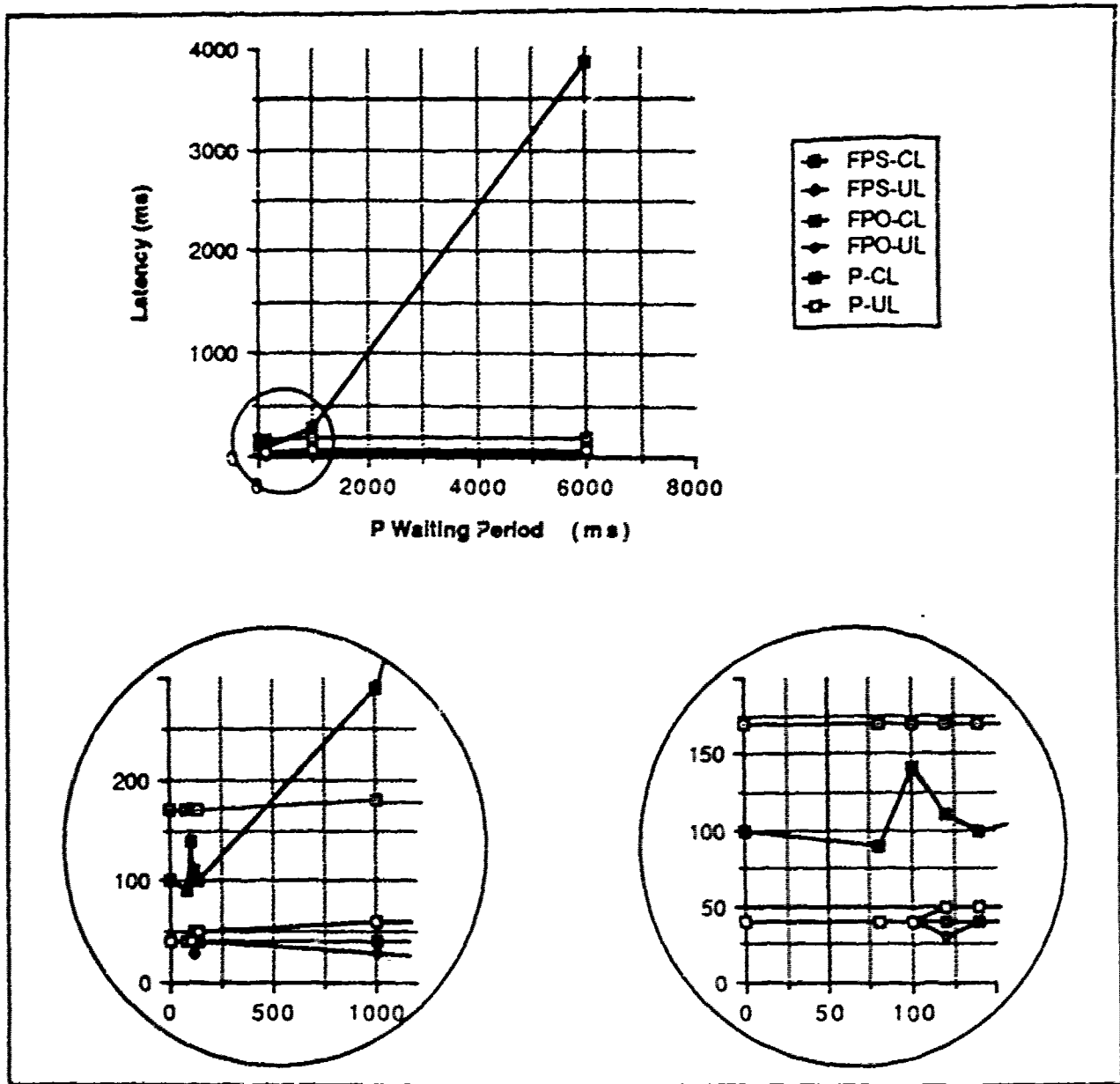


Figure 5.15 The quantitative effects of the P waiting period

The P Excess Ratio is also affected the same way, as shown in Figure 5.16. So the P Excess Ratio and P Creation Latency is once again proportionally related. As explained earlier,

when the cost is high, fusion frequently fails. Many excess P's are created as a result. But when the cost is low, fusion is likely to succeed, and few excess P's are created.

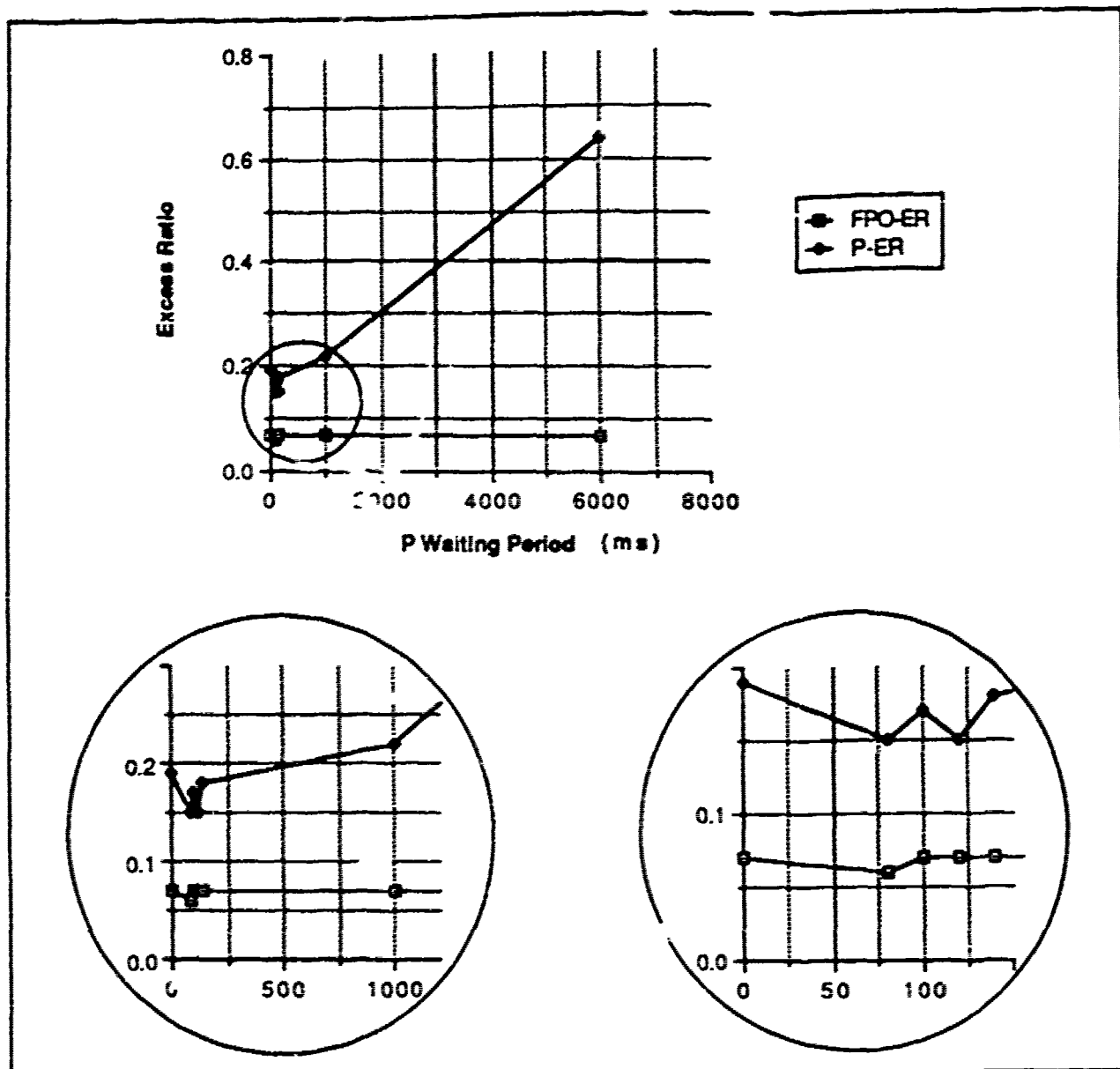


Figure 5.16 The qualitative effects of the P waiting period

Note that if inactive P's are retired instantaneously after they become inactive, i.e. the P waiting period is set to 0, the performance is not as good as if a short waiting period is applied. Since inactive P's can be active again when track information updates are received, it is important to make sure that inactive P's are indeed inactive. Otherwise, new P's have to be

created for FPO's that should fuse with those P's, because the supposedly inactive P's, now retired, are no longer checked for fusion. The P Excess Ratio becomes relatively high, and so do the fusion cost and P Creation Latency. A short delay in retiring inactive P's is necessary to make sure that no more track updates are to be received. Recall that the updates do have some latencies associated with them. Indeed the update latencies are part of the overall performance evaluation.

5.1.7. Optimal Parameter Values

The results above suggest that some parameter values yield better performance than the standard. Listed below are the parameter values of the new standard. New values are in *italic bold*.

- Free pool lengths: *0, 0; 0, 0; 0, 0; 0, 0*
- PM multiplier: *3*
- FPS history ring buffer length: *10*
- Connection search interval: *5*
- Maximum FPO fusion retry: *3*
- P waiting period: *120 ms*

The new values are chosen primarily because they yield better P-related performance measurements, i.e. lower P latencies and excess ratio. The P's, after all, are the final output of PA. The performance figures of the old and new standards are compared in Figure 5.17. The P Excess Ratio is reduced dramatically in the new standard. Other differences are small and/or insignificant.

5.2. The Effects of the Frequency of Input Data

The effects of the input data rate are investigated on grid sizes 36, 64, 128, and 256 using the C-130 scenario. The results are exemplified by the effects on the P Creation Latency and P Excess Ratio. The complete results can be found in Appendix A4. Other results are similar

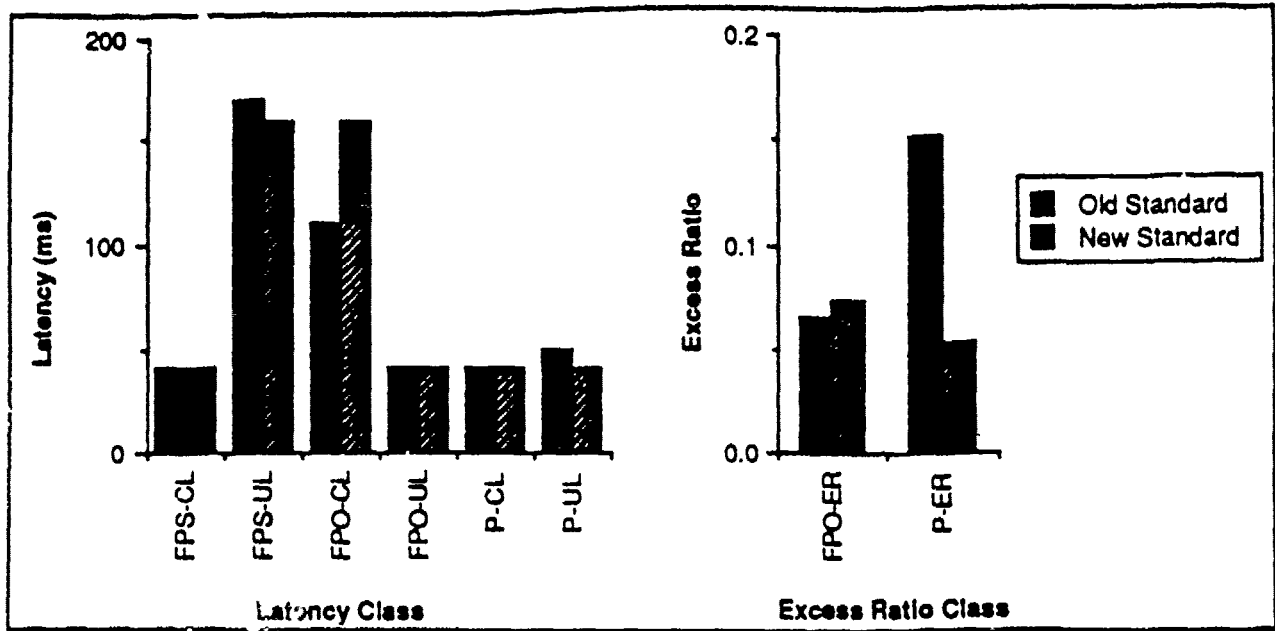


Figure 5.17 The old standard vs. the new standard

to the ones presented here, except for the FPS Excess Ratio, which is always 0, i.e. PA never misses any distribution tasks.

The effects of the frequency of input data across grid sizes on the P Creation Latency are shown in Figure 5.18, with a close-up on the lower range in Figure 5.19. The curves represent the highest latency reported. On any grid size, the latency goes up as the input data rate rises. The quantitative performance deteriorates very rapidly when the system is stressed at high input data rate.

Across grid sizes, the latency increases much more quickly on grid size 36 than on any other grid sizes. The smaller the grid size, the quicker the performance degrades as the input data rate rises. For any given level of latency, i.e. quantitative performance, the maximum data frequency achievable is higher for a higher grid size. Similarly, for any given input data rate, the latency decreases with grid size, i.e. the quantitative performance improves with more processing units.

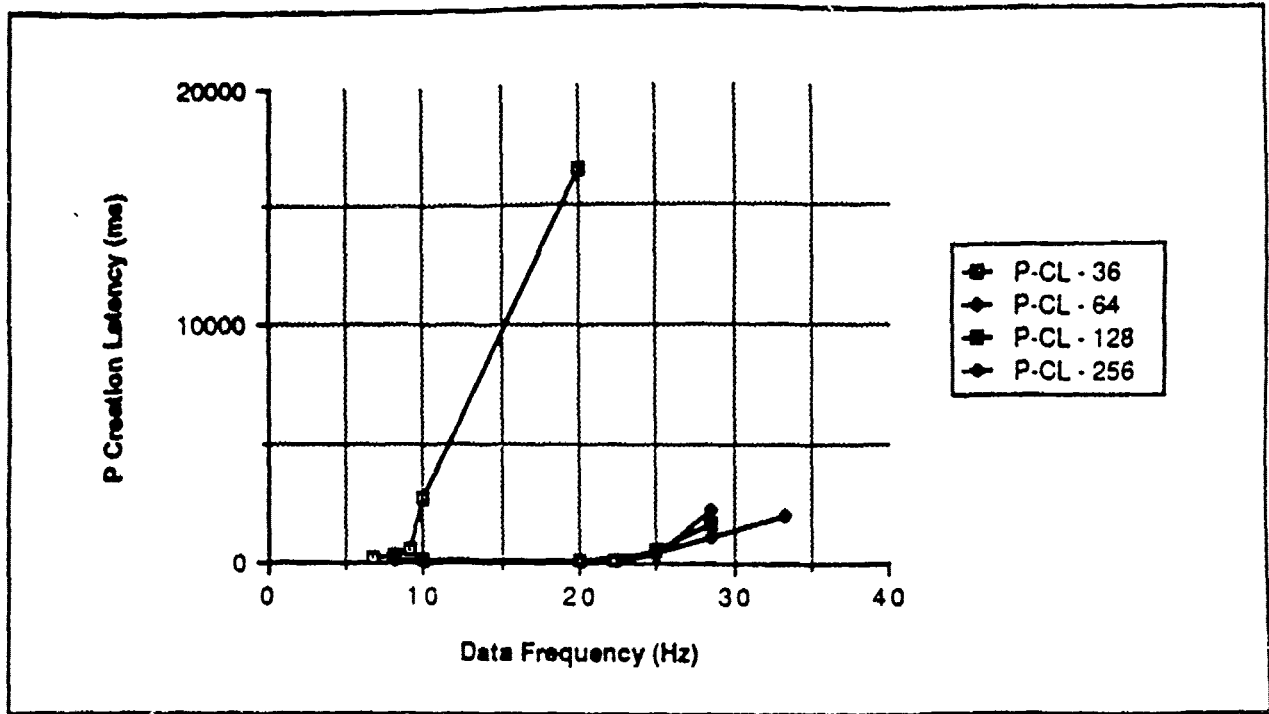


Figure 5.18 The effects of the input data rate on the P Creation Latency

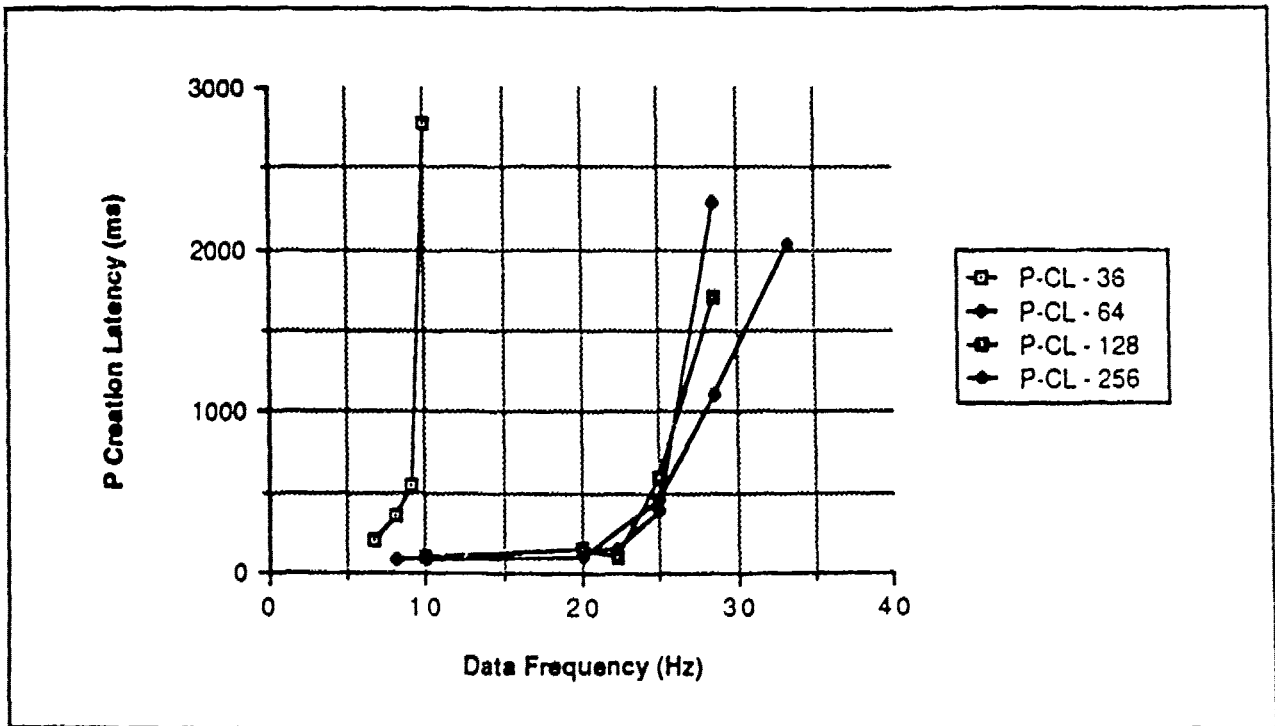


Figure 5.19 A close-up on the effects of the input data rate on the P Creation Latency

However, the knees of latency curves for grid sizes 64, 128, and 256 are about the same. Although the grid size 256's latency at 28.57 Hz is substantially lower than the grid size 64's at the same data frequency, the former latency is still relatively high. For a low range of latency, i.e. a desired level of performance, any significant reduction in the latency stops at grid size 64.

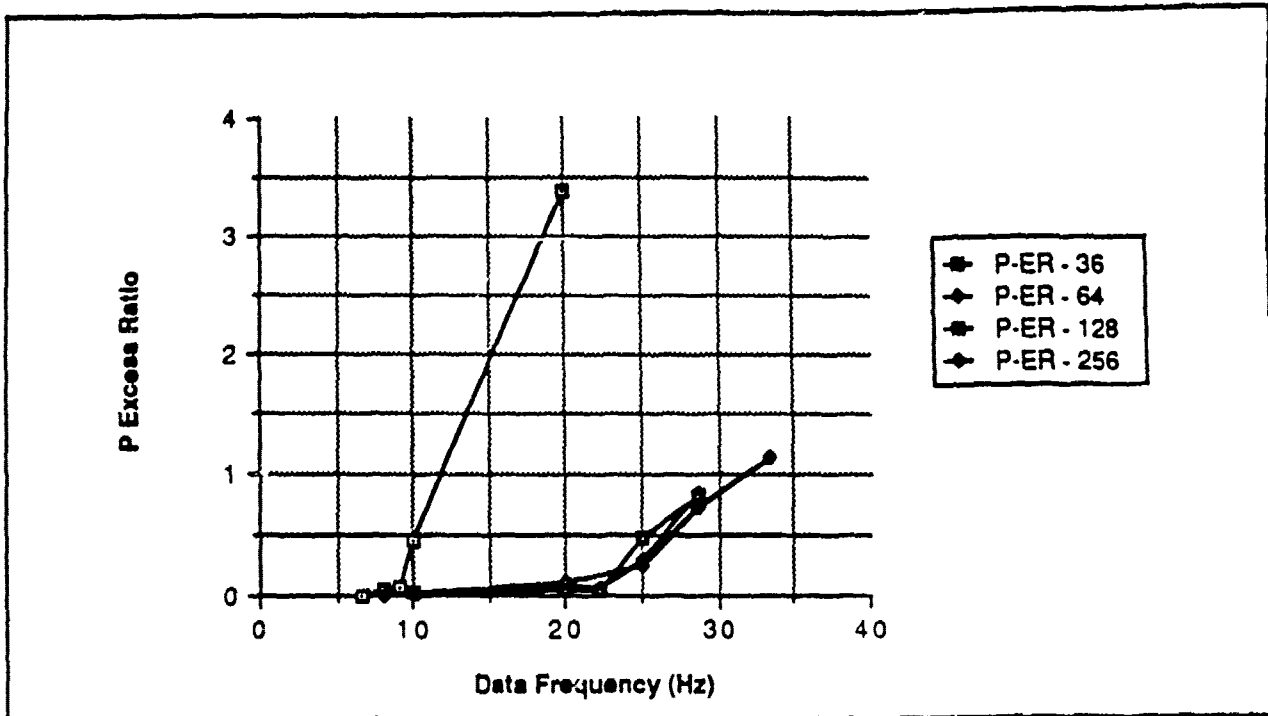


Figure 5.20 The effects of the input data rate on the P Excess Ratio

The effects of the input data rate on the P Excess Ratio, shown in Figure 5.20, are the same. The qualitative performance also degrades quickly when the system is stressed at high data rate. There is a conclusive and substantial performance improvement from grid size 36 to 64 at any given data frequency. The additional sites at grid sizes 128 and 256 do not seem to add much to the performance.

5.3. The Effects of the Width of Input Data

The effects of the width of input data are also studied on grid sizes 36, 64, 128, and 256. The scenarios described in Section 4.3 are simulated at 20 Hz, i.e. POR's come every 50

ms into the system. The results are once again exemplified by the effects on the P Creation Latency and P Excess Ratio. The complete results can be found in Appendix A5. Other results are similar to the ones presented here, except for the FPS Excess Ratio, which is always 0, i.e. PA never misses any distribution tasks.

The effects of the width of input data across grid sizes on the P Creation Latency are shown in Figure 5.21, with a close-up on the lower range in Figure 5.22. The curves represent the highest latency reported. The effects are similar to the ones described in the previous section. On any grid size, the quantitative performance deteriorates rapidly at high input data rate. Across grid sizes, the latency once again increases much more rapidly on grid size 36 than on the larger grid sizes. The knees of latency curves for grid sizes 64, 128, and 256 are about the same, too. For a desired level of performance, any substantial reduction in the latency stops at grid size 64.

The effects of the width of input data across grid sizes on the P Excess Ratio, shown in Figure 5.23, are the same. The qualitative performance degrades rapidly at high data rate. The performance improves significantly from grid size 36 to 64 at any given data width, but the additional sites at grid sizes 128 and 256 do not seem to add much to the performance.

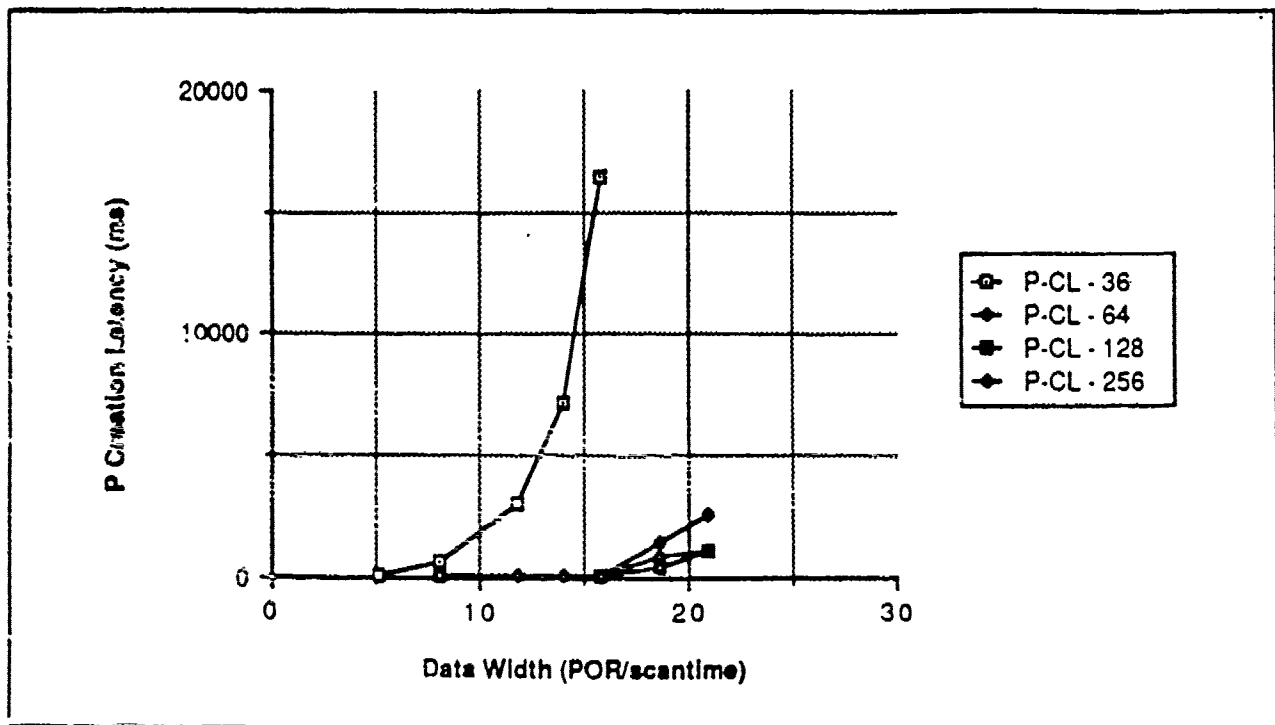


Figure 5.21 The effects of the width of input data on the P Creation Latency

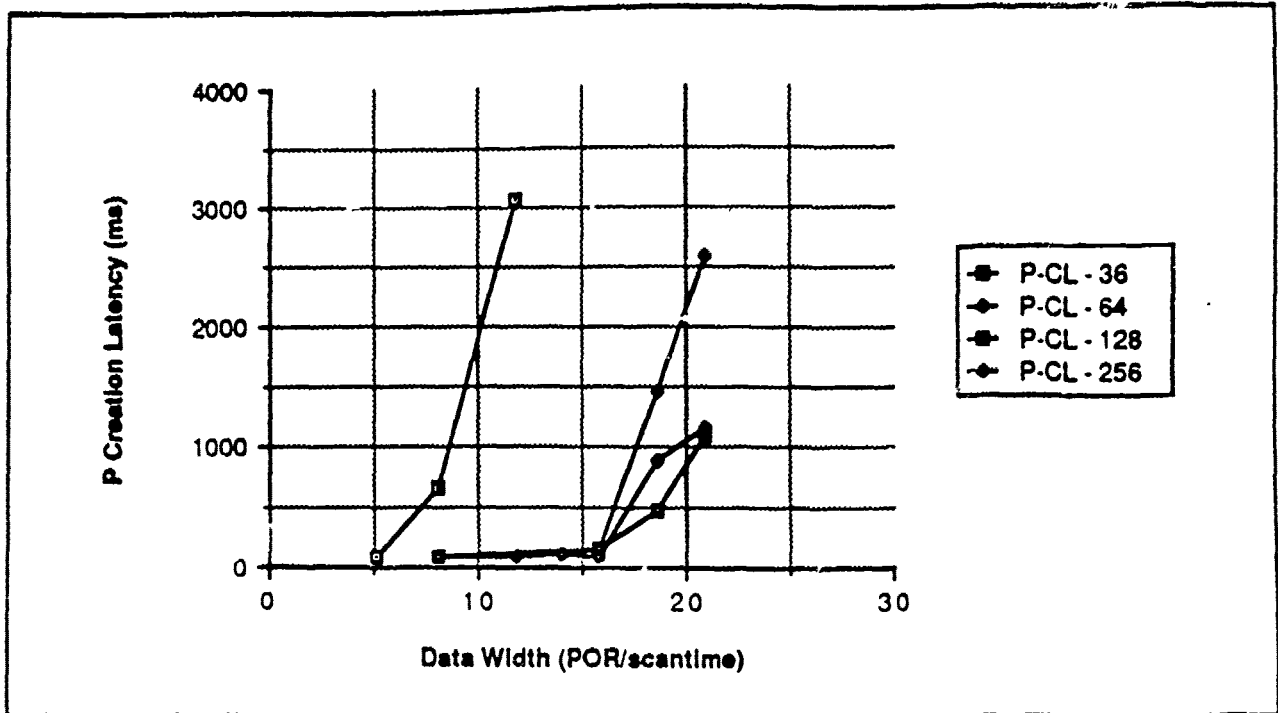


Figure 5.22 A close-up on the effects of the width of input data on the P Creation Latency

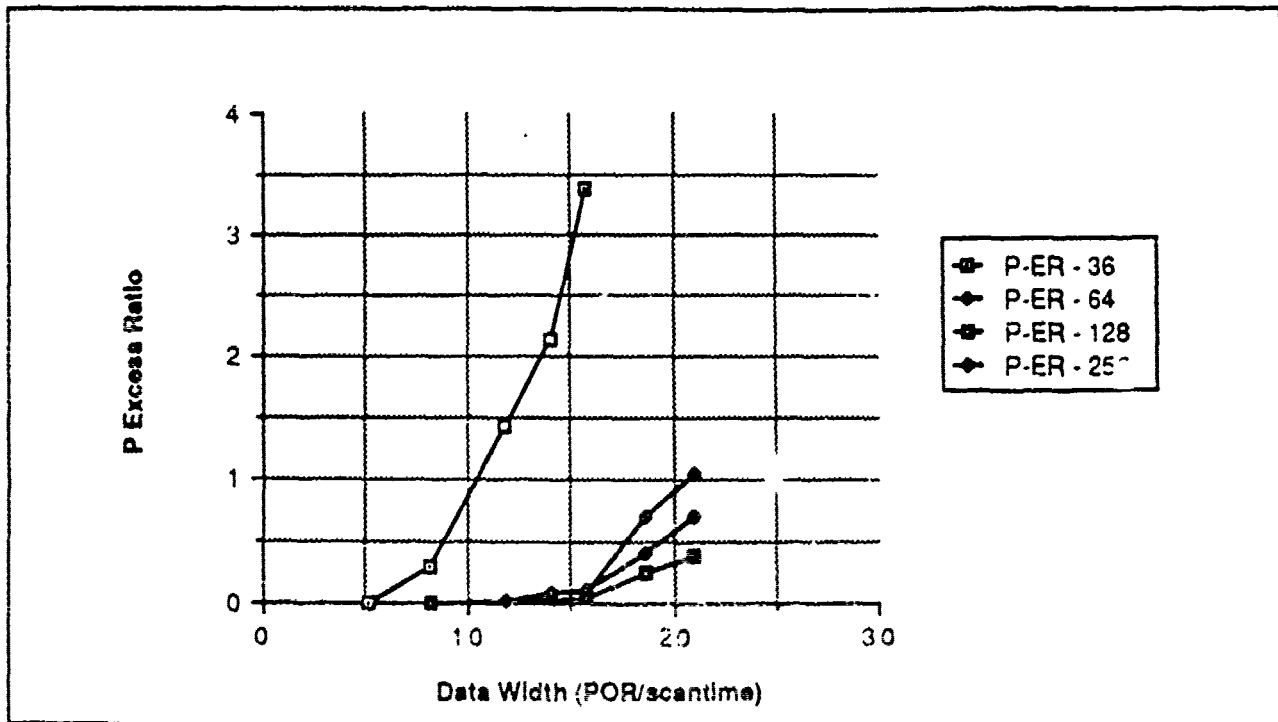


Figure 5.23 The effects of the width of input data on the P Excess Ratio

5.4. Possible Speedup Curves

A speedup curve is generated by plotting sustainable data rate or capacity versus grid size. A performance requirement must first be specified to find the sustainable data rate and capacity of each grid size. A sustainable data rate or capacity that satisfies a given pair of qn and ql , the performance requirement, is the lowest sustainable data rate or capacity that satisfies each element of qn and ql .

Figures 5.24 through 5.29 show the sustainable data rate achievable, on the grid sizes 36, 64, 128, and 256, for a selected range of latency requirement. Each figure corresponds to an element of qn , except for FPS Excess Ratio which is always 0. Figures 5.30 and 5.31 show the data frequencies on the four grid sizes that correspond to different FPO and P excess ratios. These data frequencies are not necessarily sustainable, since the notion of sustainability is tied to the notion of "not growing in time." There is no way, however, to know whether an excess ratio grows in time.

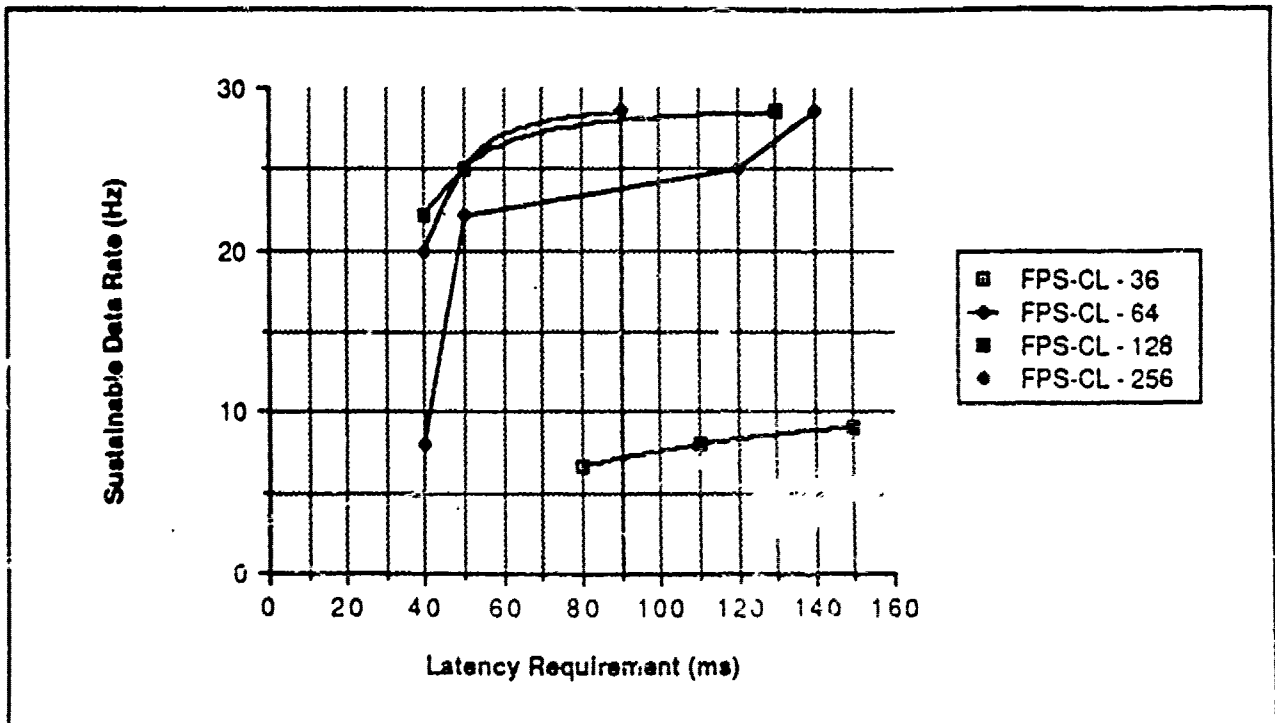


Figure 5.24 Sustainable data rates of FPS Creation Latency

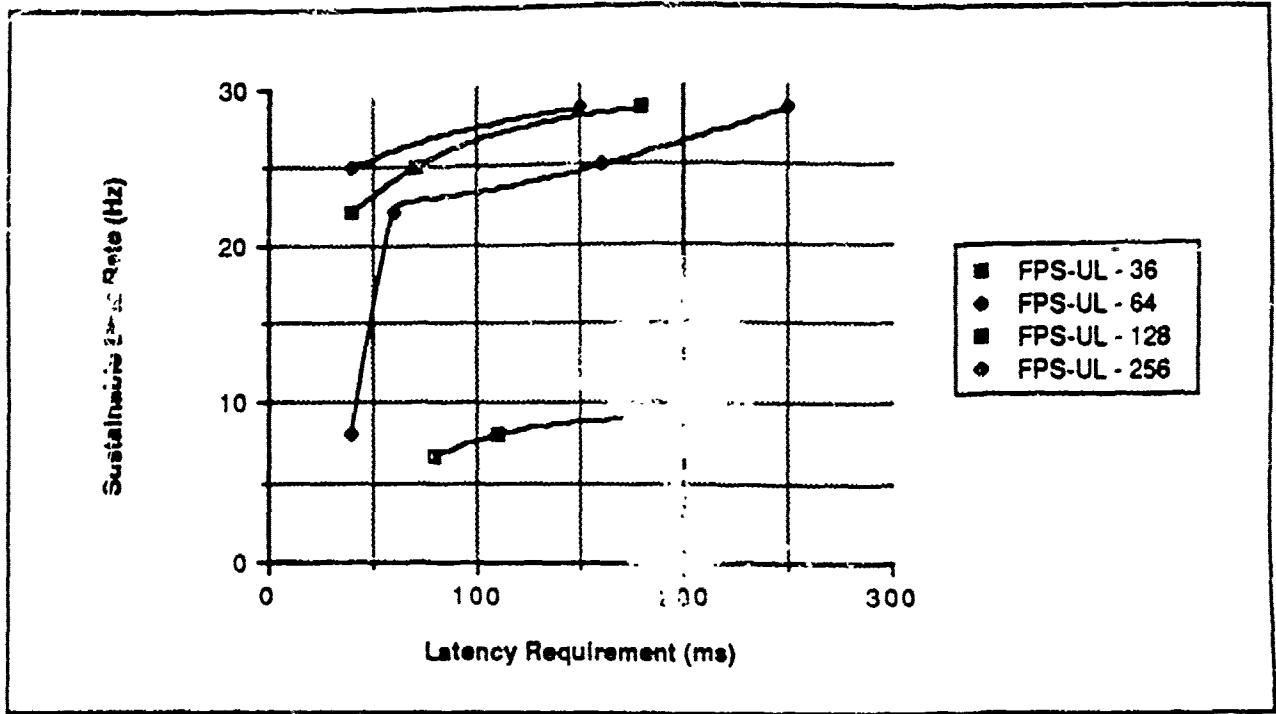


Figure 5.25 Sustainable data rates of FPS Update Latency

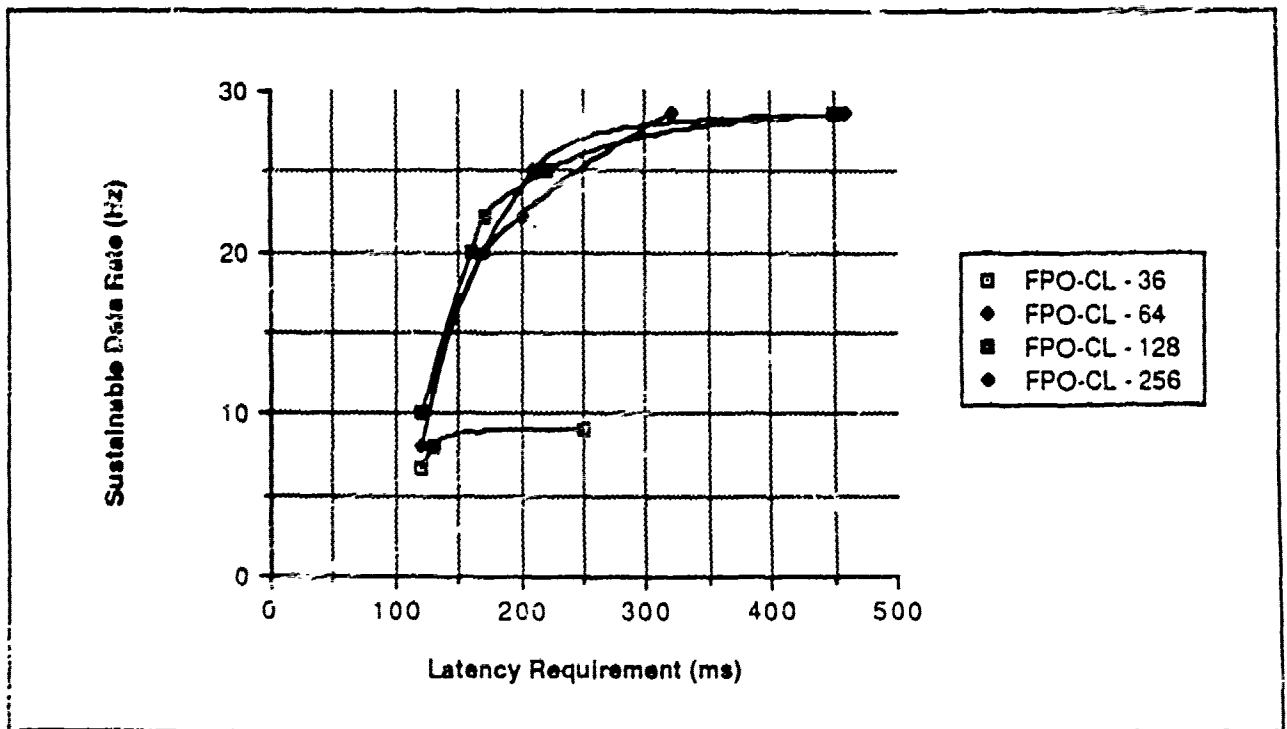


Figure 5.26 Sustainable data rates of FPO Creation Latency

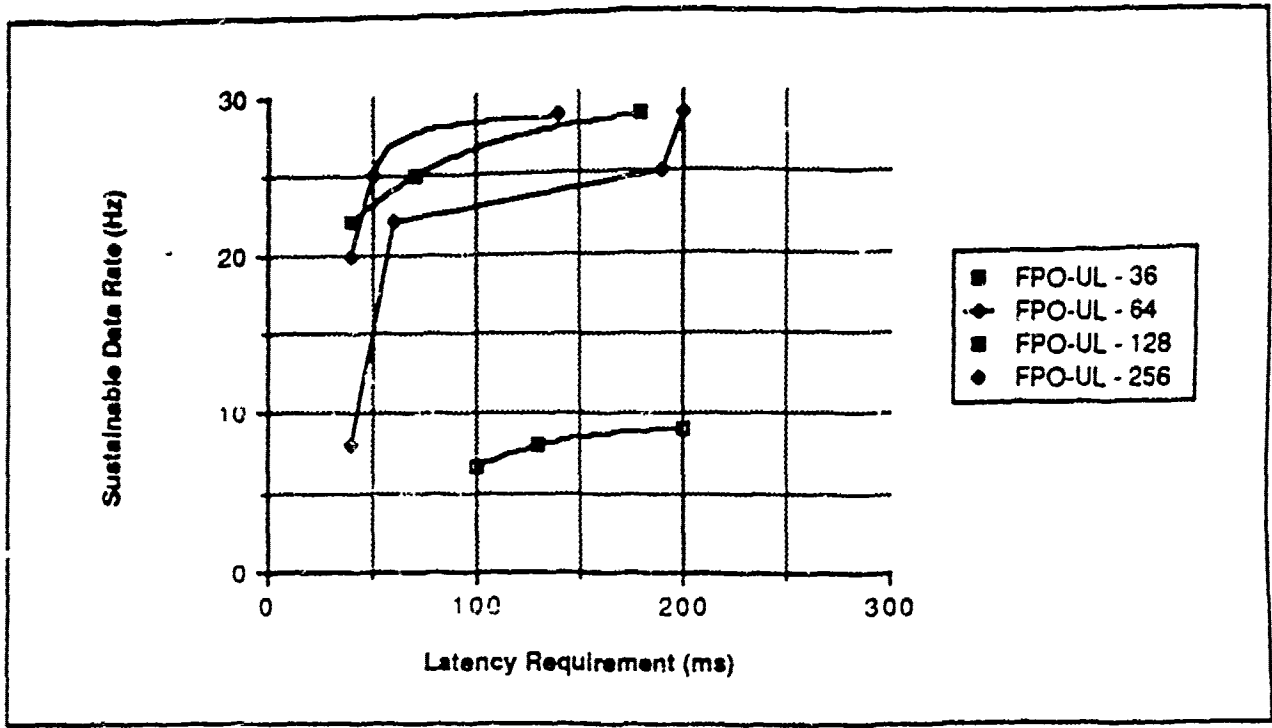


Figure 5.27 Sustainable data rates of FPO Update Latency

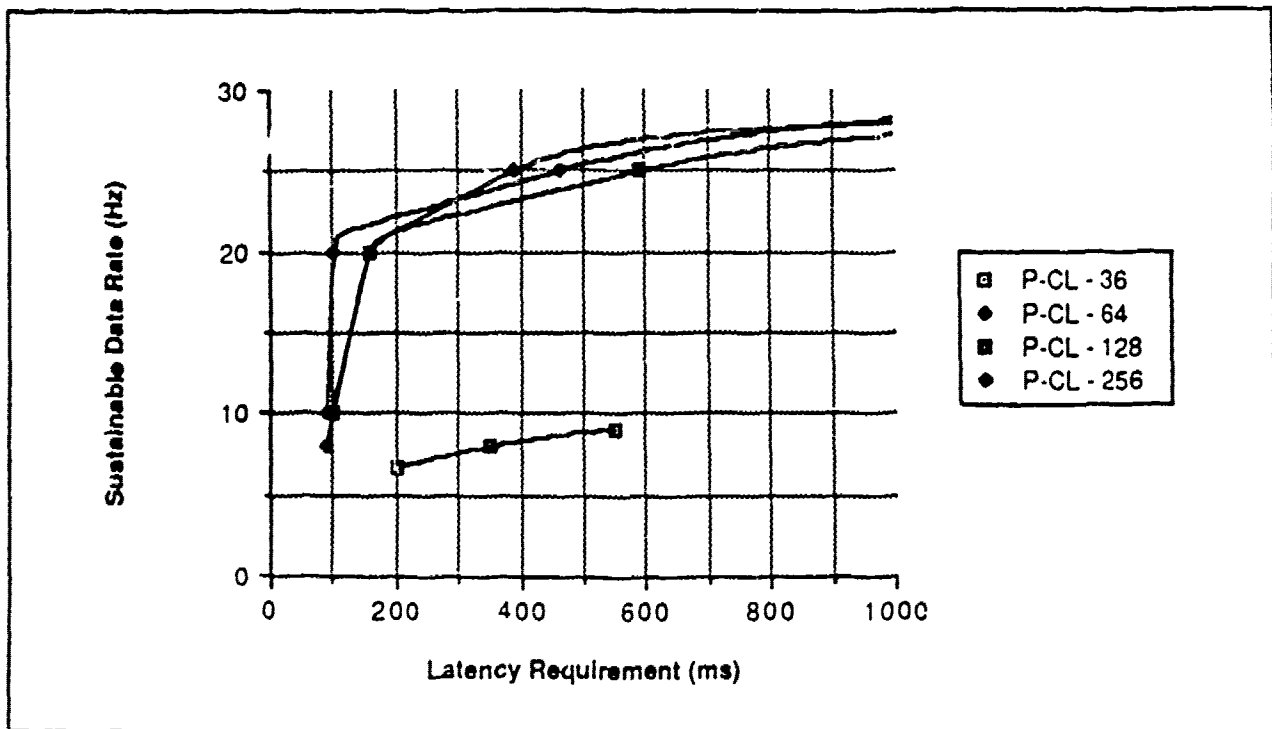


Figure 5.28 Sustainable data rates of P Creation Latency

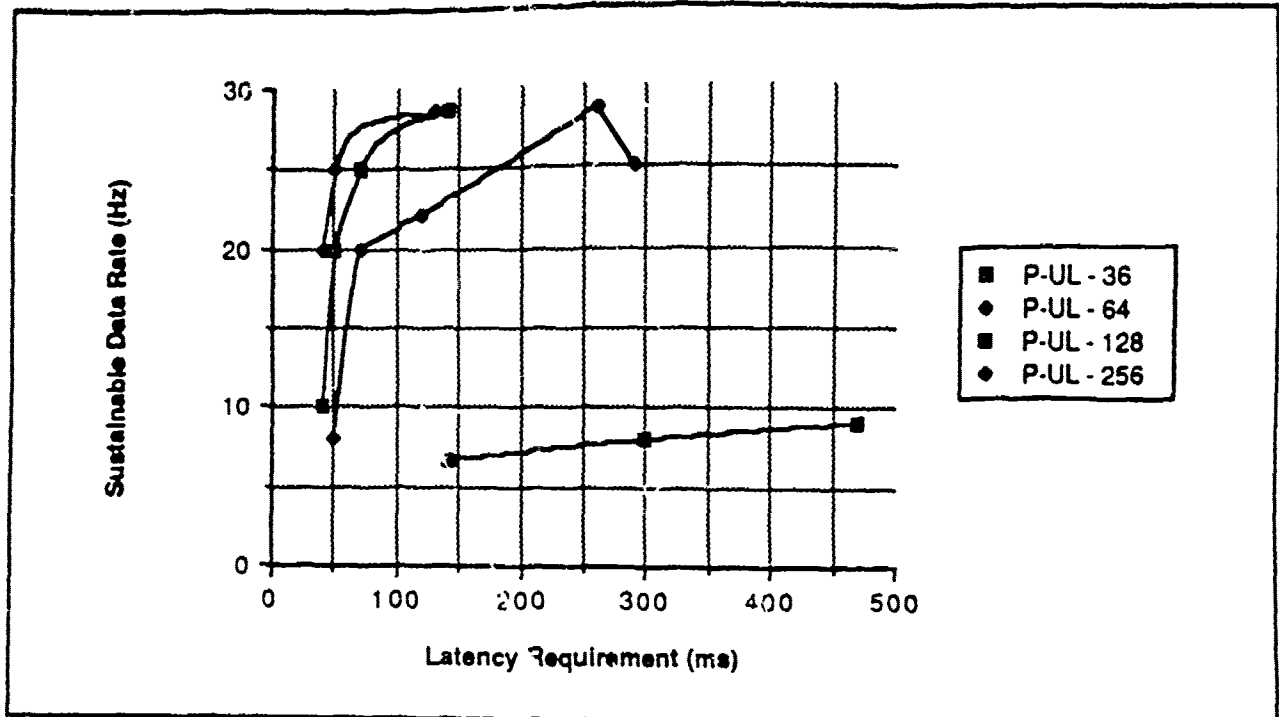


Figure 5.29 Sustainable data rates of P Update Latency

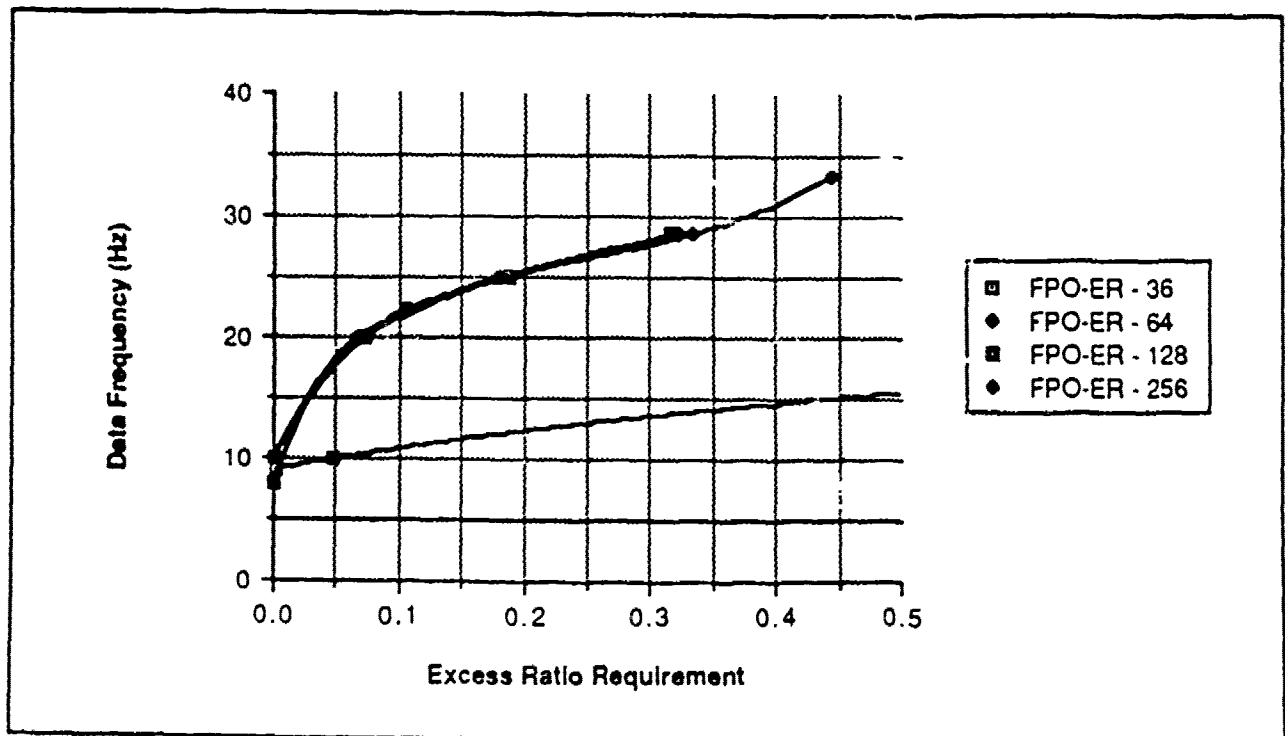


Figure 5.30 Data frequencies that correspond to different FPO Excess Ratios

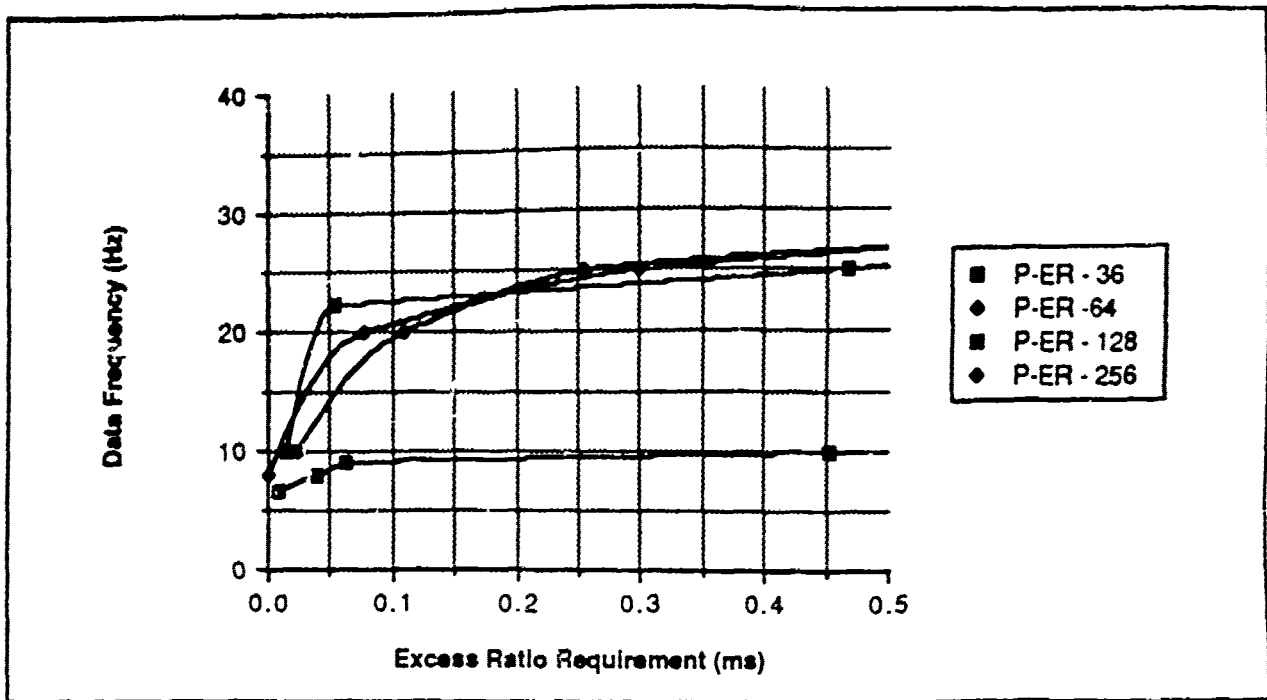


Figure 5.31 Data frequencies that correspond to different P Excess Ratios

Figure 5.32 shows the sustainable data rates of each element of the following performance requirement:

$$qn = \begin{bmatrix} \text{FPS-CL} \\ \text{FPS-UL} \\ \text{FPO-CL} \\ \text{FPO-UL} \\ \text{P-CL} \\ \text{P-UL} \end{bmatrix} = \begin{bmatrix} 100 \\ 80 \\ 200 \\ 100 \\ 300 \\ 150 \end{bmatrix} \text{ ms}$$

$$qI = \begin{bmatrix} \text{FPS-ER} \\ \text{FPO-ER} \\ \text{P-ER} \end{bmatrix} = \begin{bmatrix} 0.00 \\ 0.15 \\ 0.20 \end{bmatrix}$$

The sustainable data rates of grid size 36 that satisfy the elements of the performance requirement vary from 6.67 to 11.66 Hz. For grid size 64, the sustainable data rates vary from 22.22 to 24.21 Hz; for grid size 128, 22.28 to 28.57 Hz; and for grid size 256, 23.25 to 28.57 Hz.¹ Hence, the overall sustainable data rates for grid sizes 36, 64, 128, and 256 are 6.67, 22.22, 22.28, and 23.25 Hz, respectively.

¹ These numbers are generated by interpolation from Figures 5.24-31

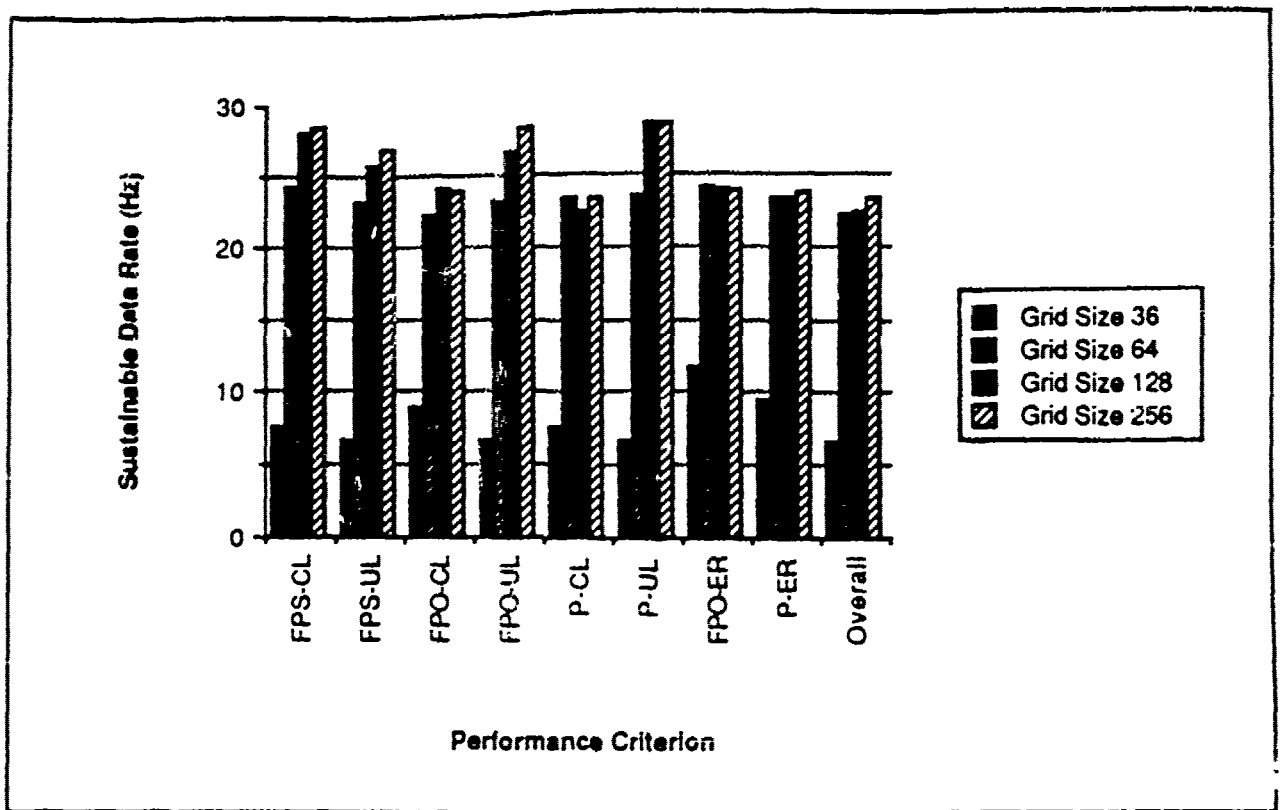


Figure 5.32 The sustainable data rates of each element of qn and ql

The SDR-based speedup curve that corresponds to the performance requirement above is shown in Figure 5.33. As indicated in the previous two sections, there is a significant performance gain from grid size 36 to 64, but not beyond 64. The overall speedup of grid size 256 over 36 is 3.49. The theoretical speedup limit of grid size 256 over 36, i.e. a linear speedup, is not exactly $\frac{256}{36} = 7.11$. Since 31 sites out of those grid sizes are dedicated for managers, only 5 sites are available for dynamic objects on grid size 36, but 220 are available on grid size 256. So the actual theoretical limit is probably closer to $\frac{220}{5} = 44$. Hence, a speedup of 3.49 is very small compared to the theoretical limit.¹

¹ For grid size 64, the demonstrated speedup over grid size 36 is 3.33. The theoretical limit is probably about $\frac{33}{5} = 6.6$. For grid size 128, the speedup is 3.34, and the limit is probably $\frac{97}{5} = 19.4$.

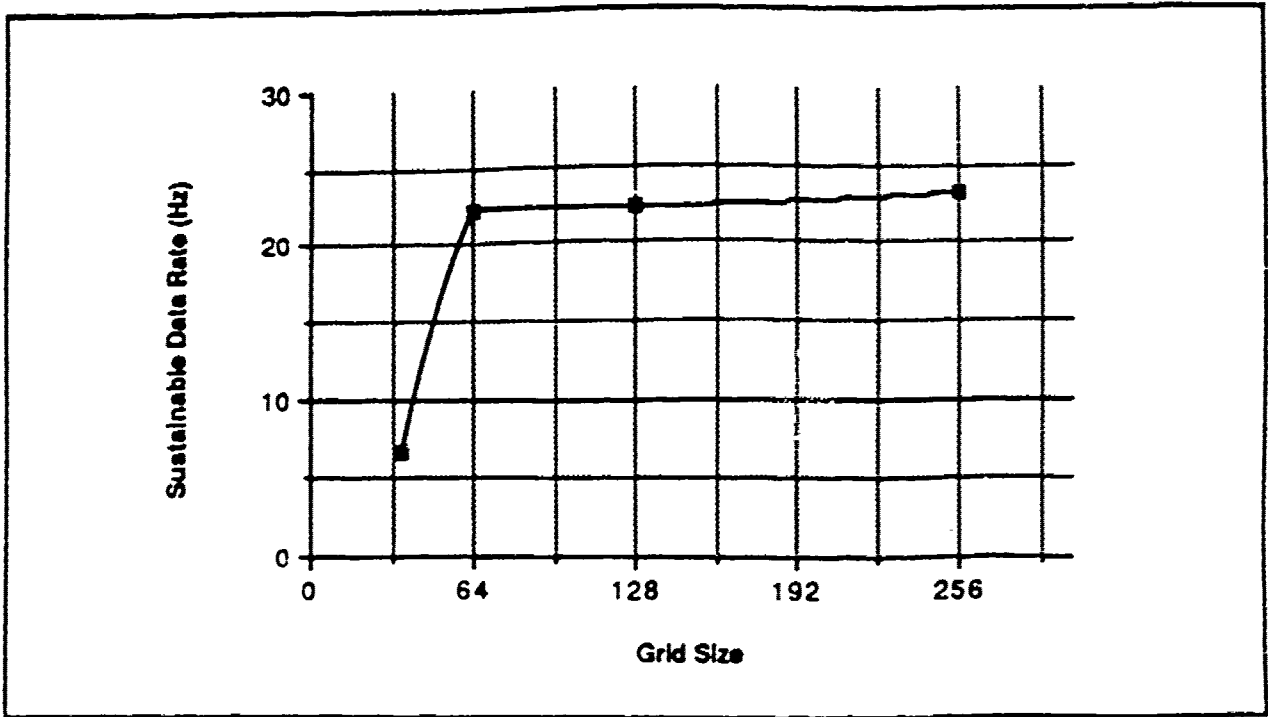


Figure 5.33 A possible SDR-based speedup curve

The corresponding capacity-based speedup curve can similarly be derived. Appendix A6 contains the details of the derivation. Figure 5.34 shows the capacities of each element of the performance requirement. The capacities of grid size 36 vary from 5.60 to 8.55 POR's/scantime; grid size 64, 16.40 to 18.61 POR's/scantime; grid size 128, 17.35 to 20.93 POR's/scantime; and grid size 256, 16.90 to 18.61 POR's/scantime. So the overall capacities for grid sizes 36, 64, 128, and 256 are 5.60, 16.40, 17.35, and 16.90 POR's/scantime, respectively.

Figure 5.35 shows the capacity-based speedup curve that corresponds to the performance requirement. This curve also shows a significant improvement from grid size 36 to 64, but not much beyond 64. The speedup of grid size 128 over 36 is 3.10.¹ It is also very small compared to the theoretical limit.

¹ The speedup of grid size 256 over 36 is somewhat smaller, 3.02. The speedup of grid size 64 over 36 is 2.93. All these capacity-based speedup figures are slightly lower than their corresponding SDR-based figures.

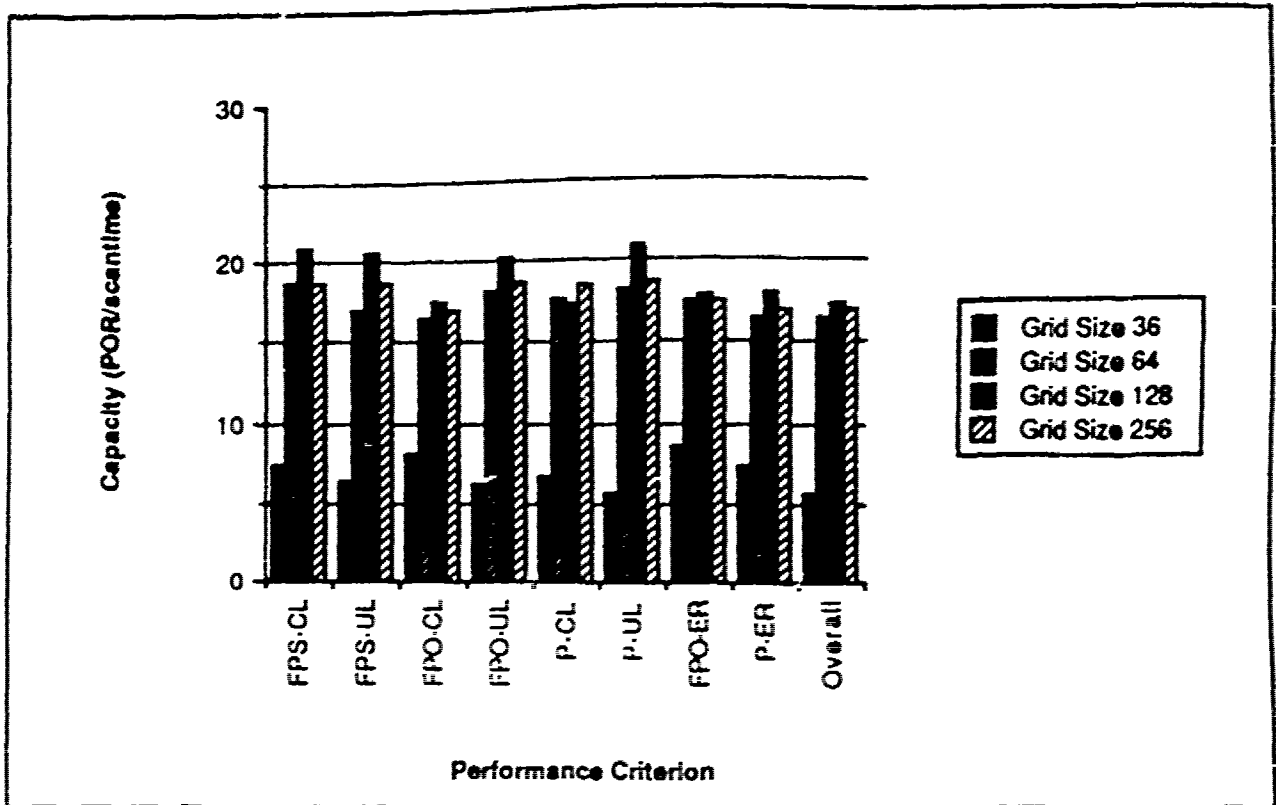


Figure 5.34 The capacities of each element of qn and ql

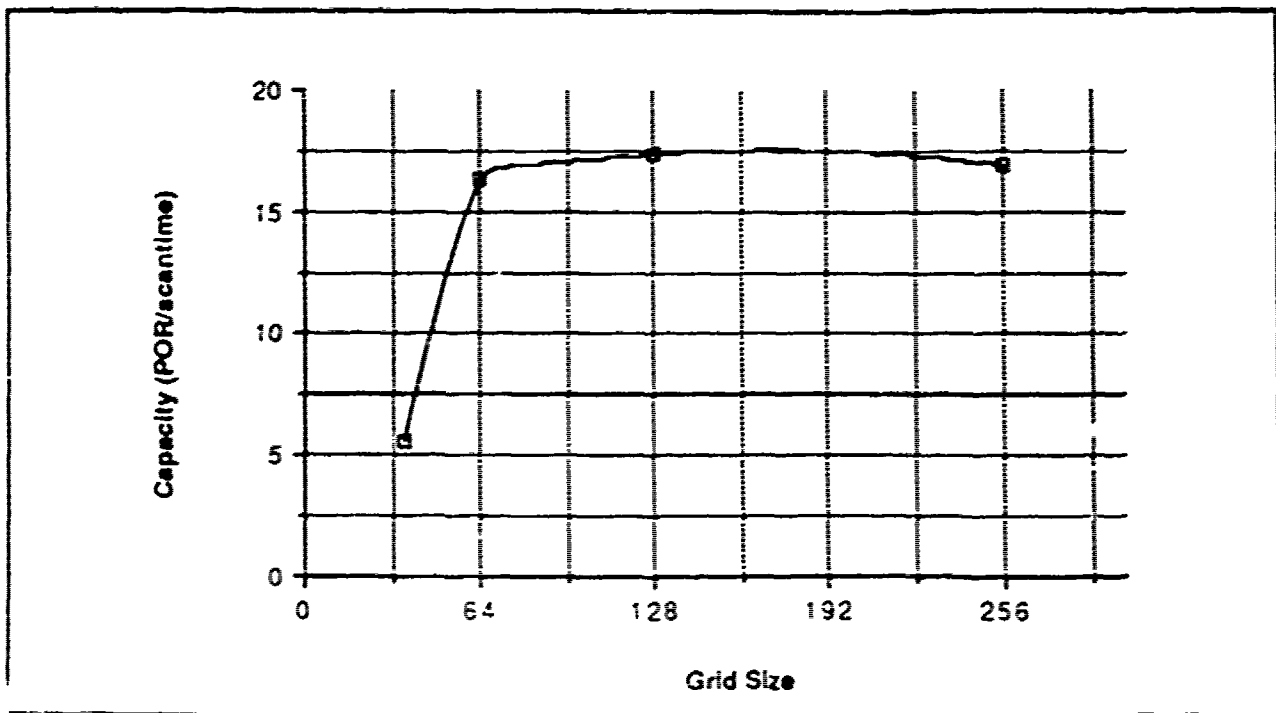


Figure 5.35 A possible capacity-based speedup curve

The SDR-based and capacity-based speedup curves can be normalized in terms of the number of POR's processed per simulation time unit (second), as shown in Figure 5.36. The SDR-based figures are multiplied by the width of the scenario used, C-130, which is 15.74 POR's/scantime. The capacity-based figures are multiplied by the frequency of input data used, which is 20 Hz.

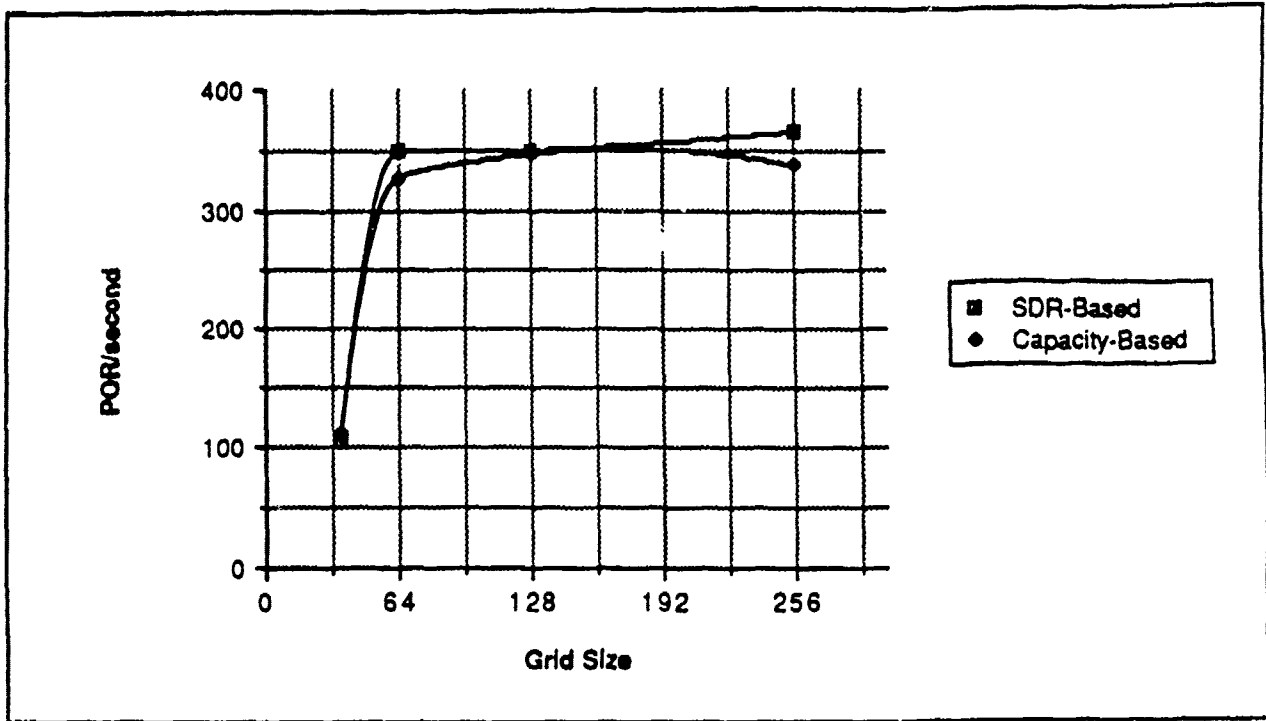


Figure 5.36 The two speedup curves are normalized in terms of POR's per simulation time unit

Of course, the speedup curves above are very much dependent on the performance requirement specified. A tighter performance requirement will eventually eliminate much of the performance gain achieved on larger grid sizes. For example, if the P Excess Ratio is required to be very close to 2.0, as shown in Figure 5.31, the speedup is reduced to only about 1.25. If the performance requirement is too tight, it will not be achievable on any grid size at any data rate or width. A more relaxed performance requirement, if tolerable, can lead to slightly higher speedup. But there is a limit: higher data rate or width that leads to higher latencies and excess ratios is eventually no longer sustainable. At that point, a looser performance requirement does not change anything.

We feel that the small gain in performance across grid sizes is intrinsic to the PA system. Its complex reasoning leads to coarse process granularity. As a result, the level of concurrency achievable is relatively low. The system fails to fully utilize the additional sites available on larger grid sizes.

5.5. Summary of Results

Six parameters of the PA system are studied: free pool lengths, PM multiplier, FPS history ring buffer length, connection search interval, maximum FPO fusion retry, and P waiting period. The free pool mechanism was implemented to reduce dynamic object creation latencies, but the experiment shows that the latencies are not reduced no matter how long the free pools are. The experiment also shows that the PM's help the fusion process by reducing both the P Creation Latency and P Excess Ratio. However, long FPS history ring buffers in the connection process increase the FPO Creation Latency while at the same time decrease the FPO Excess Ratio. A wide connection search interval also causes high FPO Creation Latency, but a too narrow connection search interval causes high FPO Excess Ratio. The FPO fusion retry process is shown to be very critical in improving the speed and quality of the fusion process. Finally, the experiment shows that obsolete P's have to be discarded to prevent the fusion performance from deteriorating.

All of the latencies and the excess ratios, except the FPS Excess Ratio which is constantly zero, go up as the input data frequency or width increases. The latencies and excess ratios increase much more quickly on grid size 36 than on any larger grid sizes. The knees of the latency and excess ratio curves for grid sizes 64, 128, and 256 are about the same. Hence, for a desired level of performance, any significant improvement stops at grid sizes 64.

An SDR-based and a capacity-based speedup curves are generated by plotting sustainable data rate and capacity, respectively, versus grid size, after a performance requirement is specified. Both speedup curves show substantial improvement from grid size 36 to 64, but not beyond grid size 64. The overall SDR-based speedup over grid size 36 is 3.49, while the capacity-based speedup is 3.10. Both figures are small compared to the theoretical limit.

6. Conclusions

Evaluating the performance of a continuous parallel knowledge-based system such as PA proves to be very difficult. The performance cannot be measured in execution time, since the system is continuous. The notion of latency proves to be very useful in indicating how the system keeps up quantitatively with the input data. Excess ratio is also useful as a measure of the quality of the result, but it fails to indicate whether it grows in time. A harder problem is in defining the notion of sustainability. Different definitions could lead to different conclusions of the same experimental results.

For many of the heuristics in PA, the qualitative and quantitative aspects of the performance are proportionally related. This is especially true when the heuristics deal with how much history or knowledge have to be processed. With long history, the system spends too much time to do useless processing with obsolete data. As a result the system cannot keep up with its input data: both the quantitative and qualitative aspects of the performance deteriorate.

The potential speedup via concurrent processing of large and complex systems such as PA is relatively small. While the DA module can achieve 2 order of magnitude speedup [5], PA can achieve only about 1 order of magnitude. The large speedup in DA is due to its simple reasoning. It consists of small and relatively independent processes. PA, on the other hand, is much more complex. PA cannot be decomposed into smaller processes without introducing a high degree of dependency and synchronization among the processes that would be enough to offset the potential gain.

The results in this experiment lead to a conclusion that there is an inverse relationship between the potential speedup via concurrency of a system and its reasoning complexity. This is very unfortunate, since systems are unmistakably becoming more complex and larger. Yet, concurrent processing does not seem to be the answer for a large speedup in their performance.

7. Future Work

Many interesting enhancements and experiments are worth pursuing with the current system. These include:

- Implement and evaluate the effects of variable FPO fusion retries (Section 2.3.3).

- Exploit the free pool mechanism to recycle obsolete dynamic objects (Section 5.3.1).
- Measure and evaluate the performance of PA on a grid size in which there is only one site available for all dynamic objects. This performance is to be used to calculate a more accurate speedup figure over the number of sites available for dynamic objects (Section 5.4).

The next step in the Airtrac development is the design and implementation of the last module, Path Interpretation (PI). As described earlier, the PI module will analyze and interpret information stored in the P's generated by the PA module. Additional sources of information such as aircraft flight plans and intelligence reports are needed to provide continuous and real-time assessments and predictions about aircraft in the monitored airspace.

Acknowledgements

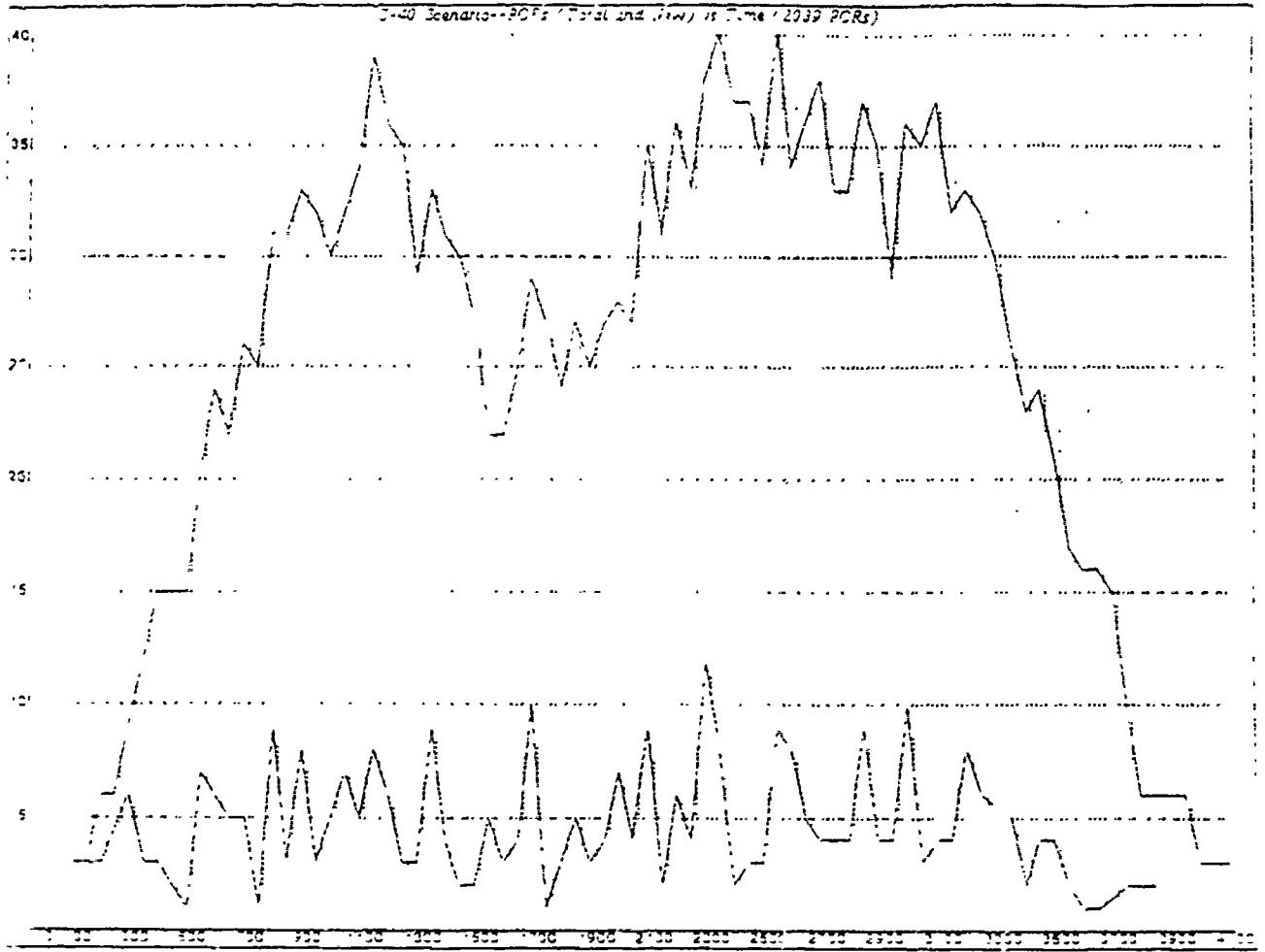
The author wishes to thank Harold Brown for providing valuable guidance and support throughout the course of this work. The author also would like to thank Alan Noble and Chris Rogers who developed most of the PA system, Bruce Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd who built and maintained the CARE/LAMINA system, Hirotohi Maegawa for useful insights, and Bob Engelmere for valuable feedback. The author is indebted to all members of the Advanced Architectures Project for nurturing a rich research environment and for their tireless support. The Symbolic Systems Resources Group of the Knowledge Systems Laboratory provided excellent support of computing environment. Finally, special thanks are due to Edward Feigenbaum for his leadership and support of the Knowledge Systems Laboratory and the Advanced Architectures Project which made this research possible. This work was supported by DARPA Contract F30602-85-C-0012 and NASA Ames Contract NCC 2-220-S1.

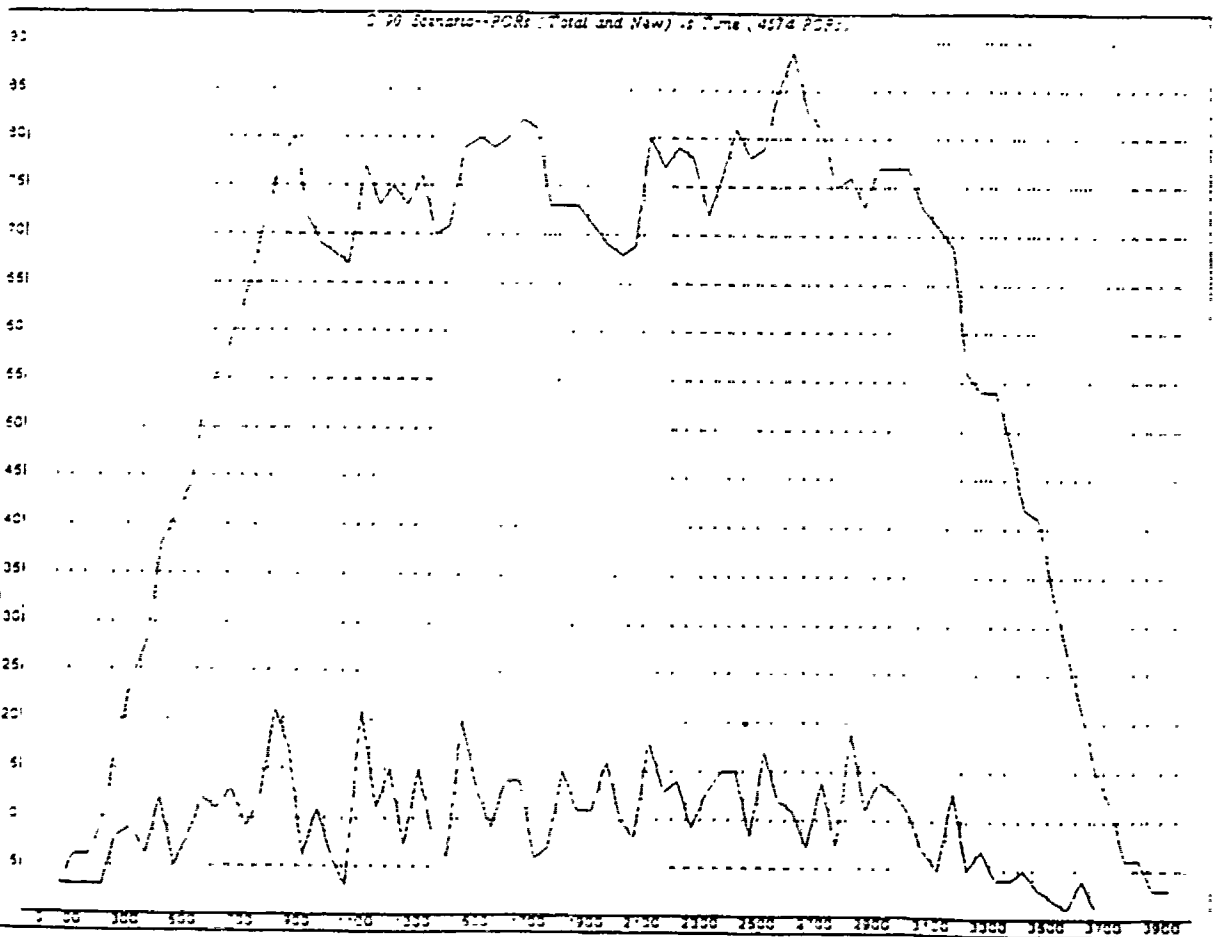
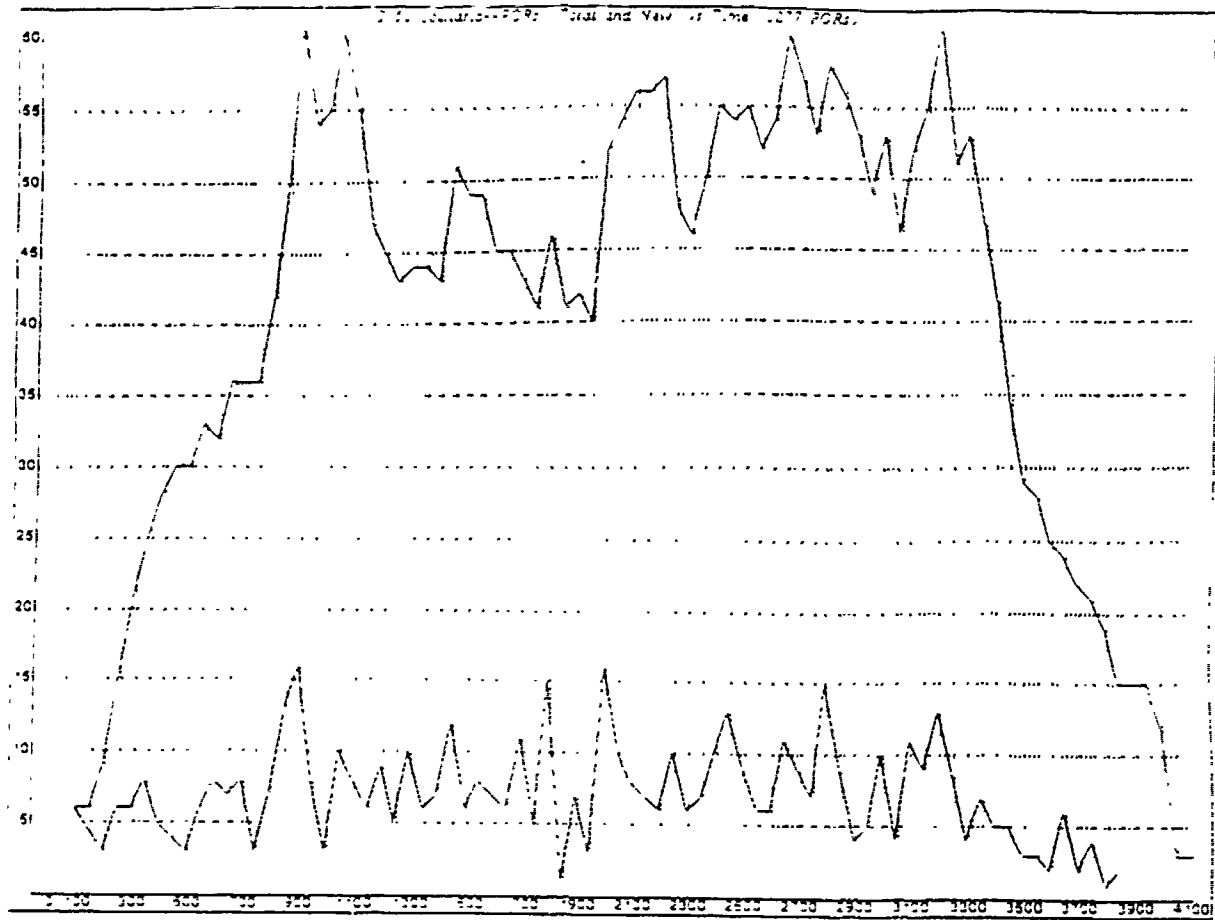
References

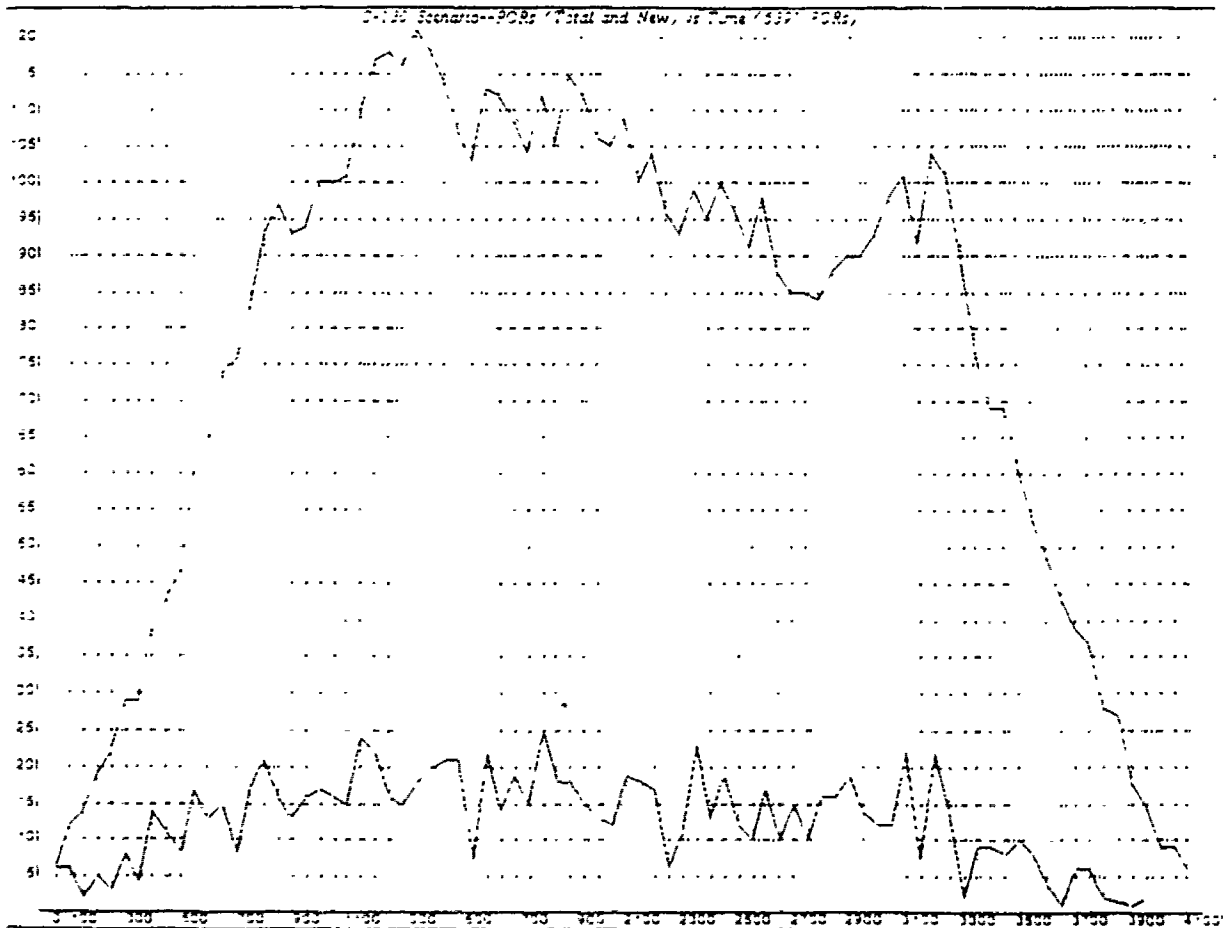
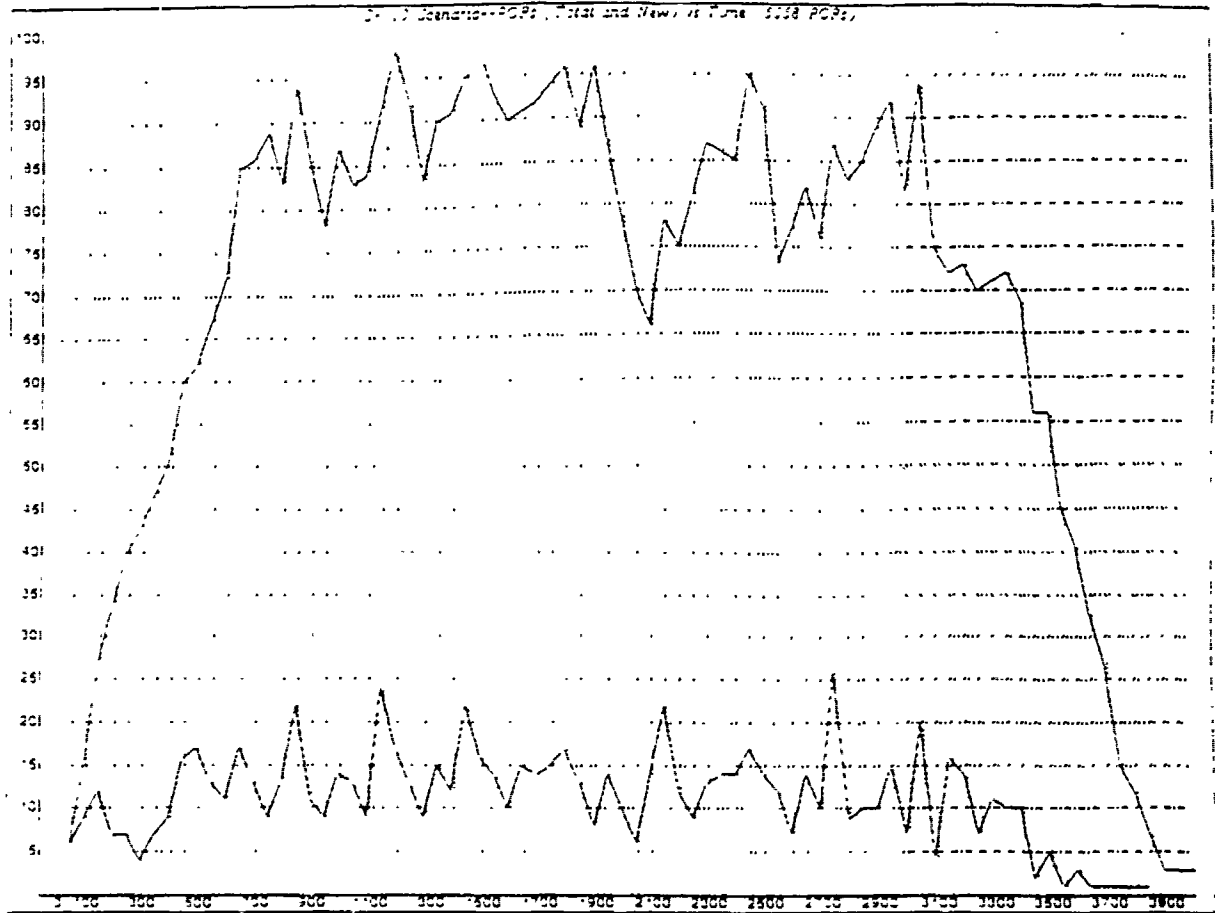
- [1] Harold O. Brown, Eric Schoen, and Bruce A. Delagi. *An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures*, Technical Report STAN-CS-86-1136 or KSL-86-69, Department of Computer Science, Stanford University, October 1986.
- [2] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. *LAMINA: CARE Applications Interface*, Technical Report KSL-86-87, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, November 1987.
- [3] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. *CARE User's Manual*, Technical Report KSL-88-53, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, June 1988.
- [4] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. *An Instrumented Architecture: Simulation System*, Technical Report STAN-CS-87-1148 or KSL-86-36, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, January 1987.
- [5] Russell Nakano and Masafumi Minami. *Experiments with a Knowledge-Based System on a Multiprocessor*, Technical Report KSL-87-61, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, October 1987.
- [6] Alan C. Noble. *ELMA Programmers Guide*, Technical Report KSL-88-42, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, August 1988.
- [7] Alan C. Noble and Everett C. Rogers. *AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor*, Technical Report KSL-88-41, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, June 1988.
- [8] James Rice. *The Advanced Architectures Project*, Technical Report KSL-88-71, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, March 1989.
- [9] Nakul P. Saraiya, Bruce A. Delagi, and Sayuri Nishimura. *Design and Performance Evaluation of a Parallel Report Integration System*, Technical Report KSL-89-16, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, April 1989.

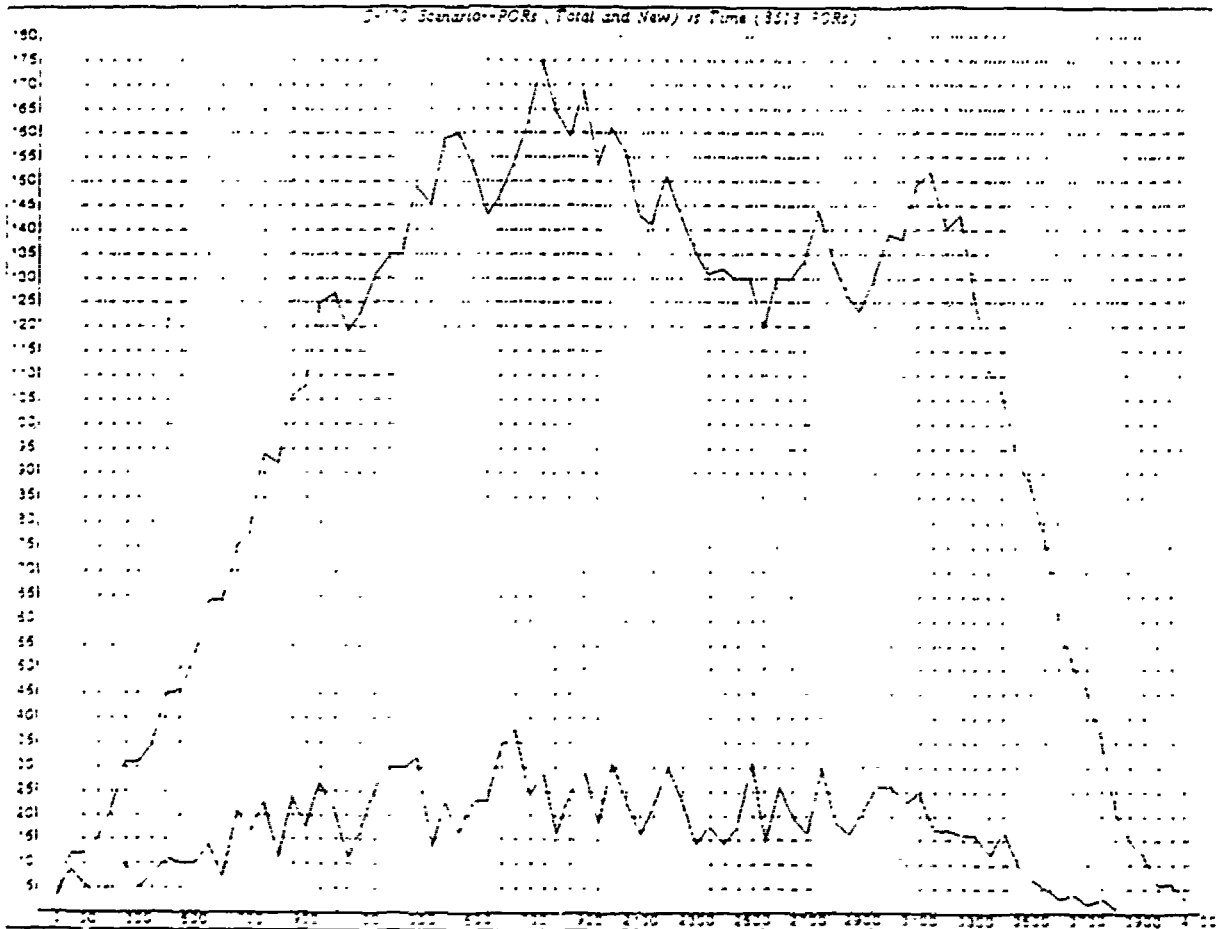
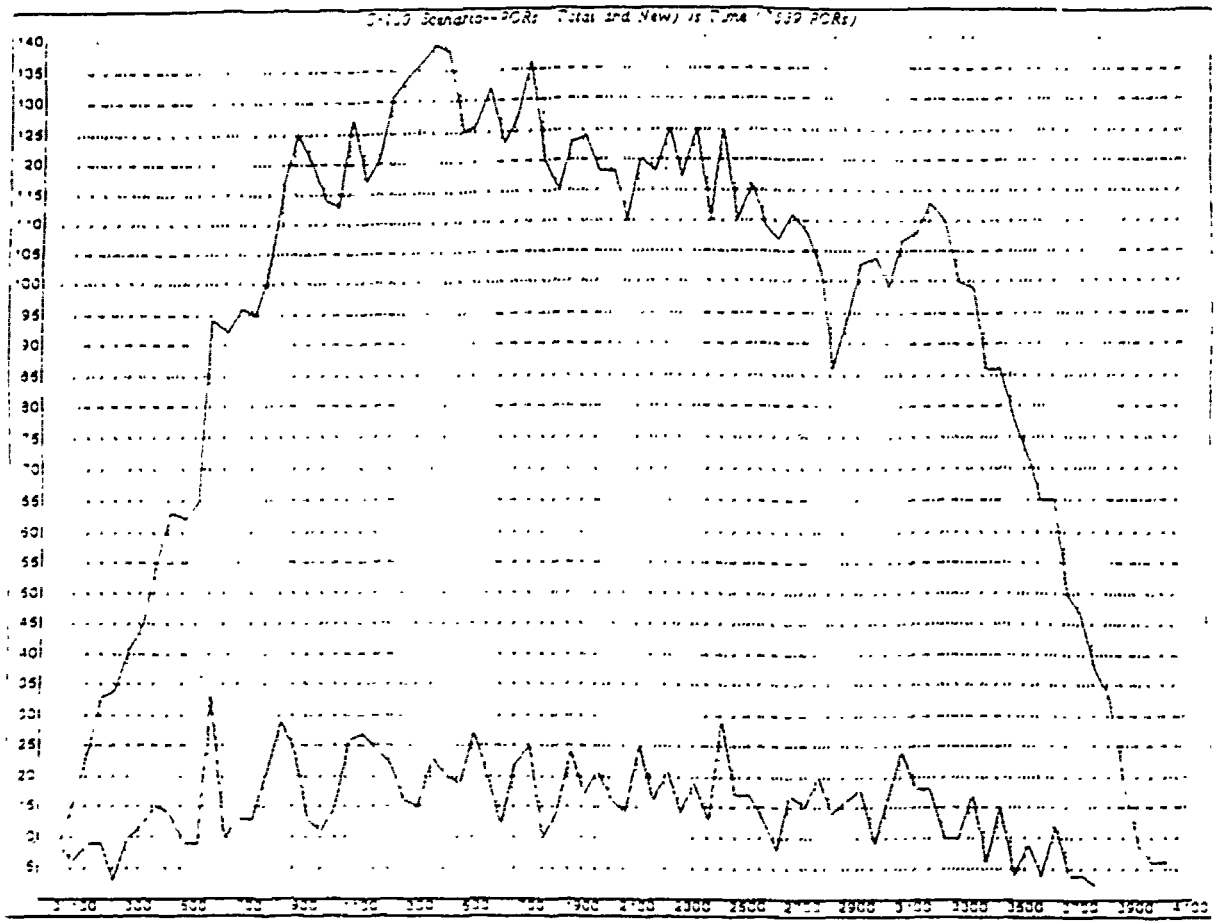
A1. Scenario Profiles

The following seven scenario profiles show the numbers of total and new POR's at every five scantimes.



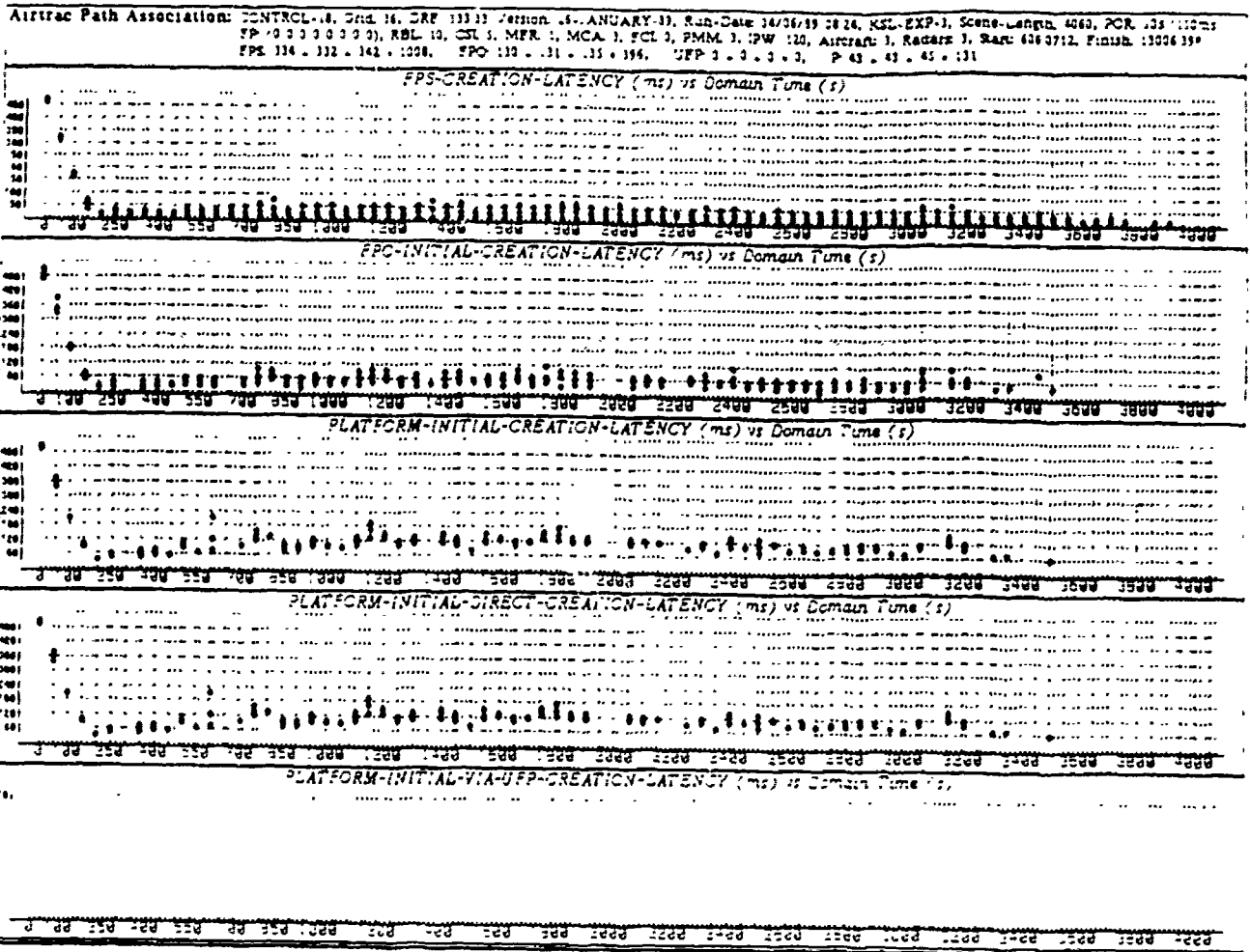




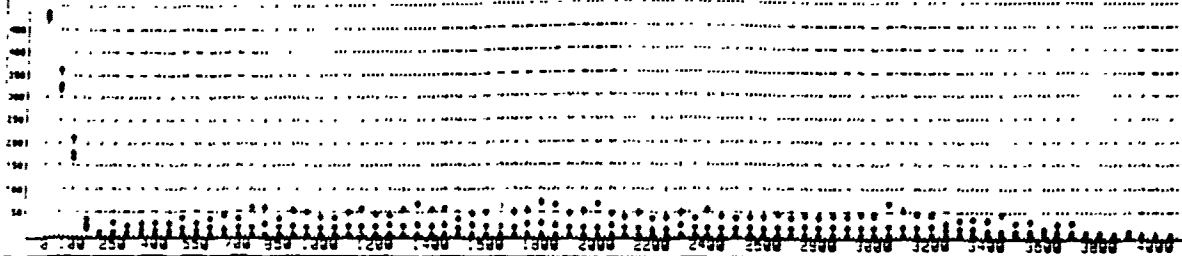


A2. Sample Simulation Results

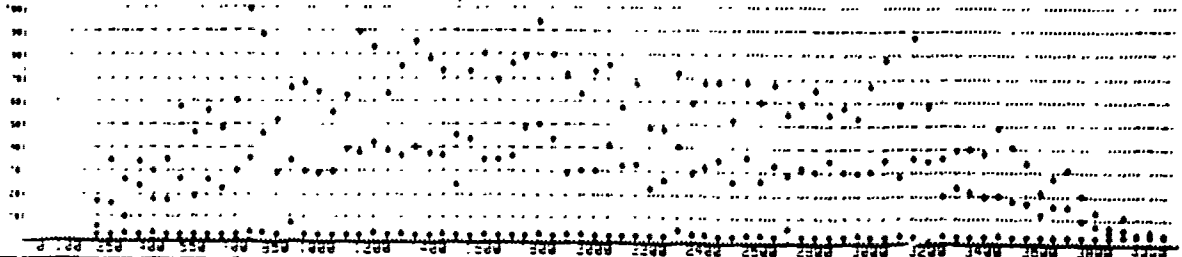
The following figures show sample simulation results: latency reports, cost reports, and queue lengths.



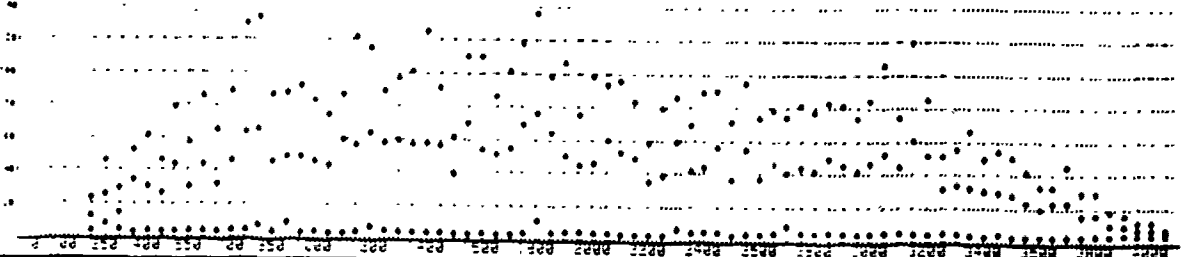
FPS-UPDATE-LATENCY (ms) vs Domain Time (s)



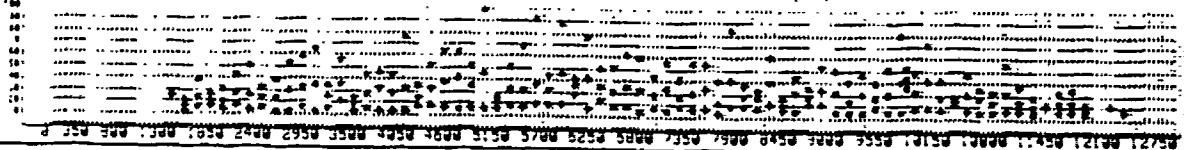
FPO-UPDATE-LATENCY (ms) vs Domain Time (s)



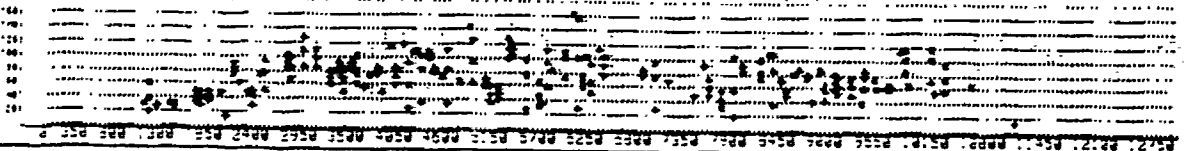
PLATFORM-UPDATE-LATENCY (ms) vs Domain Time (s)



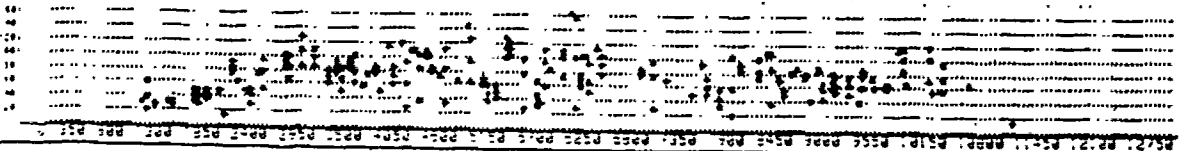
FPS-CONNECTION-COST (ms) vs Simulation Time (ms)



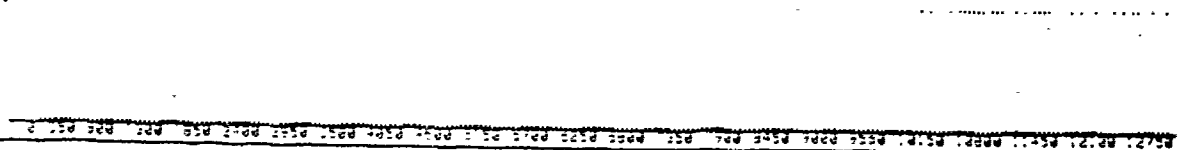
INITIAL-FUSIGN-COST (ms) vs Simulation Time (ms)



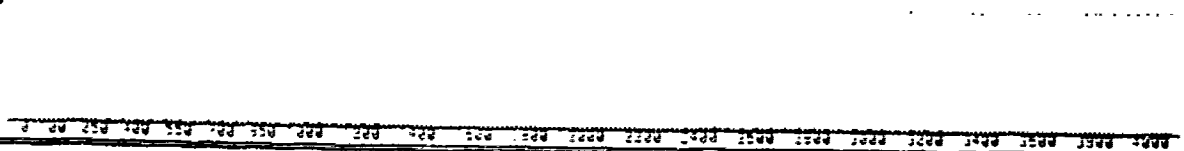
INITIAL-DIRECT-FUSIGN-COST (ms) vs Simulation Time (ms)



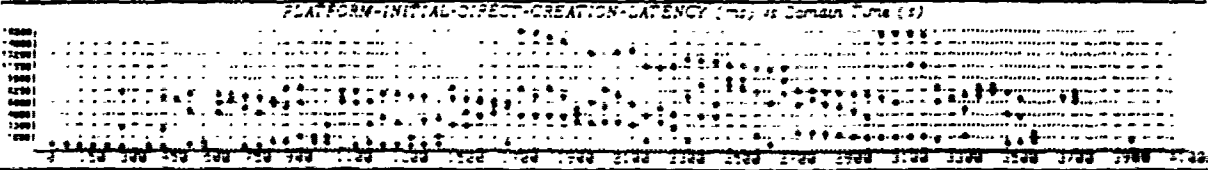
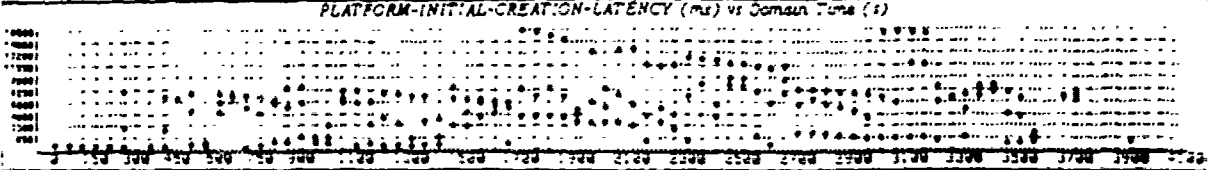
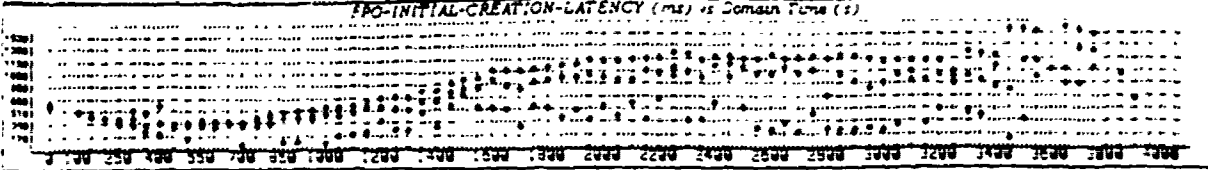
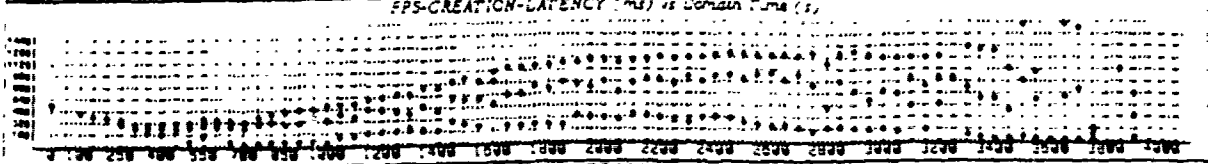
INITIAL-TRAFFIC-FUSIGN-COST (ms) vs Simulation Time (ms)



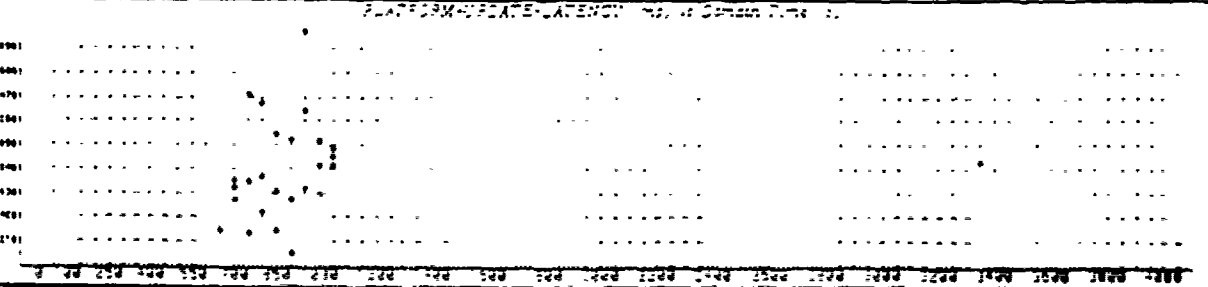
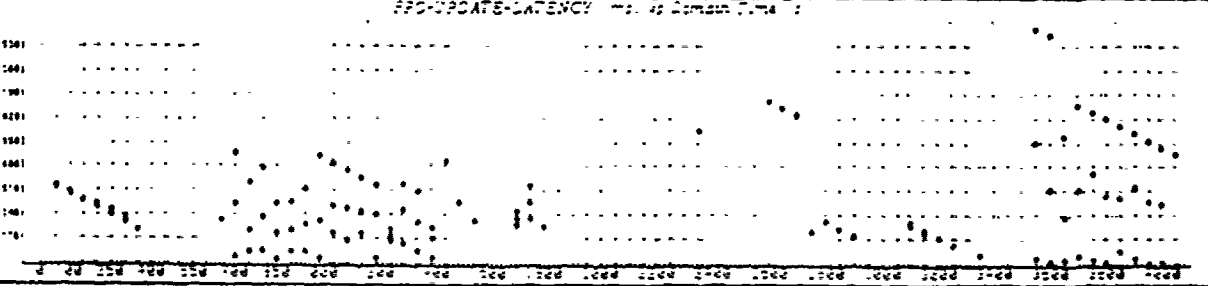
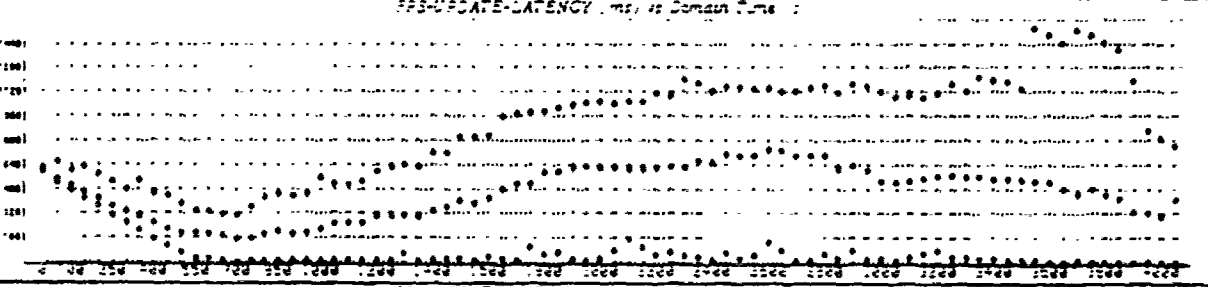
INITIAL-REP-DEGRATION (ms) vs Domain Time (s)



Airtrac Path Association, CONTROL 3, DRY 15, 315, Version 16-JANUARY-89, Run Date 12/21/88, 11:45, KIL EXP-1, Comp-Length 643, PCR 115, 11864
 FP 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600 1700 1800 1900 2000 2100 2200 2300 2400 2500 2600 2700 2800 2900 3000 3100 3200
 FPO 150 157 159 163, UPP 1 2 3 4 5, P 155 161 166 168

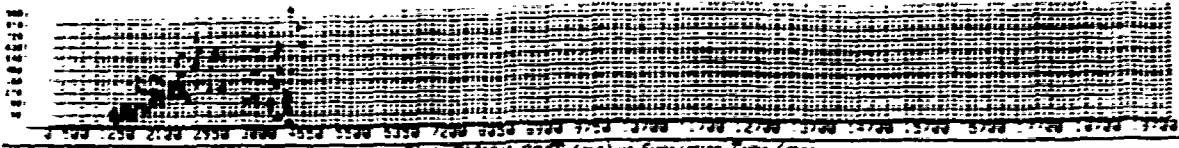


Airtrac Path Association, CONTROL 3, DRY 15, 315, Version 16-JANUARY-89, Run Date 12/21/88, 11:45, KIL EXP-1, Comp-Length 643, PCR 115, 11864
 FP 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600 1700 1800 1900 2000 2100 2200 2300 2400 2500 2600 2700 2800 2900 3000 3100 3200
 FPO 150 157 159 163, UPP 1 2 3 4 5, P 155 161 166 168

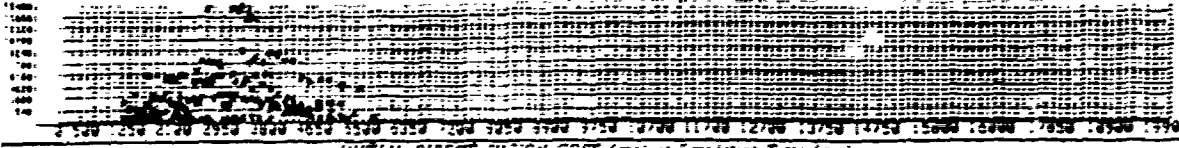


Airtrac Path Association. CONTRACT # 102-1-097. 10 JANUARY 89 Run-Date 14-02-89 17:45. KIL-EXP-1, Some-Length, 4648. FOR: 102-1-097.
PP: 102-1-097-001-101. MCA: 102-1-097-002-006. AMCAR: 1. RADAR: 1. STAT: 607-6677. FIDUC: 102-1-097-001-101-101.
EPS: 102-1-097-001-101. FPO: 102-1-097-001-101. EXP: 102-1-097-001-101. P: 102-1-097-001-101-101

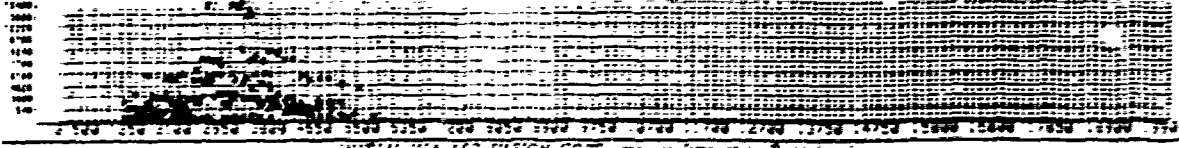
FPS-CONNECTION-COST (ms) vs Simulation Time (ms)



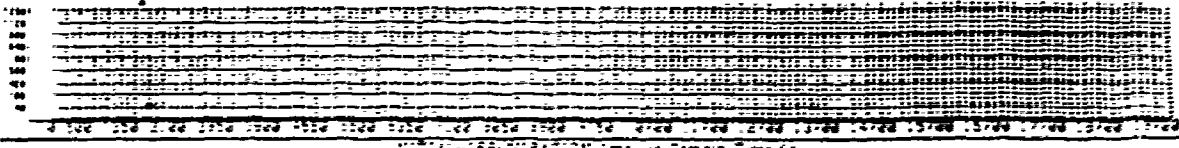
INITIAL-FUSION-COST (ms) vs Simulation Time (ms)



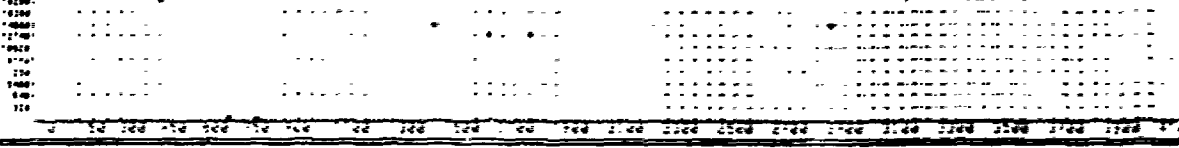
INITIAL-DIRECT-FUSION-COST (ms) vs Simulation Time (ms)



INITIAL-VIA-JFP-FUSION-COST (ms) vs Simulation Time (ms)

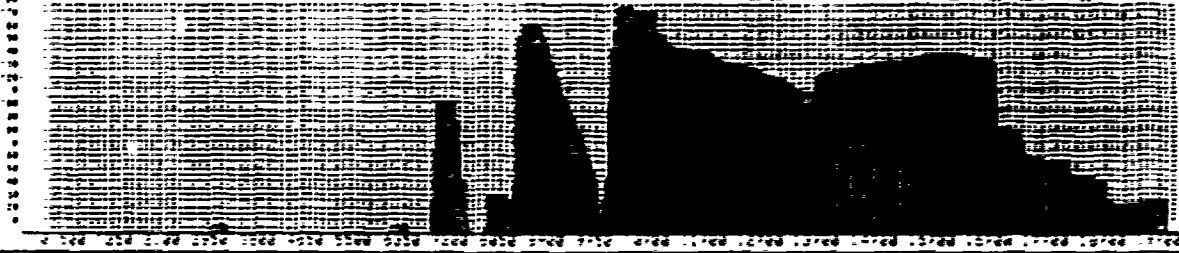


INITIAL-JFP-OCCUPATION (ms) vs Simulation Time (s)

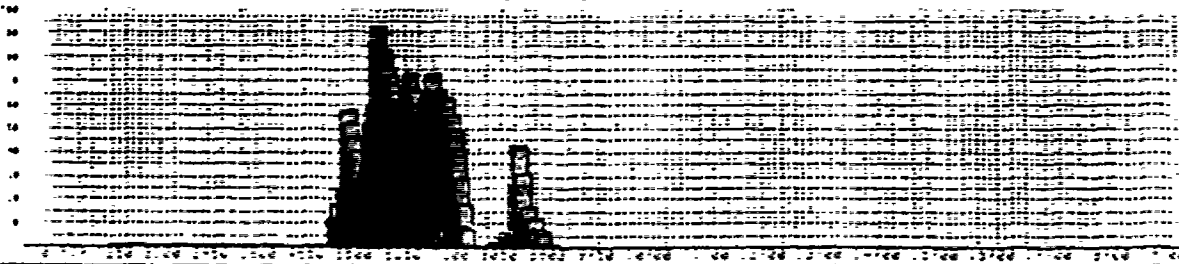


Airtrac Path Association. CONTRACT # 102-1-097. 10 JANUARY 89 Run-Date 14-02-89 17:45. KIL-EXP-1, Some-Length, 4648. FOR: 102-1-097.
PP: 102-1-097-001-101. MCA: 102-1-097-002-006. AMCAR: 1. RADAR: 1. STAT: 607-6677. FIDUC: 102-1-097-001-101-101.
EPS: 102-1-097-001-101. FPO: 102-1-097-001-101. EXP: 102-1-097-001-101. P: 102-1-097-001-101-101

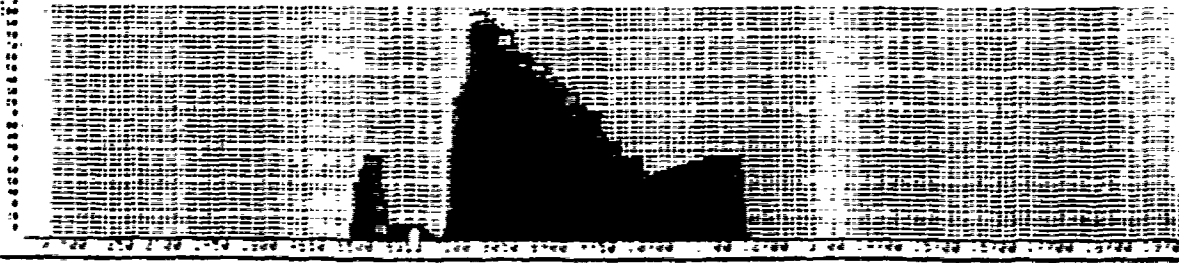
GPS-TYPE-A Queue Length vs Simulation Time (ms)



GPS-TYPE-B Queue Length vs Simulation Time (ms)



GPS-TYPE-C Queue Length vs Simulation Time (ms)



A3. Sample Load Summary

The following is a sample load summary of a simulation run on the grid size 128. Each line starts with the location of a site, the total number of objects or processes that reside on that site, and the kinds of objects or processes and their corresponding numbers. The first 31 sites listed are reserved for managers; they are allocated statically at initialization. The rest are for dynamic objects, which are allocated dynamically and randomly at run time.

```
(16 8): 2 processes ((PAI . 1) (DAS . 1))
(16 7): 1 processes ((FPM . 1))
(16 6): 1 processes ((FPM . 1))
(16 5): 1 processes ((FPM . 1))
(16 4): 1 processes ((FPM . 1))
(16 3): 1 processes ((FPM . 1))
(16 2): 1 processes ((FPM . 1))
(16 1): 1 processes ((FPM . 1))
(15 8): 1 processes ((FPM . 1))
(15 7): 1 processes ((FPM . 1))
(15 6): 1 processes ((FPC . 1))
(15 5): 1 processes ((FPC . 1))
(15 4): 1 processes ((FPC . 1))
(15 3): 1 processes ((FPC . 1))
(15 2): 1 processes ((FPC . 1))
(15 1): 1 processes ((FPC . 1))
(14 8): 1 processes ((FPC . 1))
(14 7): 1 processes ((FPC . 1))
(14 6): 1 processes ((FPC . 1))
(14 5): 1 processes ((CPM . 1))
(14 4): 1 processes ((CPM . 1))
(14 3): 1 processes ((CPM . 1))
(14 2): 1 processes ((PM . 1))
(14 1): 1 processes ((PM . 1))
(13 8): 1 processes ((PM . 1))
(13 7): 1 processes ((PM . 1))
(13 6): 1 processes ((PM . 1))
(13 5): 1 processes ((PM . 1))
(13 4): 1 processes ((PM . 1))
(13 3): 1 processes ((PM . 1))
(13 2): 1 processes ((PM . 1))
(13 1): 14 processes ((FPS . 6) (FPO . 5) (P . 3))
(12 8): 27 processes ((FPS . 14) (FPO . 11) (P . 2))
(12 7): 22 processes ((FPS . 17) (FPO . 5))
(12 6): 27 processes ((FPS . 17) (FPO . 6) (P . 4))
(12 5): 23 processes ((FPS . 17) (FPO . 4) (UFP . 1) (P . 1))
(12 4): 17 processes ((FPS . 10) (FPO . 4) (P . 2) (UFP . 1))
(12 3): 22 processes ((FPS . 13) (FPO . 7) (P . 2))
(12 2): 21 processes ((FPS . 13) (FPO . 5) (P . 2) (UFP . 1))
(12 1): 18 processes ((FPS . 15) (FPO . 3))
```

(11 8): 17 processes ((FPS . 11) (FPO . 4) (P . 2))
 (11 7): 20 processes ((FPS . 14) (FPO . 4) (P . 2))
 (11 6): 11 processes ((FPS . 10) (FPO . 1))
 (11 5): 19 processes ((FPS . 12) (FPO . 6) (P . 1))
 (11 4): 16 processes ((FPS . 10) (FPO . 5) (P . 1))
 (11 3): 17 processes ((FPS . 7) (FPO . 6) (P . 4))
 (11 2): 19 processes ((FPS . 13) (FPO . 5) (P . 1))
 (11 1): 15 processes ((FPS . 13) (FPO . 1) (P . 1))
 (10 8): 11 processes ((FPS . 7) (FPO . 4))
 (10 7): 16 processes ((FPS . 10) (FPO . 5) (P . 1))
 (10 6): 18 processes ((FPS . 12) (FPO . 4) (UFP . 2))
 (10 5): 12 processes ((FPS . 8) (FPO . 3) (P . 1))
 (10 4): 25 processes ((FPS . 16) (FPO . 6) (P . 3))
 (10 3): 18 processes ((FPS . 9) (FPO . 5) (P . 3) (UFP . 1))
 (10 2): 15 processes ((FPS . 9) (FPO . 5) (P . 1))
 (10 1): 17 processes ((FPO . 7) (FPS . 6) (P . 3) (UFP . 1))
 (9 8): 14 processes ((FPS . 9) (FPO . 4) (P . 1))
 (9 7): 16 processes ((FPS . 9) (FPO . 6) (P . 1))
 (9 6): 11 processes ((FPS . 9) (P . 1) (FPO . 1))
 (9 5): 21 processes ((FPO . 9) (FPS . 9) (P . 2) (UFP . 1))
 (9 4): 17 processes ((FPS . 11) (FPO . 5) (P . 1))
 (9 3): 18 processes ((FPS . 15) (FPO . 3))
 (9 2): 23 processes ((FPS . 15) (FPO . 5) (P . 3))
 (9 1): 18 processes ((FPS . 9) (FPO . 5) (P . 4))
 (8 8): 17 processes ((FPS . 9) (FPO . 6) (P . 2))
 (8 7): 11 processes ((FPO . 5) (FPS . 5) (P . 1))
 (8 6): 18 processes ((FPS . 11) (FPO . 4) (P . 2) (UFP . 1))
 (8 5): 20 processes ((FPS . 10) (FPO . 7) (P . 3))
 (8 4): 18 processes ((FPS . 9) (FPO . 5) (P . 4))
 (8 3): 12 processes ((FPS . 9) (FPO . 2) (P . 1))
 (8 2): 20 processes ((FPS . 12) (FPO . 6) (P . 2))
 (8 1): 23 processes ((FPS . 15) (FPO . 4) (P . 3) (UFP . 1))
 (7 8): 23 processes ((FPS . 16) (FPO . 6) (P . 1))
 (7 7): 16 processes ((FPS . 10) (FPO . 4) (UFP . 1) (P . 1))
 (7 6): 10 processes ((FPS . 7) (P . 2) (FPO . 1))
 (7 5): 12 processes ((FPS . 8) (FPO . 4))
 (7 4): 17 processes ((FPS . 11) (FPO . 6))
 (7 3): 23 processes ((FPS . 13) (P . 6) (FPO . 4))
 (7 2): 16 processes ((FPS . 11) (FPO . 4) (P . 1))
 (7 1): 14 processes ((FPS . 9) (FPO . 3) (P . 2))
 (6 8): 21 processes ((FPS . 13) (FPO . 4) (P . 3) (UFP . 1))
 (6 7): 16 processes ((FPS . 9) (FPO . 5) (P . 2))
 (6 6): 26 processes ((FPS . 12) (FPO . 9) (P . 5))
 (6 5): 15 processes ((FPS . 14) (FPO . 1))
 (6 4): 10 processes ((FPS . 7) (FPO . 2) (P . 1))
 (6 3): 10 processes ((FPS . 7) (FPO . 3))
 (6 2): 14 processes ((FPS . 10) (P . 2) (UFP . 1) (FPO . 1))
 (6 1): 25 processes ((FPS . 16) (FPO . 6) (P . 3))
 (5 8): 13 processes ((FPS . 8) (FPO . 2) (P . 2) (UFP . 1))
 (5 7): 12 processes ((FPS . 5) (FPO . 4) (P . 3))
 (5 6): 23 processes ((FPS . 11) (FPO . 10) (P . 2))
 (5 5): 18 processes ((FPS . 10) (FPO . 6) (P . 2))
 (5 4): 23 processes ((FPS . 15) (FPO . 7) (P . 1))
 (5 3): 25 processes ((FPS . 18) (FPO . 7))

(5 2): 16 processes ((FPS . 10) (FPO . 5) (P . 1))
 (5 1): 14 processes ((FPO . 7) (FPS . 5) (P . 2))
 (4 8): 13 processes ((FPS . 6) (FPO . 4) (P . 3))
 (4 7): 18 processes ((FPS . 11) (FPO . 5) (P . 2))
 (4 6): 20 processes ((FPS . 12) (P . 4) (FPO . 4))
 (4 5): 18 processes ((FPS . 12) (FPO . 5) (P . 1))
 (4 4): 11 processes ((FPS . 6) (FPO . 3) (P . 2))
 (4 3): 16 processes ((FPS . 12) (FPO . 3) (P . 1))
 (4 2): 22 processes ((FPS . 15) (FPO . 5) (P . 2))
 (4 1): 13 processes ((FPS . 7) (FPO . 5) (P . 1))
 (3 8): 13 processes ((FPS . 9) (FPO . 3) (P . 1))
 (3 7): 24 processes ((FPS . 16) (FPO . 6) (P . 2))
 (3 6): 21 processes ((FPS . 15) (FPO . 6))
 (3 5): 20 processes ((FPS . 11) (FPO . 5) (P . 4))
 (3 4): 13 processes ((FPS . 7) (FPO . 3) (P . 3))
 (3 3): 15 processes ((FPO . 7) (FPS . 6) (UFP . 1) (P . 1))
 (3 2): 24 processes ((FPS . 15) (FPO . 6) (P . 3))
 (3 1): 15 processes ((FPS . 9) (FPO . 3) (P . 3))
 (2 8): 19 processes ((FPS . 13) (FPO . 5) (P . 1))
 (2 7): 13 processes ((FPS . 11) (FPO . 2))
 (2 6): 21 processes ((FPO . 11) (FPS . 10))
 (2 5): 10 processes ((FPS . 7) (FPO . 3))
 (2 4): 22 processes ((FPS . 15) (FPO . 4) (P . 3))
 (2 3): 22 processes ((FPS . 16) (FPO . 5) (P . 1))
 (2 2): 10 processes ((FPS . 6) (P . 2) (FPO . 2))
 (2 1): 20 processes ((FPS . 12) (FPO . 5) (P . 3))
 (1 8): 12 processes ((FPS . 10) (UFP . 1) (FPO . 1))
 (1 7): 20 processes ((FPS . 10) (FPO . 7) (P . 3))
 (1 6): 22 processes ((FPS . 13) (FPO . 8) (UFP . 1))
 (1 5): 18 processes ((FPS . 8) (FPO . 7) (P . 2) (UFP . 1))
 (1 4): 16 processes ((FPS . 11) (FPO . 4) (P . 1))
 (1 3): 14 processes ((FPS . 8) (P . 3) (FPO . 3))
 (1 2): 11 processes ((FPS . 6) (FPO . 5))
 (1 1): 18 processes ((FPS . 9) (FPO . 6) (LEXICAL-CLOSURE . 2) (P . 1))

A4. The Effects of the Frequency of Input Data

This appendix contains the complete graphs of the effects of the frequency of input data on the quantitative and qualitative performance of PA.

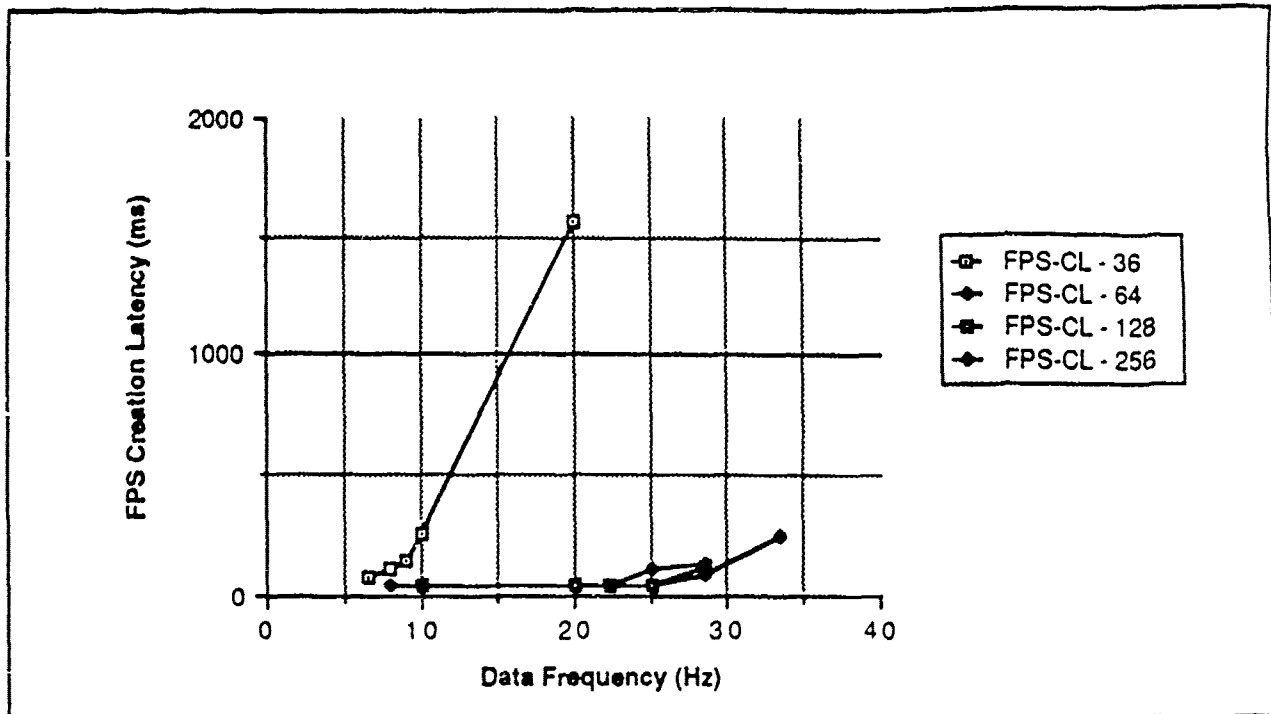


Figure A4.1 The effects of the input data rate on the FPS Creation Latency

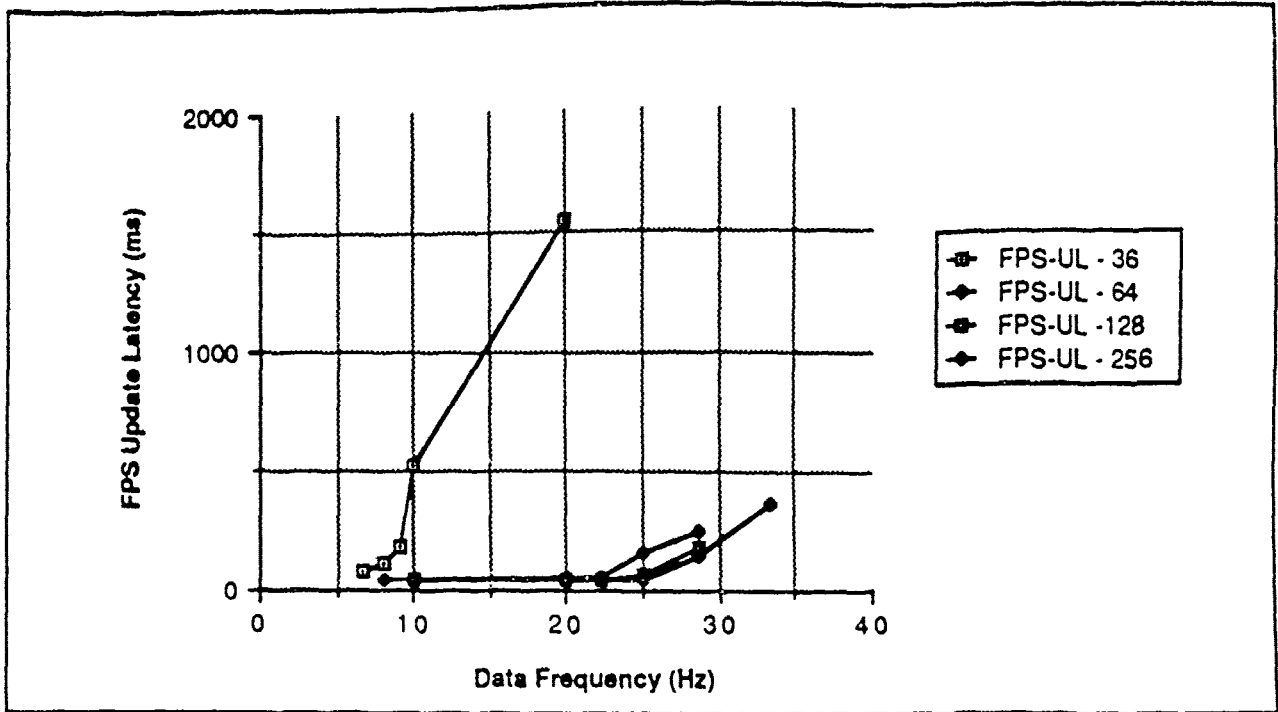


Figure A4.2 The effects of the input data rate on the FPS Update Latency

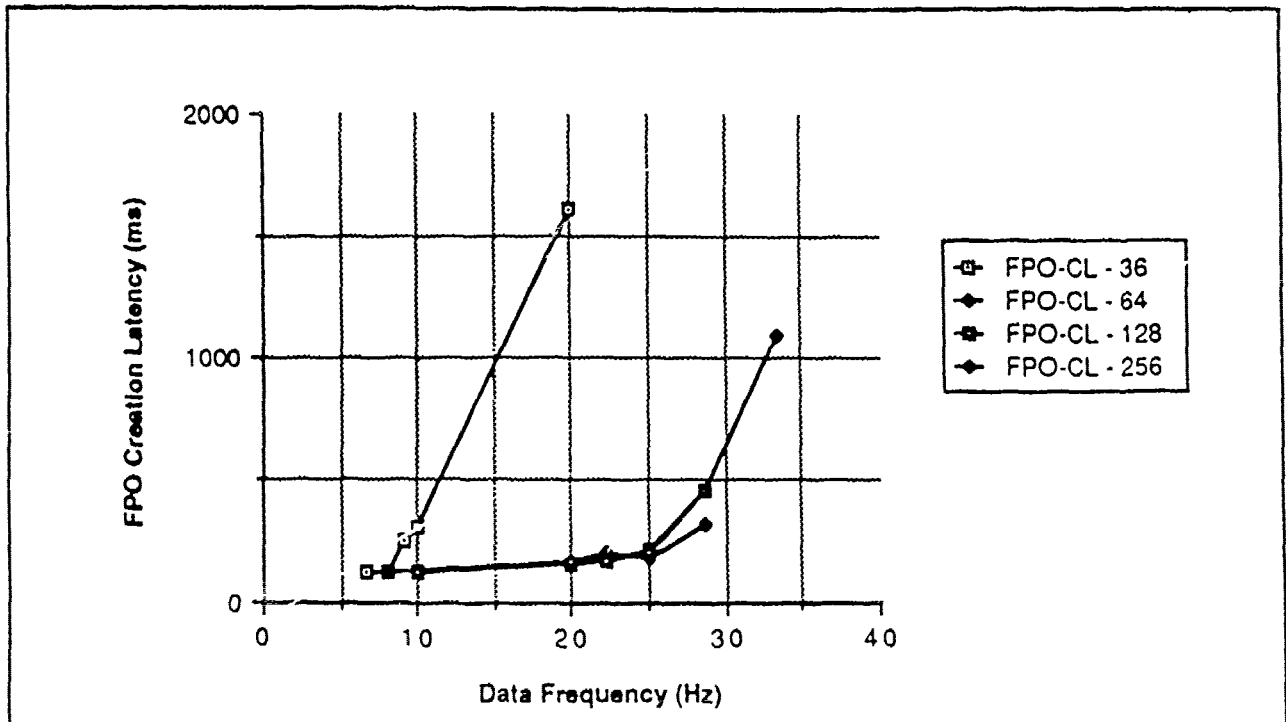


Figure A4.3 The effects of the input data rate on the FPO Creation Latency

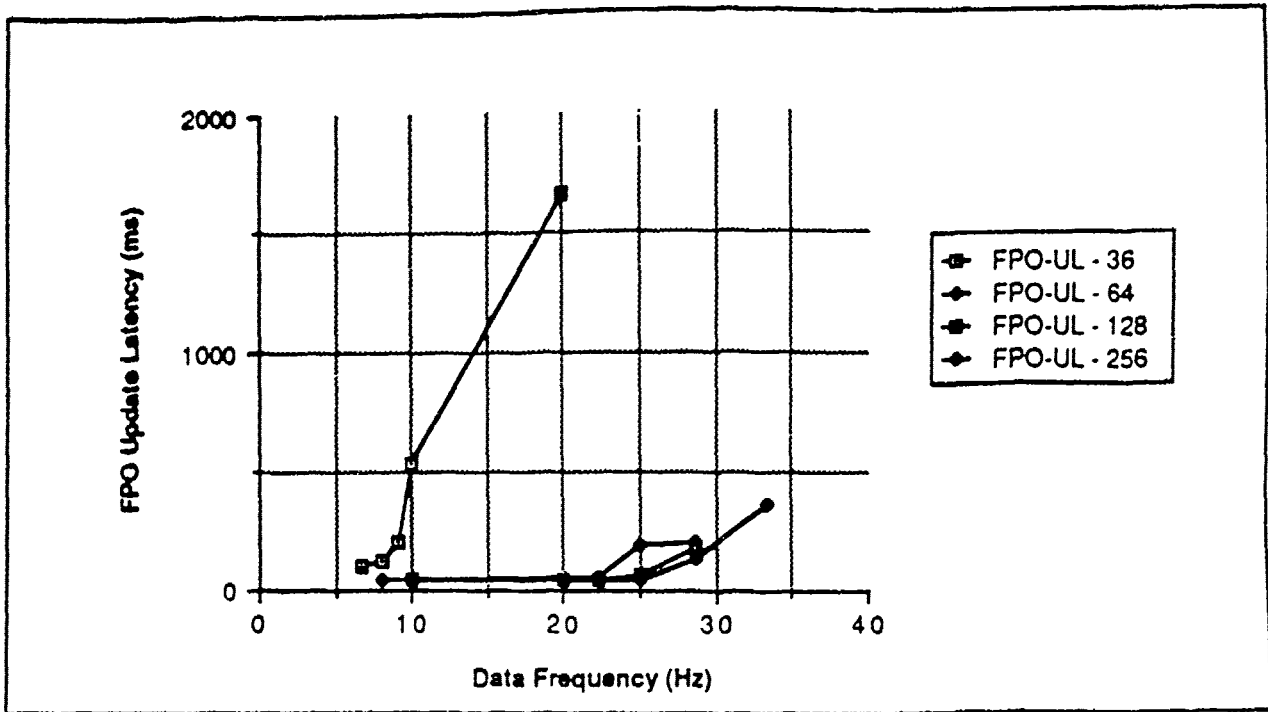


Figure A4.4 The effects of the input data rate on the FPO Update Latency

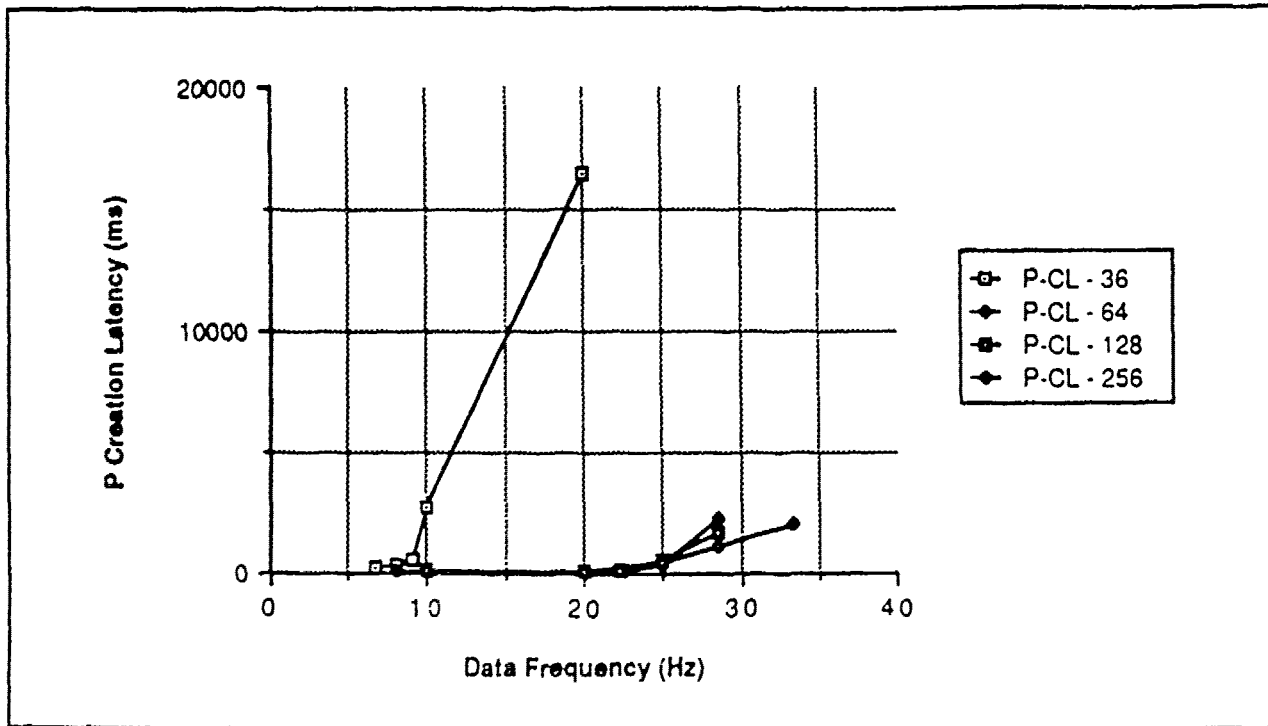


Figure A4.5 The effects of the input data rate on the P Creation Latency

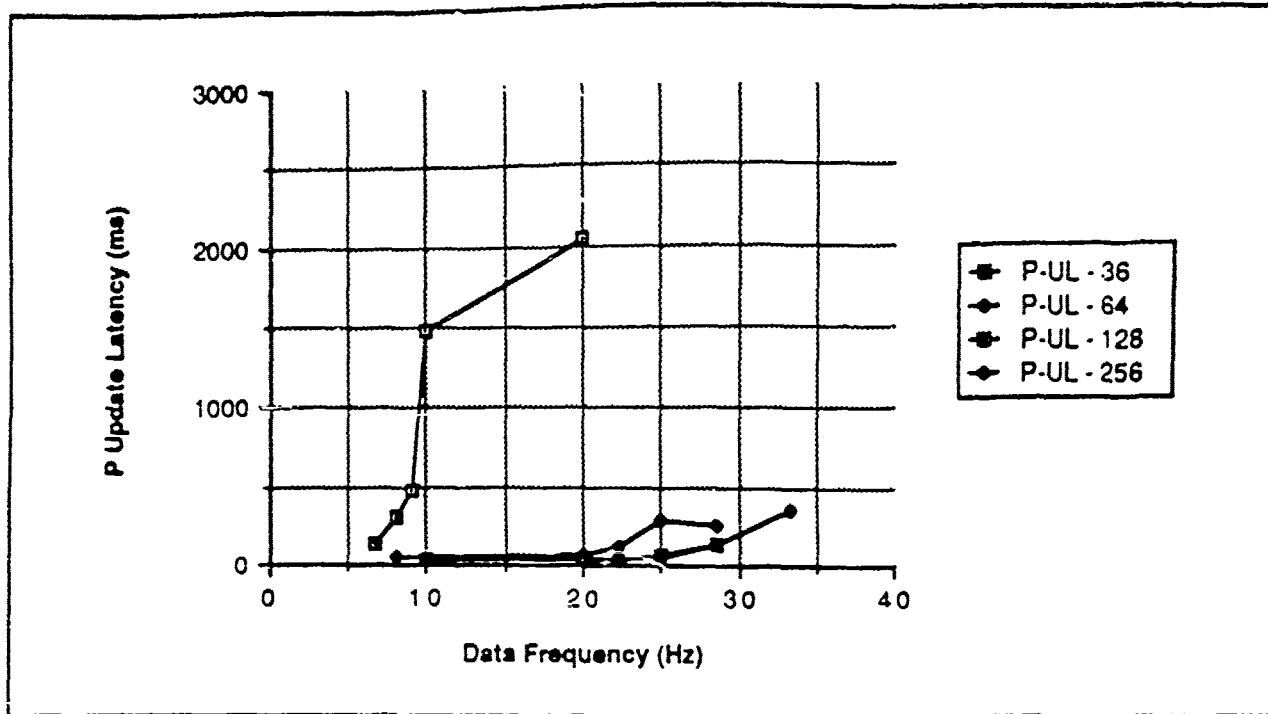


Figure A4.6 The effects of the input data rate on the P Update Latency

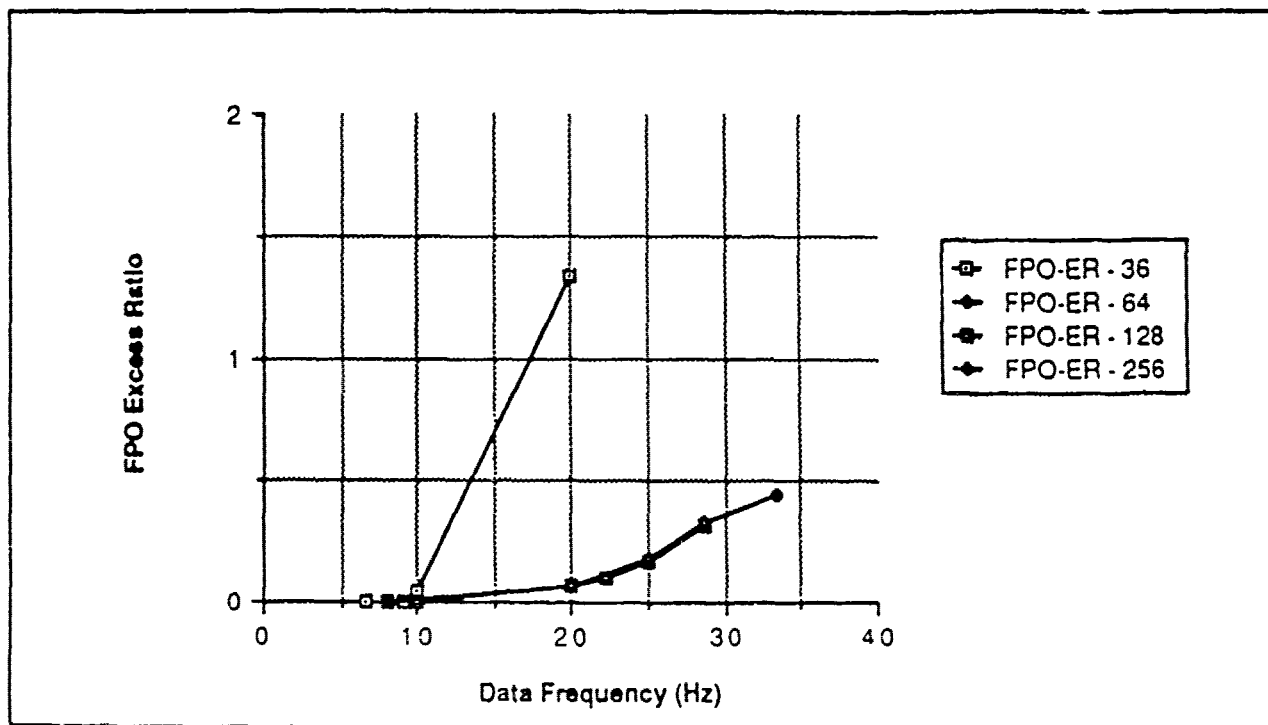


Figure A4.7 The effects of the input data rate on the FPO Excess Ratio

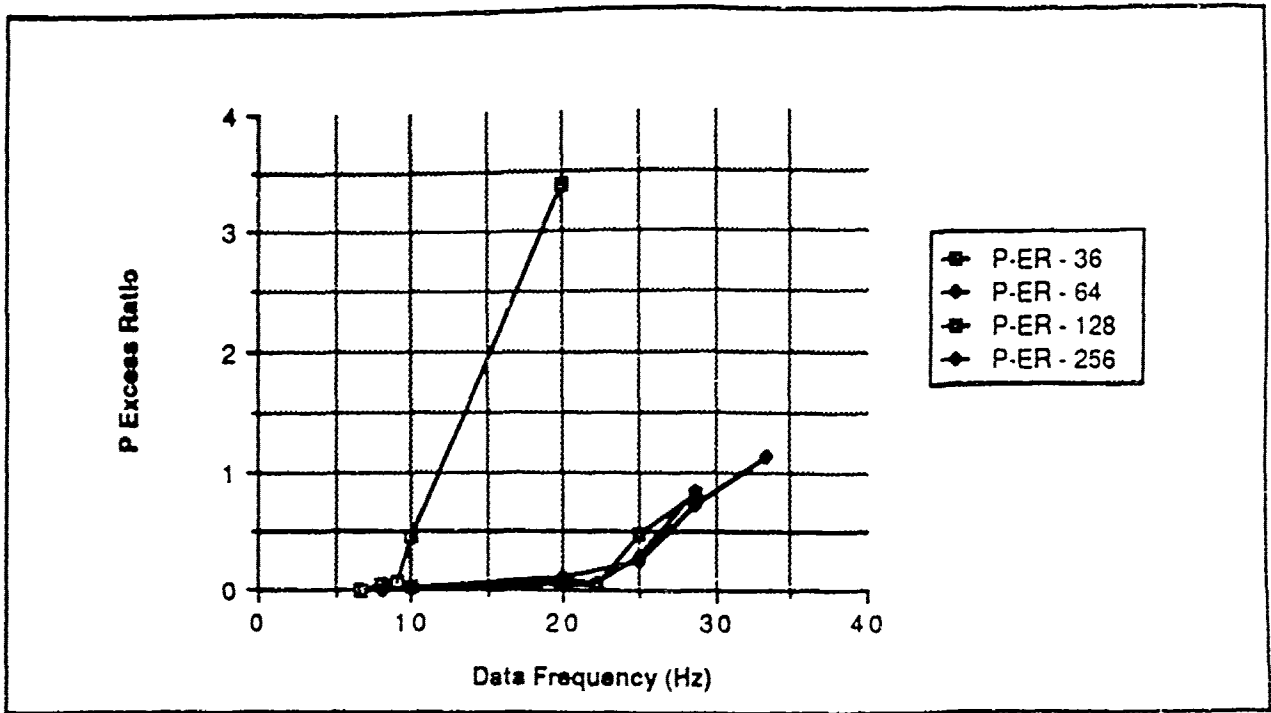


Figure A4.8 The effects of the input data rate on the P Excess Ratio

A5. The Effects of the Width of Input Data

This appendix contains the complete graphs of the effects of the width of input data on the quantitative and qualitative performance of PA.

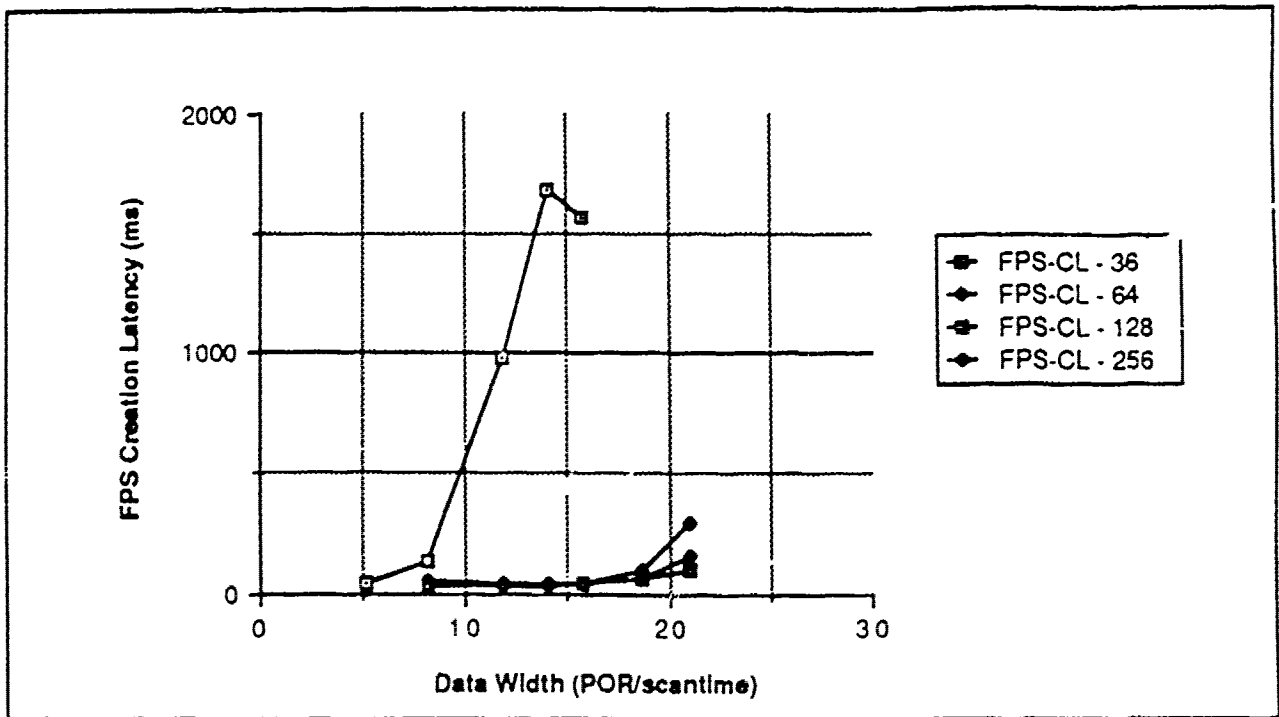


Figure A5.1 The effects of the input data width on the FPS Creation Latency

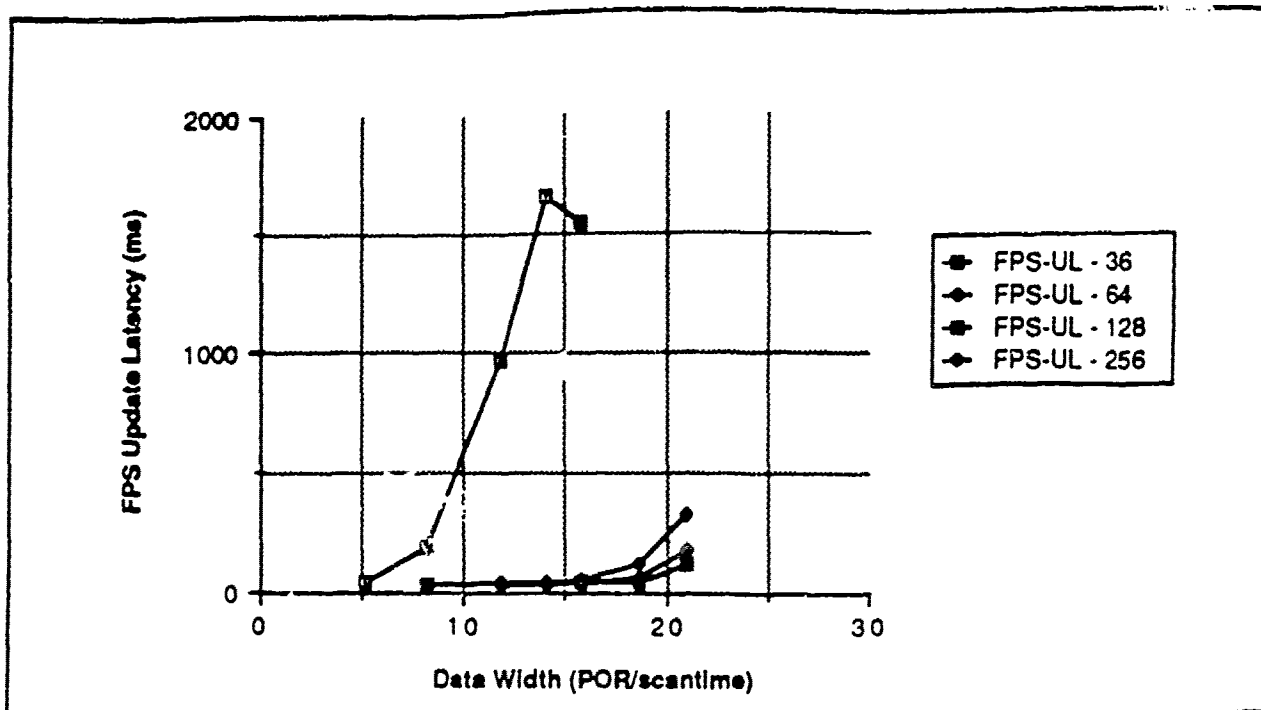


Figure A5.2 The effects of the input data width on the FPS Update Latency

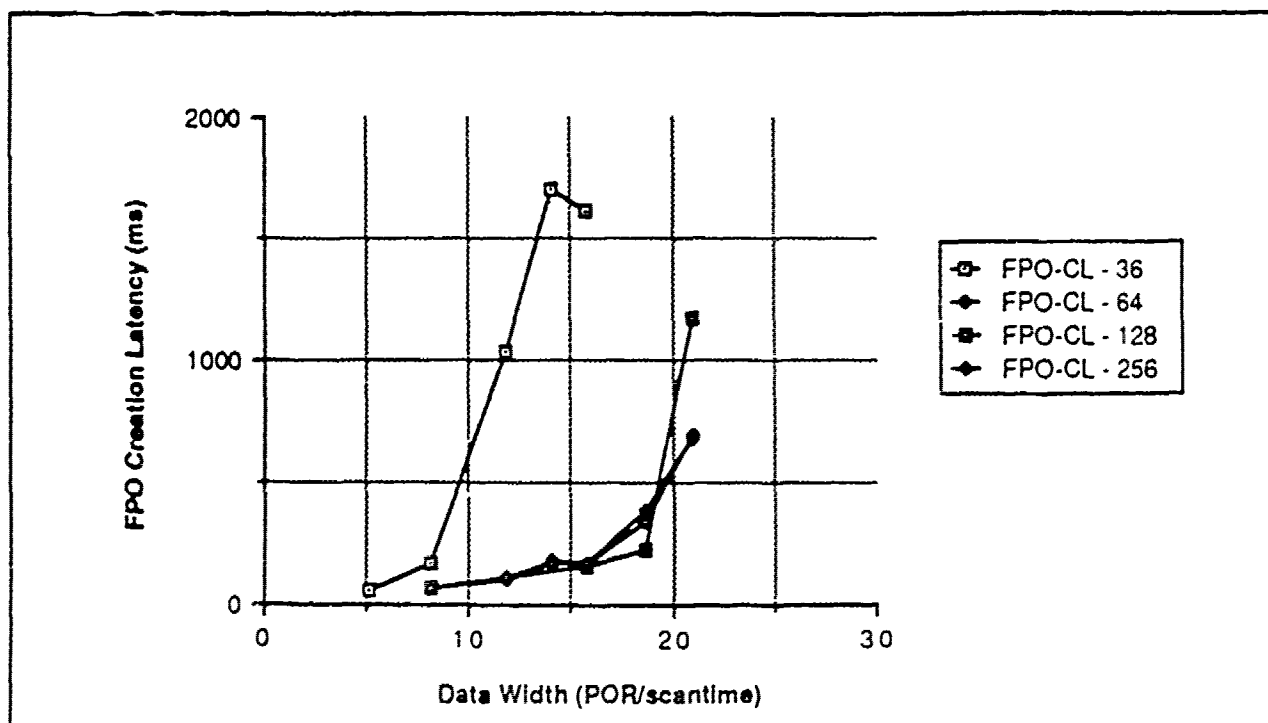


Figure A5.3 The effects of the input data width on the FPO Creation Latency

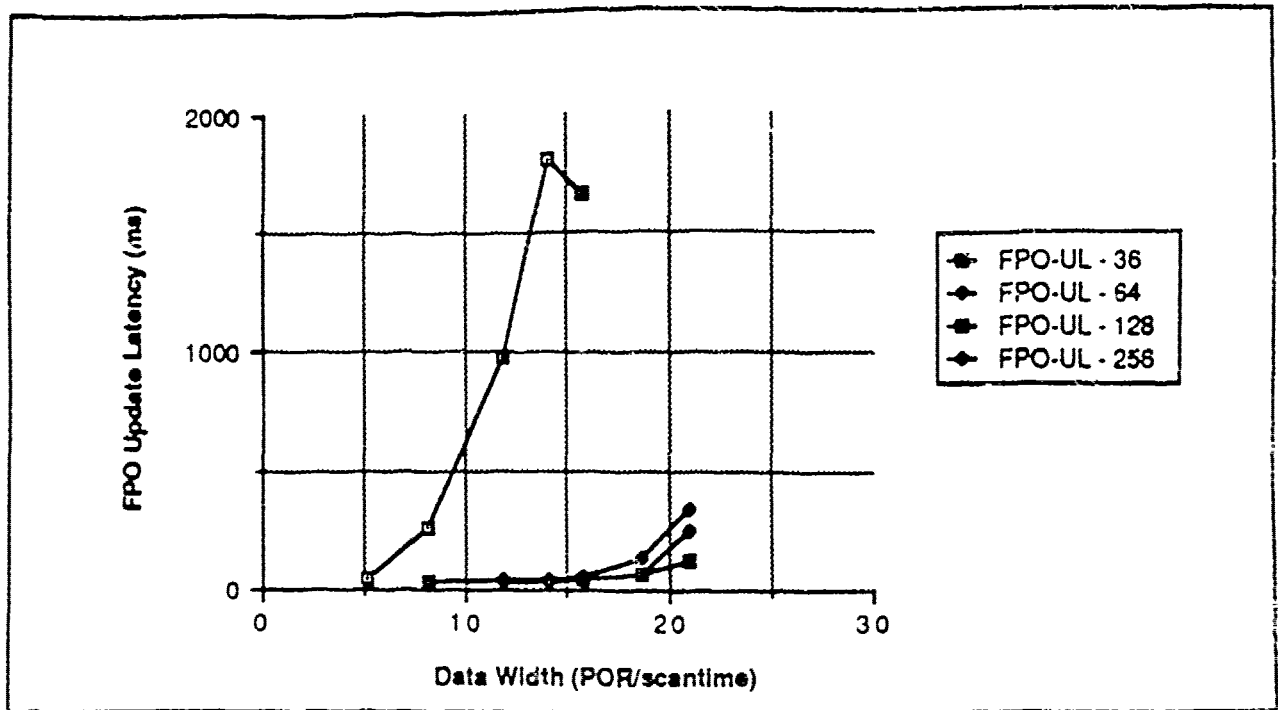


Figure A5.4 The effects of the input data width on the FPO Update Latency

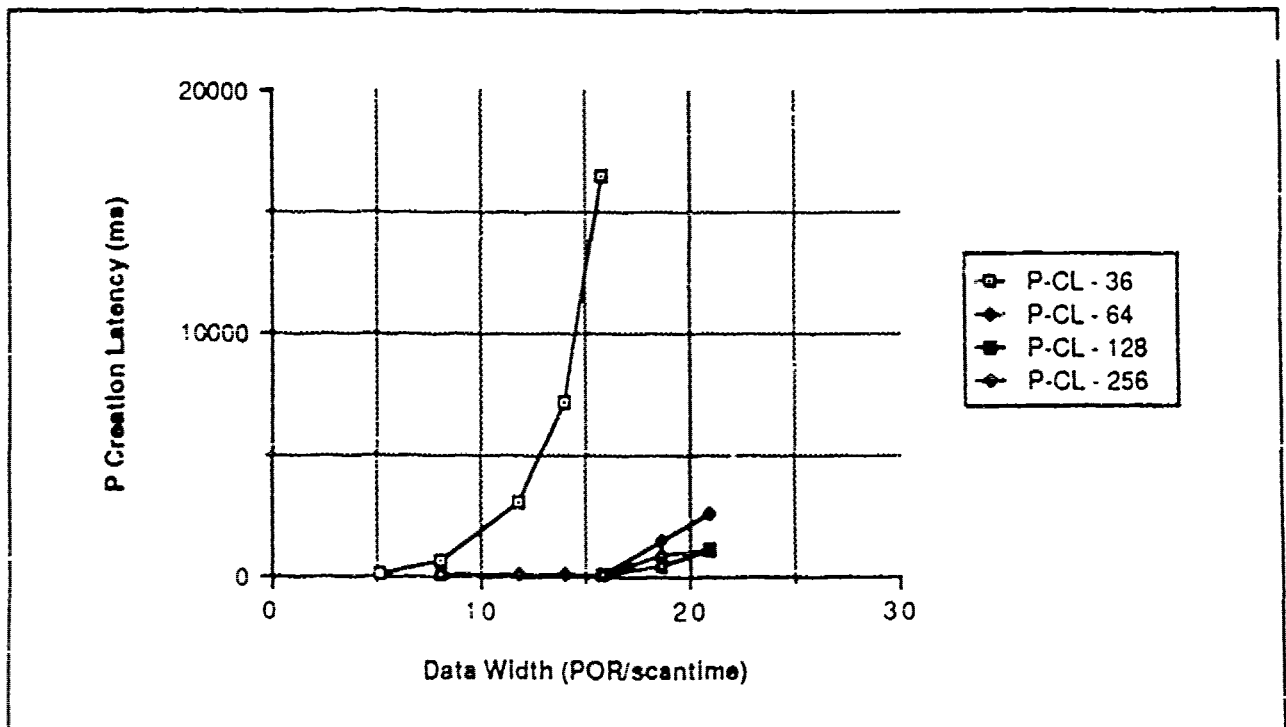


Figure A5.5 The effects of the input data width on the P Creation Latency

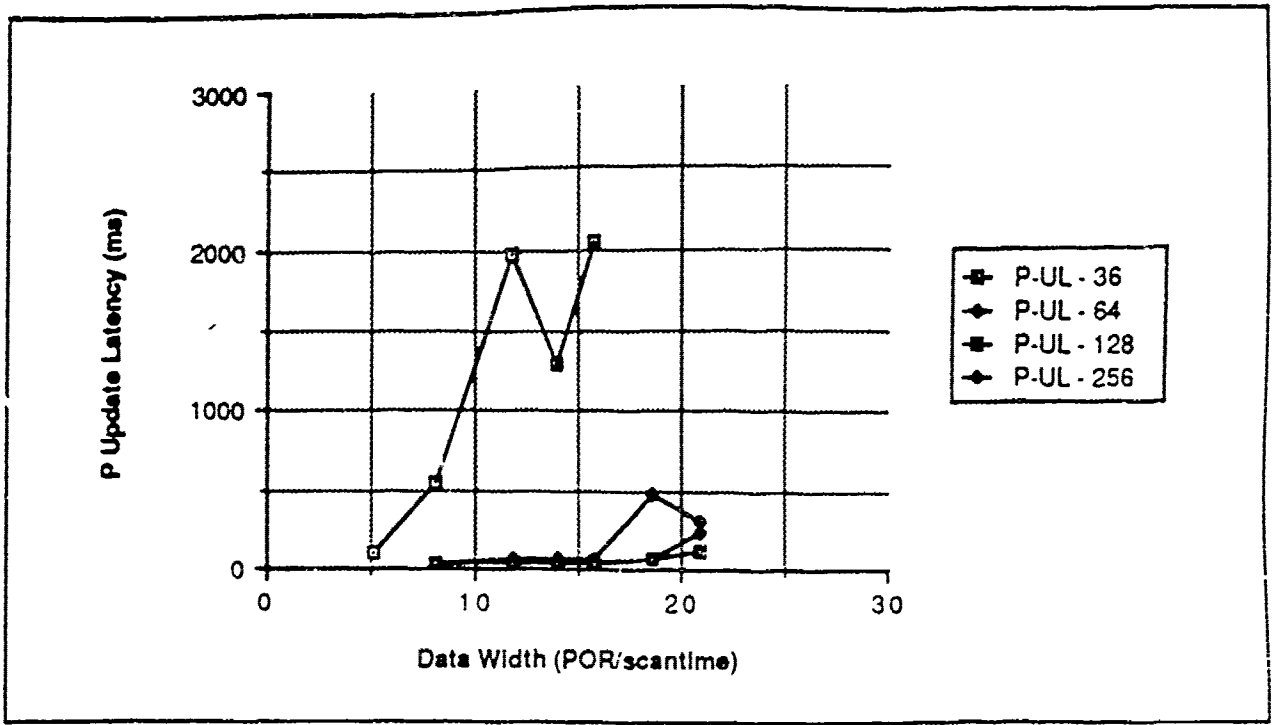


Figure A5.6 The effects of the input data width on the P Update Latency

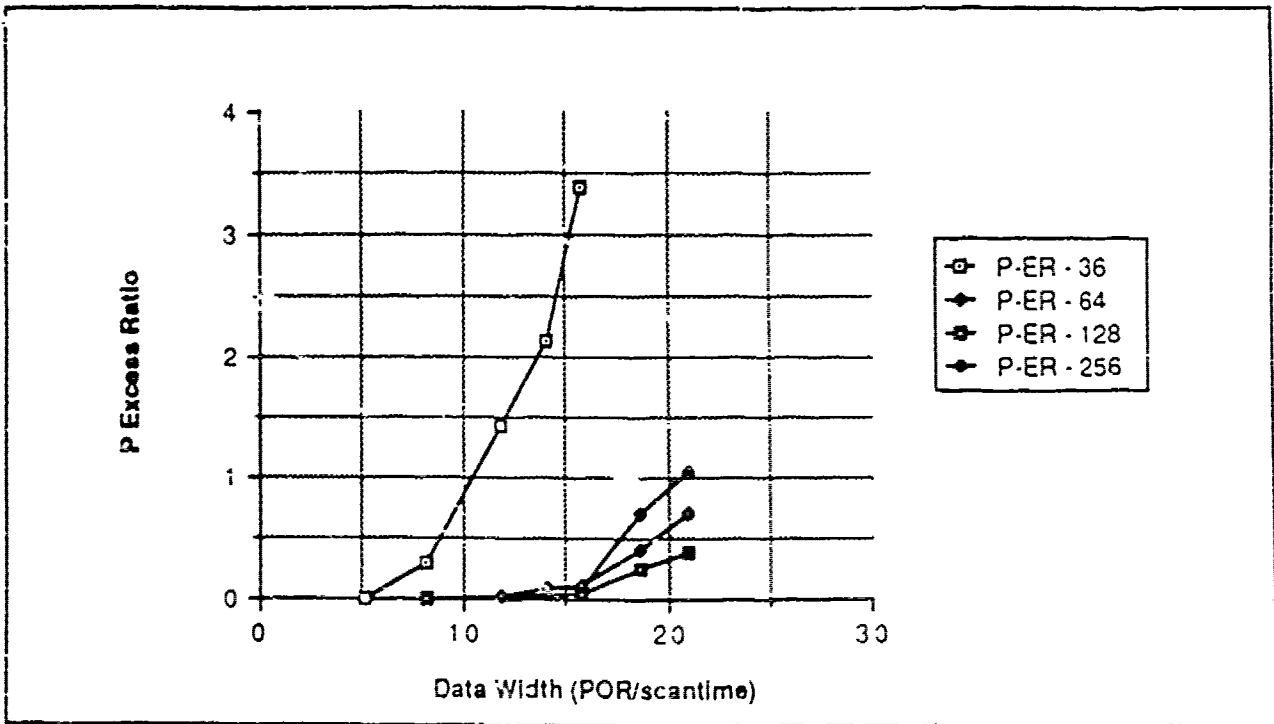


Figure A5.7 The effects of the input data width on the P Excess Ratio

A6. Derivation of a Capacity-Based Speedup Curve

This appendix shows how a capacity-based speedup curve is derived. Figures A6.1 through A6.6 show the capacity achievable, on the grid sizes 36, 64, 128, and 256, for a selected range of latency requirement. Each figure corresponds to an element of qn . Figures A6.7 and A6.8 show the data widths on the four grid sizes that correspond to different FPO and P excess ratios. These data widths are not necessarily sustainable, since the notion of sustainability is tied to the notion of "not growing in time." There is no way, however, to know whether an excess ratio grows in time.

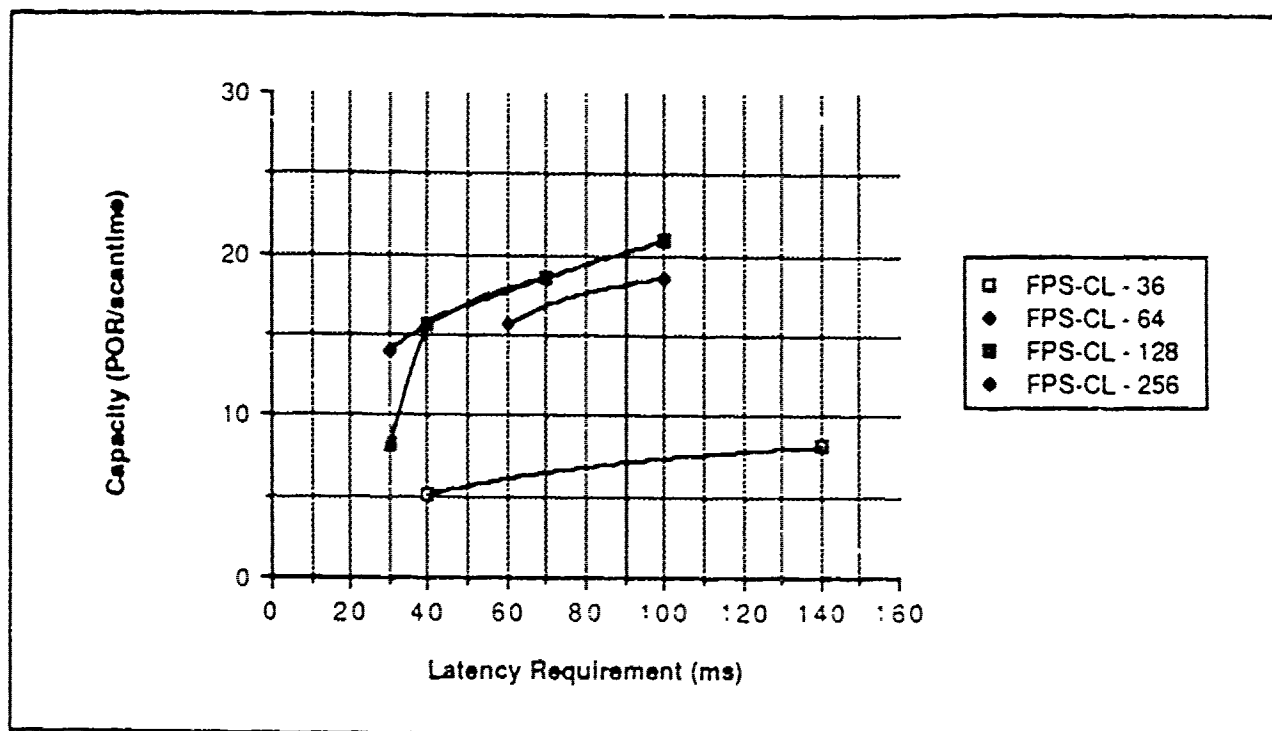


Figure A6.1 Capacities of FPS Creation Latency

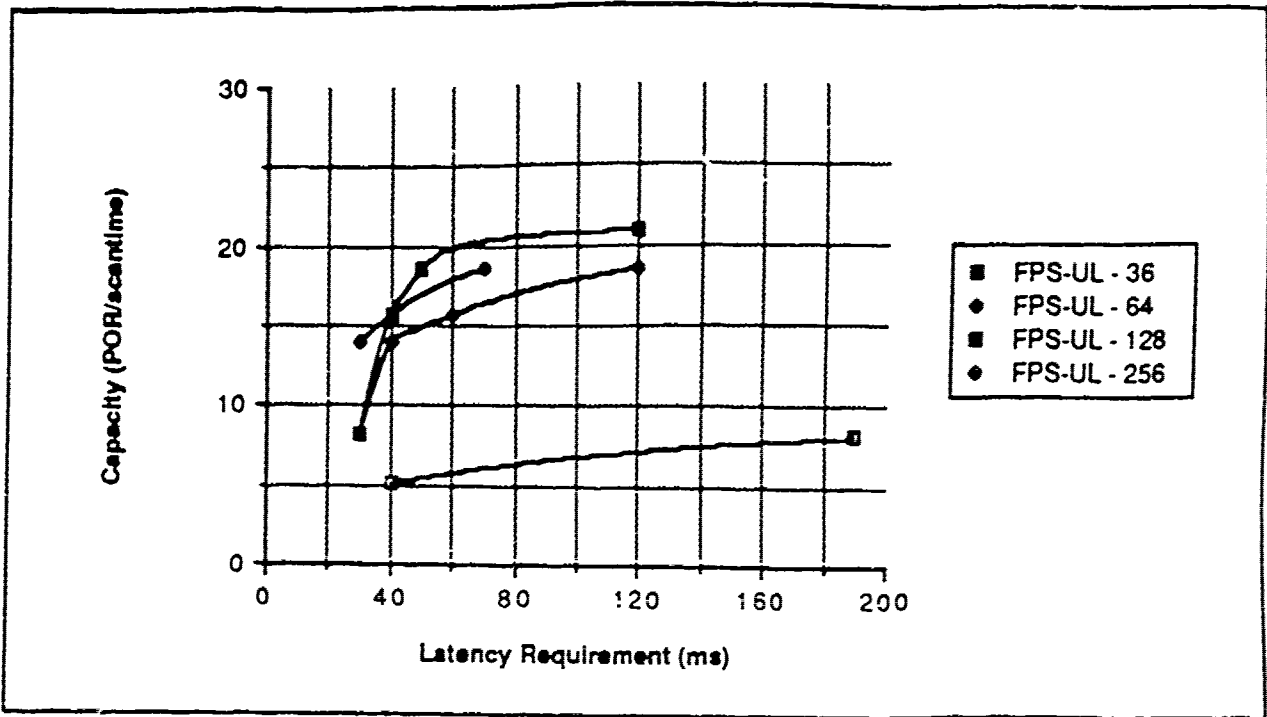


Figure A6.2 Capacities of FPS Update Latency

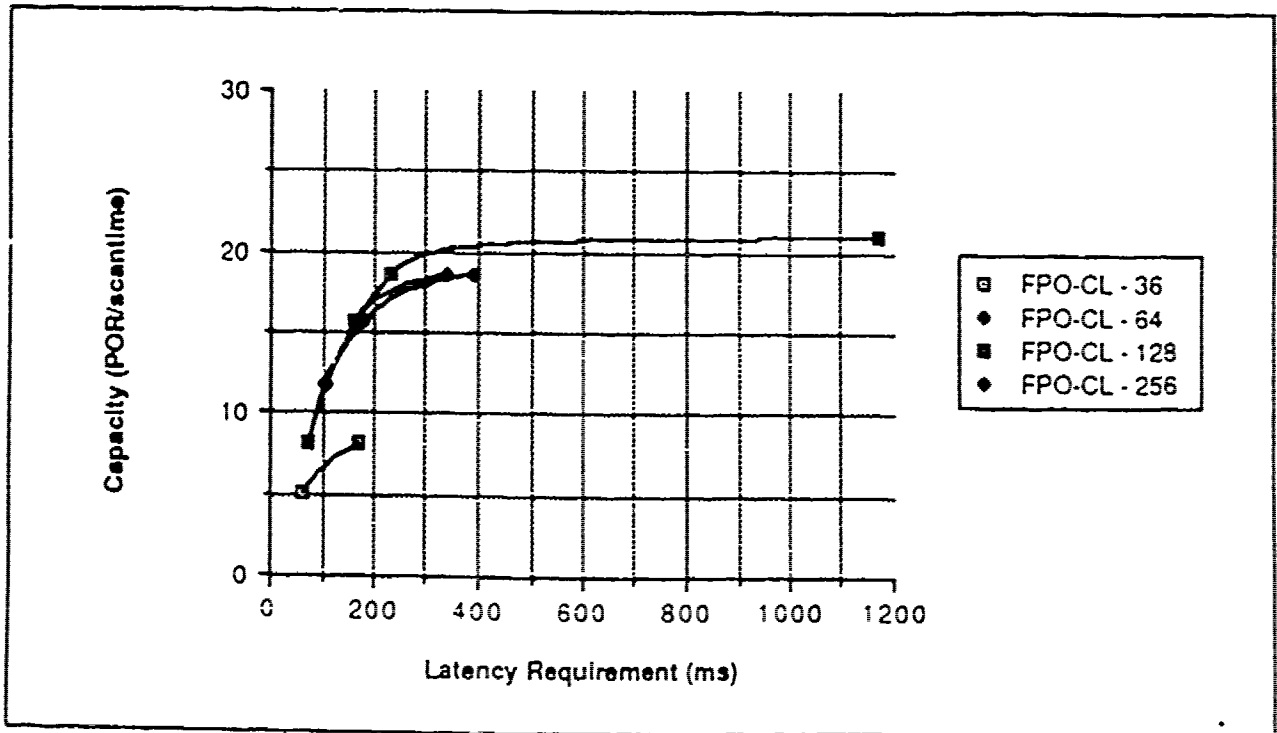


Figure A6.3 Capacities of FPO Creation Latency

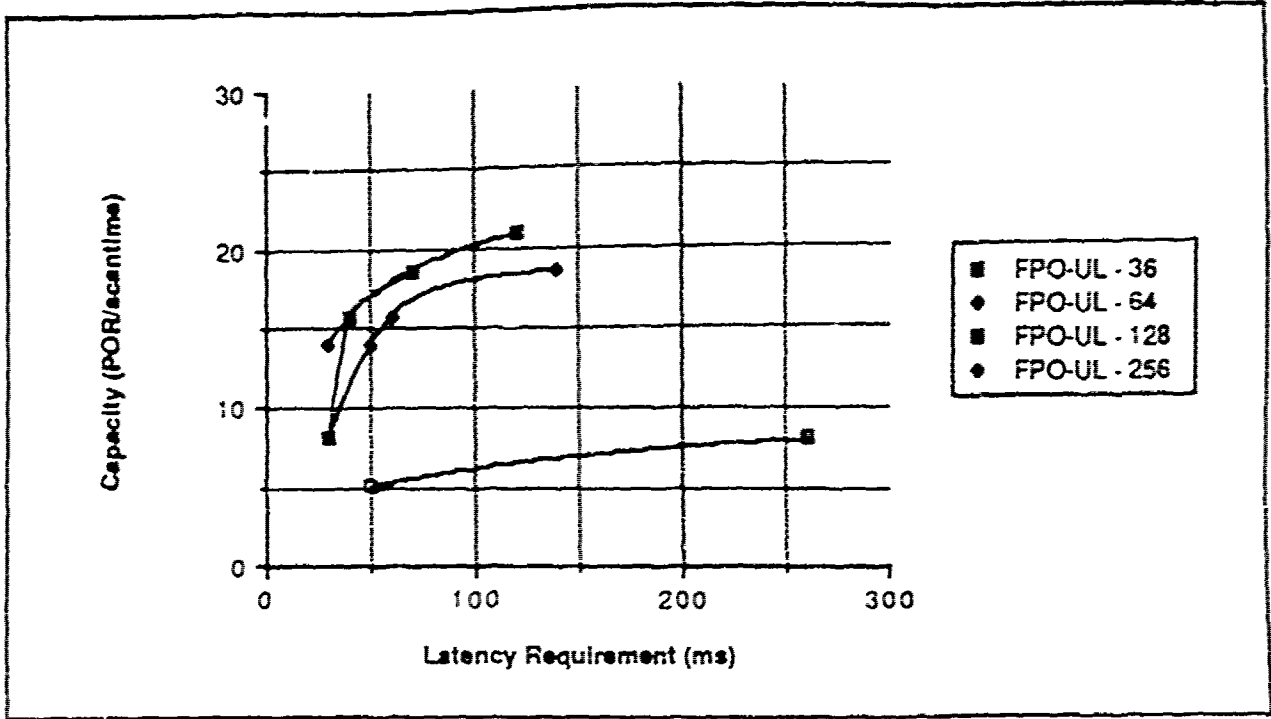


Figure A6.4 Capacities of FPO Update Latency

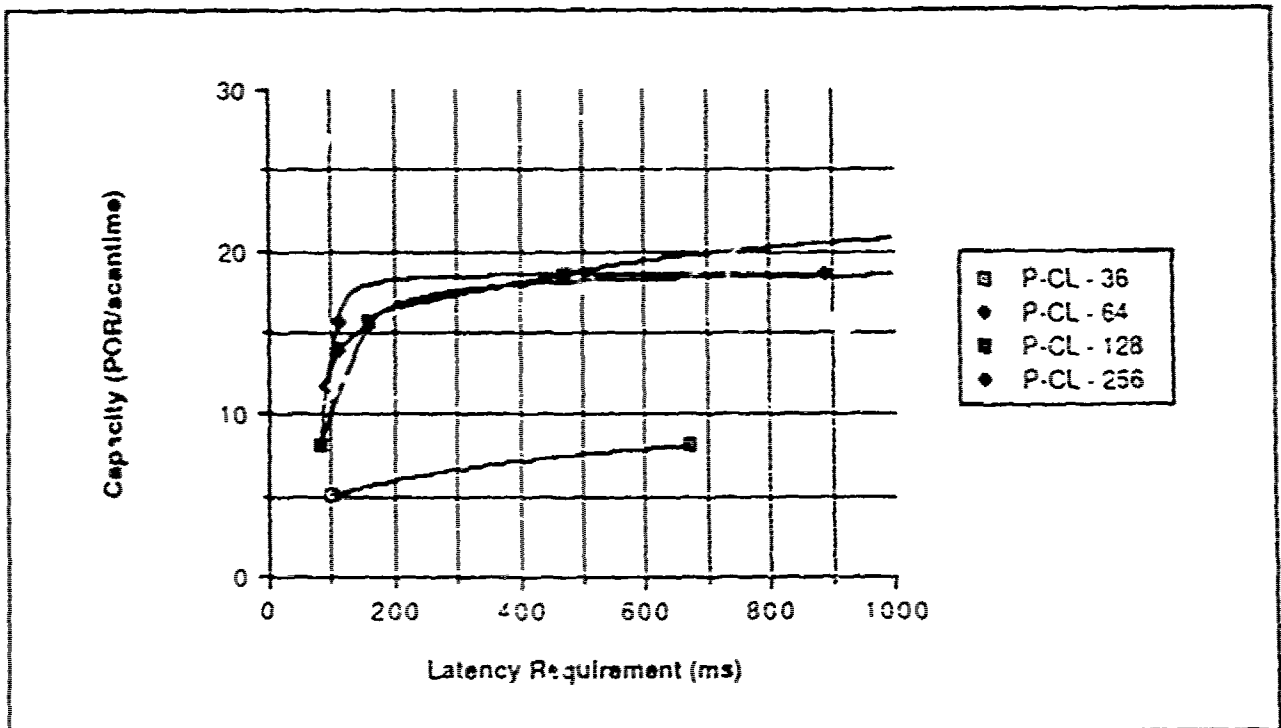


Figure A6.5 Capacities of P Creation Latency

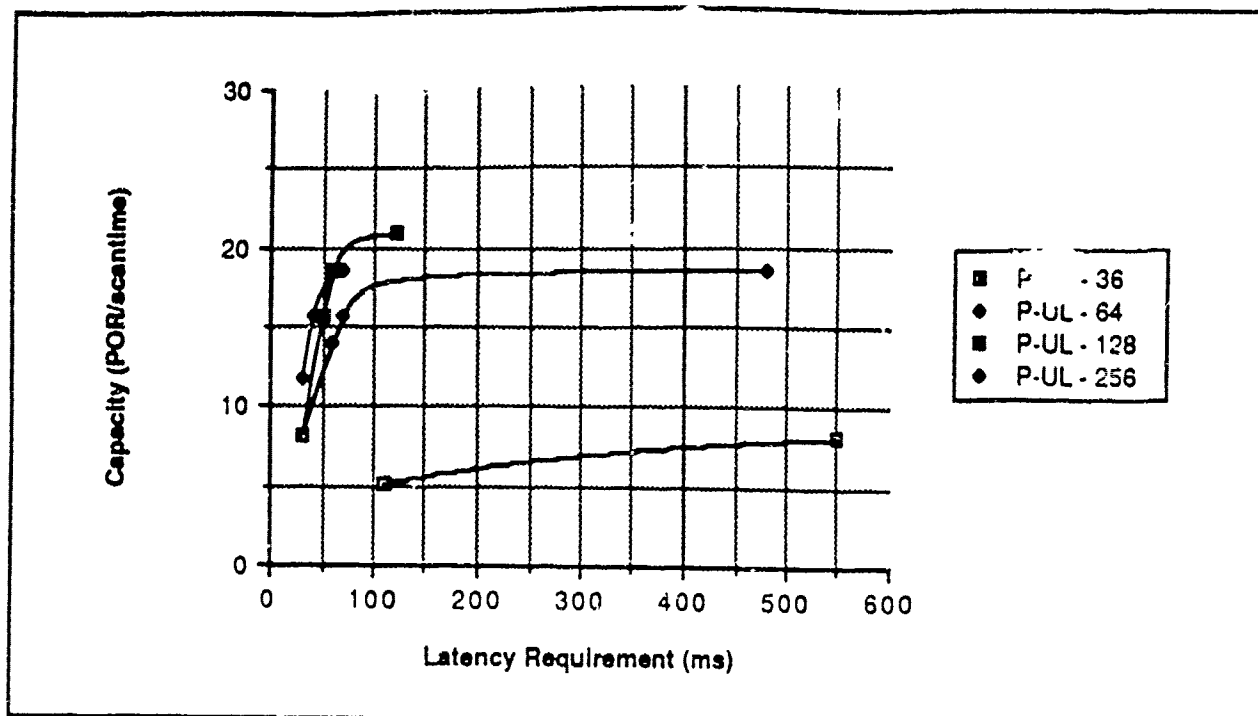


Figure A6.6 Capacities of P Update Latency

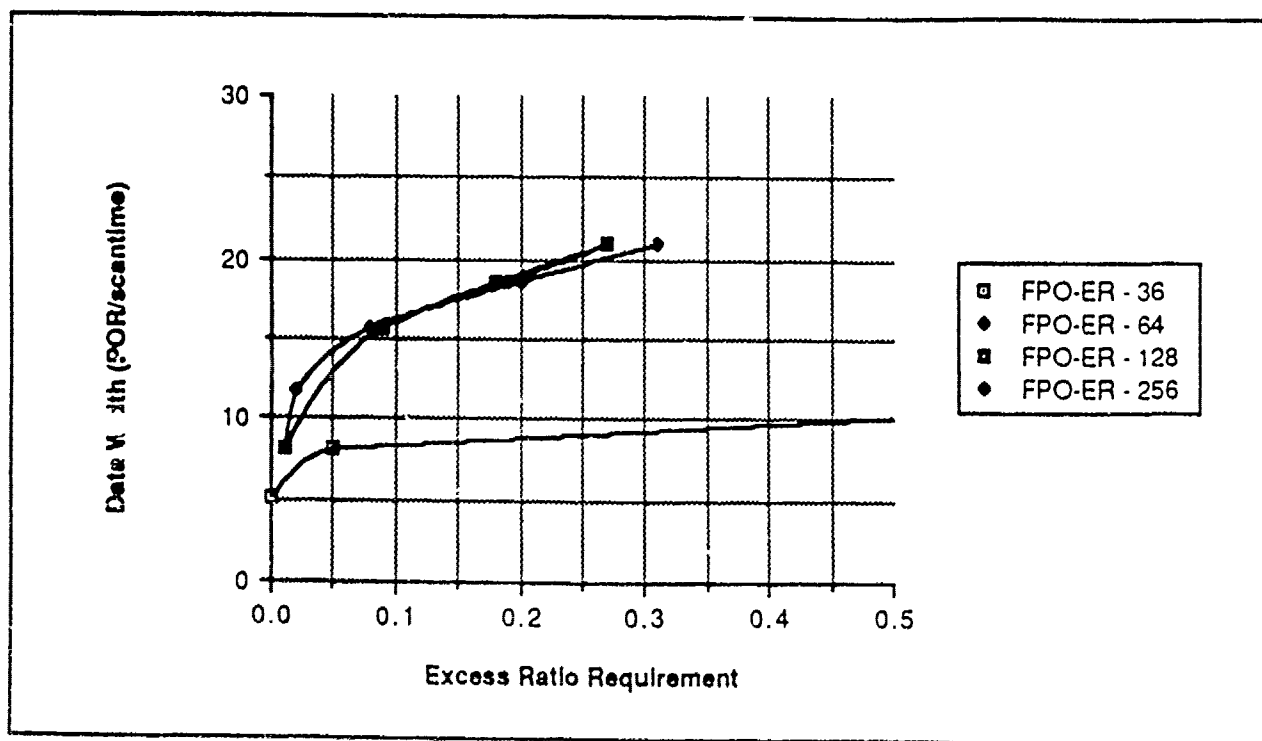


Figure A6.7 Capacities of FPO Excess Ratio

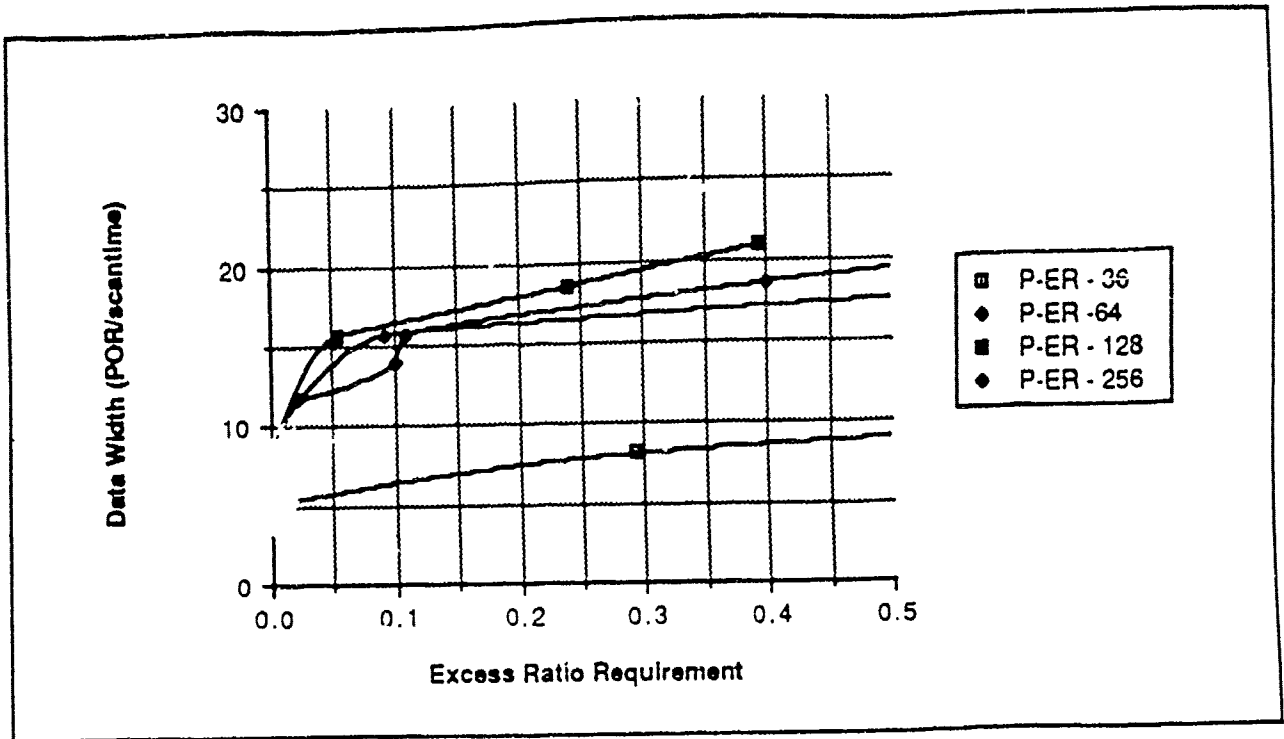


Figure A6.8 Capacities of P Excess Ratio

Figure A6.9 shows the capacities of each element of the following performance requirement:

$$q_n = \begin{bmatrix} \text{FPS-CL} \\ \text{FPS-UL} \\ \text{FPO-CL} \\ \text{FPO-UL} \\ \text{P-CL} \\ \text{P-UL} \end{bmatrix} = \begin{bmatrix} 100 \\ 80 \\ 200 \\ 100 \\ 300 \\ 150 \end{bmatrix} \text{ ms}$$

$$q_l = \begin{bmatrix} \text{FPS-ER} \\ \text{FPO-ER} \\ \text{P-ER} \end{bmatrix} = \begin{bmatrix} 0.00 \\ 0.15 \\ 0.20 \end{bmatrix}$$

The capacities of grid size 36 vary from 5.60 to 8.55; grid size 64, 16.40 to 18.61; grid size 128, 17.35 to 20.93; and grid size 256, 16.90 to 18.61. So the overall capacities for grid sizes 36, 64, 128, and 256 are 5.60, 16.40, 17.35, and 16.90, respectively.

Figure A6.10 shows the capacity-based speedup curve that corresponds to the performance requirement.

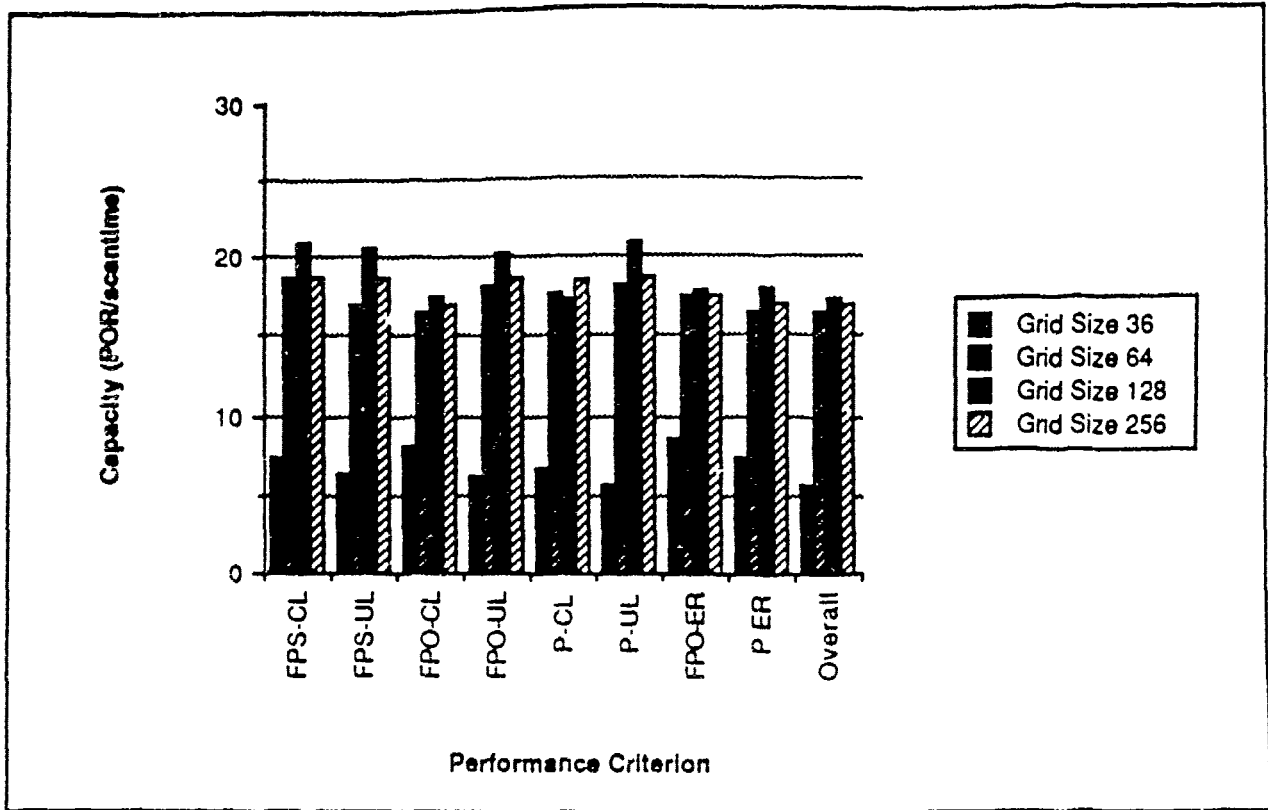


Figure A6.9 Capacities of of each element of the performance requirement

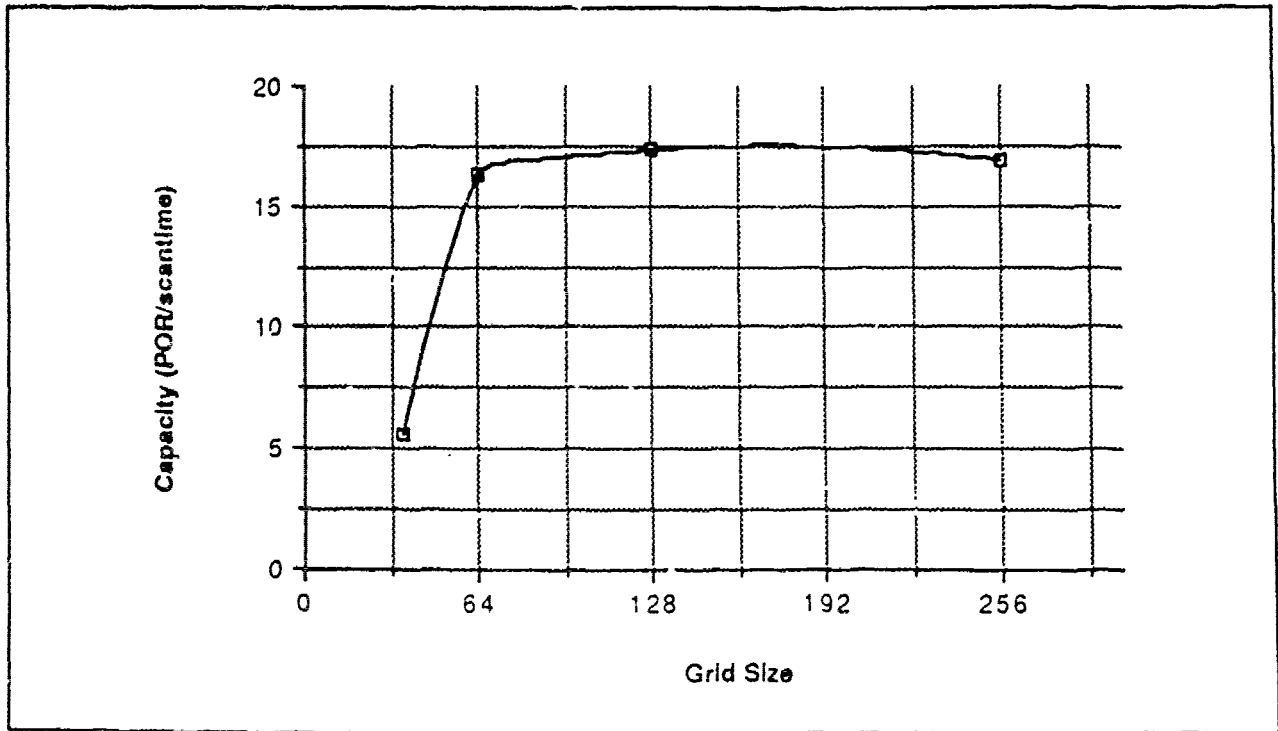


Figure A6 10 A possible capacity-based speedup curve

**Experiments with a Knowledge-Based System
on a Multiprocessor**

by

Russell Nakano[†] and Masafumi Minami

KNOWLEDGE SYSTEMS LABORATORY

Computer Science Department

Stanford University

Stanford, California 94305

[†]WORKSTATION SYSTEMS ENGINEERING

Digital Equipment Corporation

Palo Alto, California 94301

This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and the Workstation Systems Engineering group of Digital Equipment Corporation.

Abstract

This paper documents the results we obtained and the lessons we learned in the design, implementation, and execution of a simulated real-time application on a simulated parallel processor. Specifically, our parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor.

The machine architecture is a distributed-memory multiprocessor. The target machine consists of 10 to 1000 processors, but because of simulator limitations, we ran simulations of machines consisting of 1 to 100 processors. Each processor is a computer with its own local memory, executing an independent instruction stream. There is no global shared memory; all processes communicate by message passing. The target programming environment, called Lamina, encourages a programming style that stresses performance gains through problem decomposition, allowing many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

We focused on a knowledge-based application that simulates real-time understanding of radar tracks, called Airtrac. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. We confirmed the following hypotheses: 1) Performance of our concurrent program improves with additional processors, and thereby attains a significant level of speedup. 2) Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

1. Introduction

This paper focuses on the problems confronting the programmer of a concurrent program that runs on a distributed memory multiprocessor. The primary objective of our experiments is to obtain speedup from parallelism without compromising correctness. Specifically, our parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor. The goal of this paper is to explain why we made certain design choices and how those choices influence our result.

A major theme in our work is the tradeoff between speedup and correctness. We attempt to obtain speedup by decomposing our problem to allow many sub-problems to be solved concurrently. This requires deciding how to partition the data structures and procedures for concurrent execution. We take care in decomposing our problem; to a first approximation, more decomposition allows more concurrency and therefore greater speedup. At the same time, decomposition increases the interactions and dependencies between the sub-problems and makes the task of obtaining a correct solution more difficult.

This paper focuses on the implementation of a knowledge-based expert system in a concurrent object-oriented programming paradigm called Lamina [Delagi 87a]. The target is a distributed-memory machine consisting of 10 to 1000 processors, but because of simulator limitations, our simulations examine 1 to 100 processors. Each processor is a computer with a local memory and an independent instruction stream.¹ There is no global shared memory of any kind.

Airtrac is a knowledge-based application that simulates real-time understanding of radar tracks. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. We encoded and implemented the knowledge from the domain of real-time radar track interpretation for execution on a distributed-memory message-passing multiprocessor system. Our goal was to achieve a significant level of problem-solving speedup by techniques that exploited both the characteristics of our simulated parallel machine, as well as the parallelism available in our problem domain.

The remainder of this paper is organized as follows. Section 2 introduces definitions that we use throughout the paper. Section 3 describes the model of the parallel machine that we simulate, and the model of computation from the viewpoint of the programmer. Section 4 outlines a set of principles that we follow in our programming effort in order to shed light on why we take the approach that we do. Section 5 describes the signal understanding problem that our parallel program addresses. Section 6 describes the design of our experiments, and Section 7 presents the results. Section 8 discusses a number of design issues, and Section 9 summarizes the paper.

¹Each processor is roughly comparable to a 32-bit microprocessor-based system equipped with a multitasking kernel that supports interprocessor communication and restartable processes (as opposed to resumable processes). The hardware system is assumed to support high-bandwidth, low-latency interprocessor communications as described in Byrd et.al. [Byrd 87].

2. Definitions

Using the definitions of Andrews and Schneider [Andrews 83], a *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*.

We define $S_{n,m}$ *speedup* as the ratio $\frac{T_m}{T_n}$, where T_k denotes the time for a given task to be completed on a k -processor multiprocessor. Both T_m and T_n represent the *same* concurrent program running on m -processor and n -processor multiprocessors, respectively. When we compare an n -processor multiprocessor to a 1-processor multiprocessor, we obtain a measure for $S_{n/1}$ speedup which should be distinguished from *true speedup*, defined as the ratio $\frac{T^*}{T_n}$, where T^* denotes the time for a given task to be completed by the *best* implementation possible on a uniprocessor.² In particular, T^* excludes overhead tasks (e.g. message-passing, synchronization, etc.) that T_1 counts.

We define *correctness* to be the degree to which a concurrent program executing on a k -processor multiprocessor obtains the same solution as a conventional uniprocessor-based sequential program embodying the same knowledge as contained in the concurrent program. We call the latter solution a *reference solution*. We use a serial version of our system to generate a reference solution, to evaluate the correctness of the parallel implementation.³

MacLennan [MacLennan 82] distinguishes between value-oriented and object-oriented programming styles. A *value* has the following properties:

- A value is read-only.
- A value is atemporal (i.e. timeless and unchanging).
- A value exhibits referential transparency, that is, there is never the danger of one expression altering something used by another expression.

These properties make values extremely attractive for concurrent programs. Values are immutable and may be read by many processes, either directly or through "copies" of values that are equal; this facilitates the achievement of correctness as well as concurrency. A well-known example of value-oriented programming is functional programming [Henderson 80]. Other examples of value-oriented programming in the realm of parallel computing include systolic programs [Kung 82] and scalar data flow programs [Arvind 83, Dennis 85], where the data flowing from processor to processor may be viewed as values that represent abstractions of various intermediate problem-solving stages.

²A 1-processor multiprocessor executes the same parallel program that runs on a n -processor multiprocessor. In particular, it creates processes that communicate by sending messages, as opposed to sharing a common memory.

³Unfortunately, our reference program is not a valid producer of T^* estimates, and we cannot use it to obtain true speedup estimates. Project resource limitations prevented us from developing an optimized program to serve as a *best* serial implementation.

In contrast, MacLennan defines *objects* in computer programming to have one or more of the following properties:

- An object may be created and destroyed.
- An object has state.
- An object may be changed.
- An object may be shared.

Computer programs often simulate some physical or logical situation, where objects represent the entities in the simulated domain. For example, a record in an employee database corresponds to an employee. An entry in a symbol table corresponds to a variable in the source text of a program. Variables in most high-level programming languages represent objects. Objects provide an abstraction of the state of physical or logical entities, and reflect changes that those entities undergo during the simulation. These properties make objects particularly useful and attractive to a programmer.

Objects in a concurrent program introduce complications. In particular, many parallel processes may attempt to create, destroy, change, or share an object, thereby causing potential problems. For instance, one process may read an object, perform a computation, and change the object. Another process may concurrently perform a similar sequence of actions on the same object, leading to the possibility that operations may interleave, and render the state of the object inconsistent. Many solutions have been proposed, including semaphores, conditional critical regions, and monitors; all of these techniques strive to achieve correctness and involve some loss of concurrency.

Our programming paradigm, Lamina, supports a variation of *monitors*, defined as a collection of permanent variables (we use the term *instance variables*), used to store a resource's state, and some procedures, which implement a set of allowed operations on the resource [Andrews 83]. Although monitors provide mutual exclusion, concurrency considerations force us to abandon mutual exclusion as the sole technique to obtain correctness.

We classify techniques for obtaining speedup in problem-solving into two categories: replication and pipelining. *Replication* is defined as the decomposition of a problem or sub-problem into many independent or partially independent sub-problems that may be concurrently processed. *Pipelining* is defined as the decomposition of a problem or sub-problem into a sequence of operations that may be performed by successive stages of a processing pipeline. The output of one stage is the input to the next stage.

3. Computational model

3.1. Machine model

Our machine architecture, referred to as CARE [Delagi 87a], may be modeled as an asynchronous message-passing distributed system with reliable datagram service [Tanenbaum 81]. After sending a message, a process may continue to execute (i.e. message passing is asynchronous). Arrival order of messages may differ from the order in which they were sent (i.e. datagram as opposed to virtual circuit). The network guarantees that no message is ever lost (i.e. reliable), although it does not guarantee when a message

will arrive. Each processor within the distributed system is a computer that supports interprocessor communication and restartable processes. Each processor operates on its own instruction stream, asynchronously with respect to other processors.

In synchronous message passing, maintaining consistent state between communicating processes is simplified because the sender blocks until the message is received, giving implicit synchronization at the send and receive points. For example, the receiver may correctly make inferences about the sender's program state from the contents of the message it has received, without the possibility that the sender program continued to execute, possibly negating a condition that held at the time the original message was sent.

In asynchronous message passing, the sender continues to execute after sending a message. This has the advantage of introducing more concurrency, which holds the promise of additional speedup. Unfortunately, in its pure form, asynchronous message passing allows the sender to get arbitrarily far ahead of the receiver. This means that the contents of the message reflects the state of the sender at the time the message was sent, which may not necessarily be true at the time the message is received. This consideration makes the maintenance of consistent state across processes difficult, and is discussed more fully in Section 4.

3.2. Programmer model

Our programming paradigm, Lamina, provides language constructs that allows us to exploit the distributed memory machine architecture described earlier [Delagi 87b]. In particular, we focused our programming efforts on the concurrent object-oriented programming model that Lamina provides. As in other object-oriented programming systems, objects encapsulate state information as instance variables. Instance variables may be accessed and manipulated only through methods. Methods are invoked by message-passing.

However, despite the apparent similarity with conventional object-oriented systems, programming within Lamina has fundamental differences:

- Concurrent processes may execute during both object creation and message sending.
- The time required to create an object is visible to the programmer.
- The time required to send a message is visible to the programmer.
- Messages may be received in a different order from which they were sent.

These differences reflect the strong emphasis Lamina places on concurrency. While all object-oriented systems encounter delays in object creation and message sending, these delays are significant within the Lamina paradigm because of the other activities that may proceed concurrently *during* these periods. Subtle and not-so-subtle problems become apparent when concurrent processes communicate, whether to send a message or to create a new object. For instance, a process might detect that a particular condition holds, and respond by sending a message to another process. But because processes continue to execute during message sending, the condition may no longer hold when the message is received. This example illustrates a situation where the recipient of the message cannot correctly assume that because the sender responds to a particular condition by sending a message, that the condition still holds when the message is received.

Regarding message ordering, partly as a result of our experimentation, versions of Lamina subsequent to the one we used provide the ability for the programmer to specify that messages be handled by the receiver in the same order that they were sent [Delagi 87c]. Use of this feature imposes a performance penalty, which places a responsibility on the programmer to determine that message ordering is truly warranted. In the Airtrac application, we believed that ordering was necessary and imposed it through application level routines that examined message sequence numbers (time tags) and queued messages for which all predecessors had not already been handled.

In Lamina, an object is a process. Following the definition of a process provided earlier, we make no commitment to whether a process has a unique virtual address space associated with it. Each object has a top-level dispatch process that accepts incoming messages and invokes the appropriate message handler; otherwise, if there is no available message, the process blocks. Sending a message to an object corresponds to asynchronous message-passing at the machine level. A method executes atomically. Since each object has a single process, and only that process has access to the internal state (instance variables), mutual exclusion is assured. An object and its methods effectively constitute a non-nested monitor.

Our problem-solving approach has evolved from the blackboard model, where nodes on the blackboard form the basic data objects, and knowledge sources consisting of rules are applied to transform nodes (i.e. objects) and create new nodes [Nii 86a, Nii 86b]. Brown et. al. used concepts from the blackboard model to implement a signal-interpretation application on the CARE multiprocessor simulator [Brown 86]. Lamina evolved from the experiences from that effort. In addition, lessons learned in that earlier effort have been incorporated into our work, including the use of replication and pipelining to gain performance, and improving efficiency and correctness by enforcing a degree of consistency control over many agents computing concurrently.

4. Design principles

Lamina represents a programming philosophy that relies on the concepts of replication and pipelining to achieve speedup on parallel hardware. The key to successful application of these principles relies on finding an appropriate problem decomposition that exploits concurrent execution with minimal dependency between replicated or pipelined processing elements.

The price of concurrency and speedup is the cost of maintaining consistency among objects. When writing a sequential program, a programmer automatically gains mutual exclusion between read/write operations on data structures. This follows directly from the fact that a sequential program has only a single process; a single process has sole control over reads and writes to a variable, for instance. This convenience vanishes when the programmer writes a concurrent program. Since a concurrent program has many concurrently executing processes, coordinating the activities of the processes becomes a significant task.

In this section, we develop the concept of a dependence graph program to provide a framework in which tradeoffs between alternate problem decompositions may be examined. Choosing a decomposition that admits high concurrency gives speedup, but it may do so with the expense of higher effort in maintaining consistency. We introduce dependence graph programs to make the tradeoffs more explicit.

4.1. Speedup

Researchers have debated how much speedup is obtainable on parallel hardware, on both theoretical and empirical grounds; Kruskal has surveyed this area [Kruskal 85]. We take the empirical approach because our goal is to test ideas about parallel problem solving using multiprocessor architectures. Our thinking is guided, however, by a number of principles describing how to decompose problems to obtain speedup.

4.1.1. Pipelining

Consider a concurrent program consisting of three cooperating processes: Reader, Executor, and Printer. The Reader process obtains a line consisting of characters from an input source, sends it to the Executor process, and then repeats this loop. The Executor performs a similar function, receiving a line from the Reader, processing it in some way, and sending it to the Printer. The Printer receives lines from the Executor, and prints out the line. These processes cooperate to form a pipeline; see Figure 1. By using asynchronous message passing, we obtain concurrent operation of the processes; for instance, the Printer may be working on one line, while the Executor is working on another. This means that by assigning each process to a different processor, we can obtain speedup, despite the fact that each line must be inputted, processed, and output sequentially. By overlapping the operations we can achieve a higher throughput than is possible with a single process performing all three tasks.

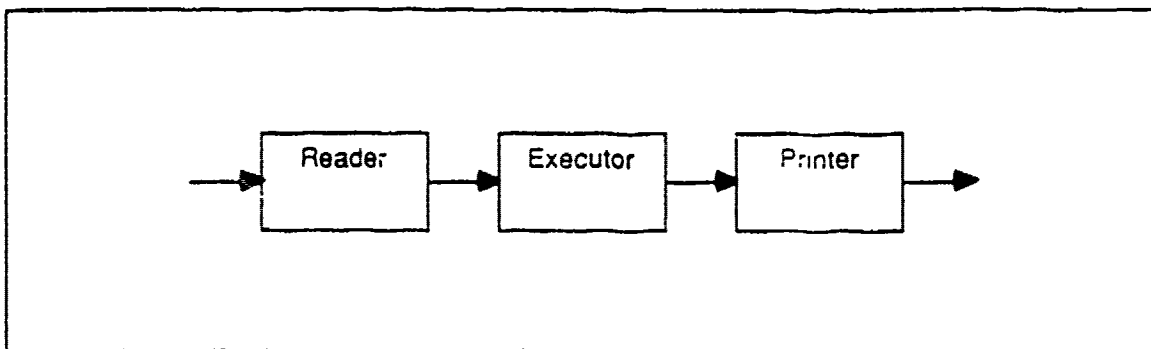


Figure 1. Decomposing a problem to obtain pipeline speedup.

By decomposing a problem in sequential stages, we can obtain speedup from pipelining.

4.1.2. Replication

Consider a variation of Reader-Executor-Printer problem. Suppose that we are able to achieve some overlap in the operations, but we discover that the Executor stage consistently takes longer than the other stages. This causes the Printer to be continually starved for data, while the Reader completes its task quickly and spends most of its time idle. We can improve the overall throughput by replicating the function of the Executor stage by creating many Executors. See Figure 2. By increasing the number of processes performing a given function, we do not reduce the time it takes a single Executor to perform its function, but we allow many lines to be processed concurrently, improving the utilization of the Reader and Printer processes, and boosting overall throughput. This principle of replicating a stage applies equally well if the Reader or the Printer is the bottleneck.

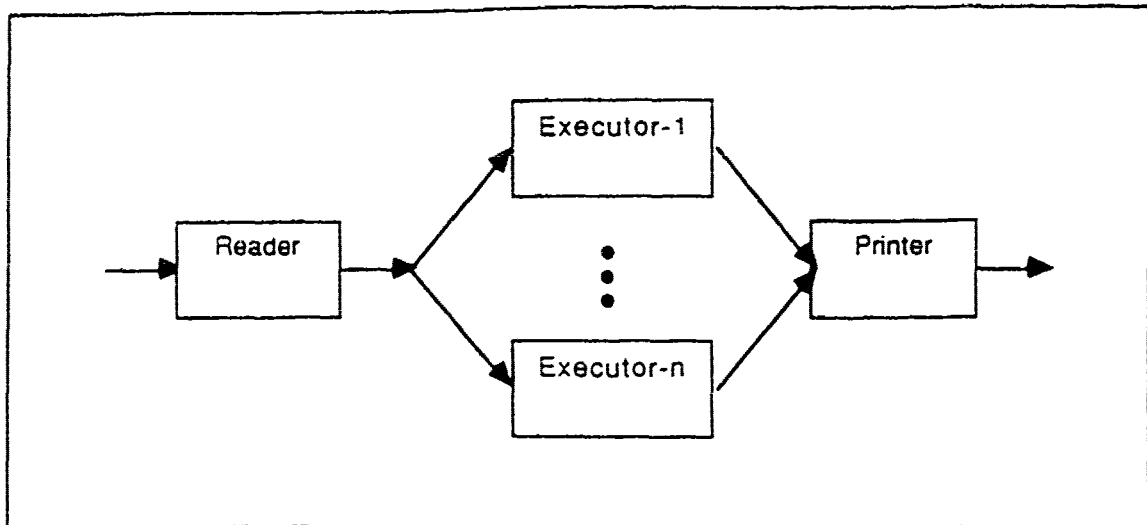


Figure 2. Decomposing a problem to obtain replication speedup.

By duplicating identical problem solving stages, we can obtain speedup from replication.

4.2. Correctness

4.2.1. Consistency

In order to achieve speedup from parallelism, we decompose a problem into smaller sub-problems, where each sub-problem is represented by an object. By doing this, we lose the luxury of mutual exclusion between the sub-problems because of interactions and dependencies that typically exist between sub-parts of a problem. For example, in the Reader-Executor-Printer problem, the simplest version is where a line may be operated upon by one process truly independently; we might want to perform ASCII to EBCDIC character conversion of each line, for instance. We organize the problem solving so that the Reader assembles fixed-length text strings, the Executor performs the conversion, and the Printer does output duties. This problem is well-suited to speedup from the simple pipeline parallelism illustrated in Figure 1. In MacLennan's value/object terminology, a "fixed-length text string" may be viewed as a value that represents the *i*-th line in the input text; the text string is read-only and it is atemporal. The trick is to show the ASCII and EBCDIC versions of a text strings as different *values* corresponding to the *i*-th line; the Executor's role is to take in ASCII values and transform them into EBCDIC values of the same line. As we will see, value passing has desirable properties in concurrent message-passing systems.

In a more complicated example, we might want to perform text compression by encoding words according to their frequency of appearance, where the Reader process counts the appearance of words and the Executor assigns words to a variable length output symbol set. The frequency table is a source of trouble; it is an object which the Reader writes and updates, and which the Executor reads. Unfortunately, the semantics we impose on the text compression task requires that the Reader complete its scan of the input text before the Executor can begin its encoding task. This dependency prevents us from exploiting pipeline parallelism.

As another example, we might want to compile a high-level language source program text (e.g. Pascal, Lisp, C) into assembly code. Suppose we allow the Reader to build a symbol table for functions and variables, and we let the Executor parse the

tokenized output from the Reader, while the Printer outputs assembly code from the Executor's syntax graph structures. In the scheme outlined here, the symbol table resides with the Reader, so whenever the Executor or Printer needs to access or update the symbol table, it must send a message to the Reader. Consistency becomes an important issue within this setup. For instance, suppose that the Executor determines on the basis of its parse, that the variable *x* has been declared global. Within a procedure, a local variable also named *x* is defined, which requires that expressions referring to *x* within this procedure use a local storage location. Suppose the end of the procedure is encountered, and since we want all subsequent occurrences to *x* to refer to the global location, the Executor marks the entry for *x* accordingly (via a message to the Reader). When the Printer sees a reference to *x*, it consults the symbol table (via a message to the Reader) to determine which storage location should be used: if by misfortune the Printer happens to be handling an expression within the procedure containing the local *x*, and the symbol table has already been updated, incorrect code will be generated. The essential point is that the symbol table is an object; as we defined earlier, it is shared by several parallel processes, and it changes. A number of fixes are possible, including distinguishing variables by the procedure they occur within, but this example illustrates that the presence of objects in concurrent program raises a need to deal with consistency.

Consistency is the property that some invariant condition or conditions describing correct behavior of a program holds over all objects in all parallel processes. This is typically difficult to achieve in a concurrent program, since the program itself consists of a sequential list of statements for each individual process or object, while consistency applies to an ensemble of objects. The field of distributed systems focuses on difficulties arising from consistency maintenance [Comafion 85, Weihl 85, Filman 84]. Smith [Smith 81] refers to this programming goal as the development of a problem-solving protocol.

The work of Schlichting and Schneider [Schlichting 83] is particularly relevant for our situation: they study partial correctness properties of unreliable datagram asynchronous message-passing distributed systems from an axiomatic point of view. They describe a number of sufficient conditions for partial correctness on an asynchronous distributed system:

- monotonic predicates,
- predicate transfer with acknowledgements.

An predicate is *monotonic* if once it becomes true, it remains so. For example, if the Reader process maintains a count of the lines in the variable `totalLines`, and it encounters the last line in the input text, as well having seen all previous lines, then it might send the predicate *P*, "`totalLines = 16.`" to the Executor and to the Printer. The Printer process might use this informat. . even before it has received all the lines, to check if sufficient resources exist to complete the job, for instance. Intuitively, it is valid to assert the total number of lines in the input text because that fact remains unchanged (assuming the input text remains fixed for the duration of the job). Formally, the Reader maintains the following invariant condition on the predicate *P*:

Invariant: "message not sent" or "*P* is true"

In contrast, an assertion that the current line is 12, as in "`currentLine = 12.`" changes as each line is processed by the Reader. The monotonic criterion cannot be used to guarantee the correctness of this assertion.

A technique to achieve correctness without monotonic predicates is to use acknowledgements. The idea is to require the sender to maintain the truth condition of a predicate or assertion until an acknowledgement from the receiver returns. In the Reader-Executor-Printer example, the Reader follows the convention that once it asserts "currentline = 12," it will refrain from further actions that would violate this fact until it receives an acknowledgement from the Executor. This protocol allows the Executor to perform internal processing, queries to the Reader, and updates to the Reader, all with the assurance that the current line will remain unchanged until the Executor acknowledges the assertion, thereby signalling that the Reader may proceed to change the assertion. Formally, the Reader and Executor maintain the following invariant condition on the predicate P:

Invariant: "message not sent" or "P is true" or "acknowledgement received"

Note that each technique has drawbacks, despite their guarantees of correctness. For the monotonic predicate technique, the challenge is to define a problem decomposition and solution protocol for which monotonic predicates are meaningful. In particular, if a problem decomposition truly allows transfer of values between processes, then by the semantics of values as we have defined them, values are automatically monotonic. This explains in formal terms why a "data flow" problem decomposition that passes values avoids difficult problems related to consistency. For the predicate acknowledgement technique, we may address problems that do not clearly admit monotonic predicates, but we lose concurrency in the assert-acknowledge cycle. Less concurrency tends to translate into less speedup. In the worst case, we may lose so much concurrency in the assert-acknowledge cycle that we find that we have spent our efforts in decomposing the problem into sub-problems only to discover that our concurrent program performs no faster than an equivalent sequential program:

Throughout the design process, we are motivated by a desire to obtain the highest possible performance while maintaining correctness. For tasks in the problem whose durations impact the performance measures, we take the approach of looking first for problem decompositions that allow either value-passing or monotonic predicate protocols. Where neither of these are possible, we implement predicate acknowledgement protocols. In the implementation of Airtrac-Lamina, we did not have to resort to heuristic schemes that did not guarantee correctness.

For initialization tasks, the time to perform initialization tasks (e.g. creating manager objects and distributing lookup tables) is not counted in the performance metrics, but correctness is paramount. Since initialization requires the establishment of a consistent beginning state over many objects, we use the predicate acknowledgement technique to have objects initialize their internal state based on information contained in an initialization message, and then signal their readiness to proceed by responding with an acknowledgement message.

4.2.2. Mutual exclusion

Lamina objects are encapsulations of data, together with methods that manipulate the data. They constitute monitors which provide mutual exclusion over the resources they encapsulate. These monitors are "non-nested" because when a Lamina method (i.e. message handler) in the current CARE implementation invokes another Lamina method, it does so by asynchronous message passing (where the sender continues executing after the message is sent), thereby losing the mutual exclusion required for nested monitor calls. In return, Lamina gains opportunities to increase concurrency by pipelining sequences of operations.

Within the restriction of non-nested monitor calls, the programmer may use Lamina monitors to define atomic operations. If correctness were the sole concern, the programmer could develop the entire problem solution within a single method on a single object; but this is an extreme case. The entire enterprise of designing programs for multiprocessors is motivated by a desire for speedup, and monitors provide a base level of mutual exclusion from which a correct concurrent program may be constructed.

The critical design task is to determine the data structures and methods which deserve the atomicity that monitors provide. The choice is far from obvious. For example, in the ASCII-to-EBCDIC translator example, we assumed the Executor process sequentially scanning through the string, translating one character at a time. We see that the translation of each character may be performed independently, so a finer-grained problem decomposition is to have many Executor processes, each translating a section of the text line. In the extreme, we can arrange for each character to be translated by one of many replicated Executor processes. Choosing the best decomposition is a function of the relative costs of performing the character translation versus the overhead associated with partitioning the line, sending messages, and reassembling the translated text fragments (in the correct order!). The answer depends on specific machine performance parameters and the type of task involved, which in our example is the very simple job of character translation, but might in general be a time-consuming operation. Unfortunately, the programmer often lacks the specific performance figures on which to base such decisions, and must choose a decomposition based on subjective assessments of the complexity of the task at hand, weighed against the perceived run-time overhead of decomposition, together with the run-time worries associated with consistency maintenance. On the issue of how to choose the best "grain-size" for problem solving, we can offer no specific guidance. However, since the CARE-Lamina simulator is heavily instrumented, it lets the programmer observe the relative amount of time spent in actual computation versus overhead activities.

In addition to providing mutual exclusion, Lamina also encourages the structured programming style that results from the use of objects and methods. In particular, mutual exclusion may be exploited without necessarily building large, monolithic objects and methods that might reflect poor software engineering practice. Since Lamina itself is built on Zetalisp's Flavors system [Weinreb 80], it is easy for the programmer to define object "flavors" with instance variables and associated methods to be atomically executed within a Lamina monitor. This can provide important benefits of modularity and structure to the software engineering effort.

To summarize, Lamina objects and methods may be viewed as non-nested monitor constructs that provide the programmer with a base level of mutual exclusion. The potential for additional concurrency and problem-solving speedup increases as finer decompositions of data and methods are adopted. However, these benefits must be weighed against the difficulties of maintaining consistency between objects in a concurrent program. Two techniques for maintaining consistency have been described, differing in their applicability and impact on concurrency.

4.3. Dependence graph programs

The previous sections have defined concepts relevant to the dual goals of achieving speedup and correctness. This section builds upon those concepts to provide a framework in which tradeoffs between speedup and correctness may be examined. A *dependence graph program* is an abstract representation of a solution to a given problem in which values flow between nodes in a directed graph, where each node applies a function to the values arriving on its incoming edges and sends out a value on zero or more outgoing

edges. The edges correspond to the dependencies which exist between the functions [Arvind 83]. A *pure* dependence graph program is one in which the functions on the nodes are free from side effects; in particular, a pure dependence graph program prohibits a function from saving state on any node. (Note that this definition does not preclude a system-level program on a node from handling a function $f(x, y)$ by saving the value of x if the value of x arrives before the value for y . Strictly speaking, an implementation of an f function node must save state, but this state is invisible to the programmer.) A *hybrid* dependence graph program is one in which one or more nodes save state in the form of local instance variables on the node. Functions have access to those instance variables.

Gajski et. al. [Gajski 82] summarize the principles underlying pure data flow computation:

- asynchrony
- functionality.

Asynchrony means that all operations are executed when and only when the required operands are available. *Functionality* means that all operations are functions, that is, there are no side effects.

Pure dependence graph programs have two desirable properties. First, consistency is guaranteed by design. As we have defined it, there are only values and transformations applied to those values. There are no objects to cause inconsistency problems. Second, we can theoretically achieve the maximal amount of parallelism in the solution, and if we ignore overhead costs, maximize speedup in overall performance. This follows from the asynchrony principle, which means that in the ideal case, we can arrange for each computation on a node to proceed as soon as all values on the incoming edges are available.

Hybrid dependence graph programs allow side effects to instance variables on nodes, thereby making it more convenient and straightforward to perform certain operations, especially those associated with lookup and matching. This immediately introduces objects into the computational model and raises the usual concerns about consistency and correctness.

We will use dependence graph programs to serve two purposes. First, we depict the dependencies contained within a problem. Second, we explain why we made certain design decisions in solving the Airtrac problem; in particular, we show why we impose certain consistency requirements on the problem solving protocol. A dependence graph serves as an abstract representation of a problem solution, rather than a blueprint for actual implementation. Specifically, we want to avoid the pitfall of using a dependence graph program to dictate the actual problem decomposition. Overhead delays associated with message routing/sending and process invocation degrade speedup from the theoretical ideal if the actual implementation chooses to decompose the problem down to the grain-size typically found in a dependence graph representation. Given an arithmetic expression, for instance, it may not be desirable to define the grain-size of primitive operations at the level of add, subtract, and multiply. This may lead to the undesirable situation where excessive overhead time is consumed in message packing, tagging, routing, packing, matching, unpacking, and so forth, only to support a simple add operation.

Consider the following numerical example from Gajski et. al. [Gajski 82]. The pseudo-code representation of the problem is as follows:

```

input d, e, f
  c0 = 0
for i from 1 to 8 do
  begin
    ai = di / ei
    bi = ai * fi
    ci = bi + ci-1
  end
output a, b, c

```

One possible dependence graph program for this problem is shown in Figure 3. This is the same graph presented by Gajski et. al. They assume that division takes three processing units, multiplication takes two units, and addition takes one unit. As noted in their paper, the critical path is the computational sequence $a_1, b_1, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8$; the lower bound on the execution time is 13 time units.

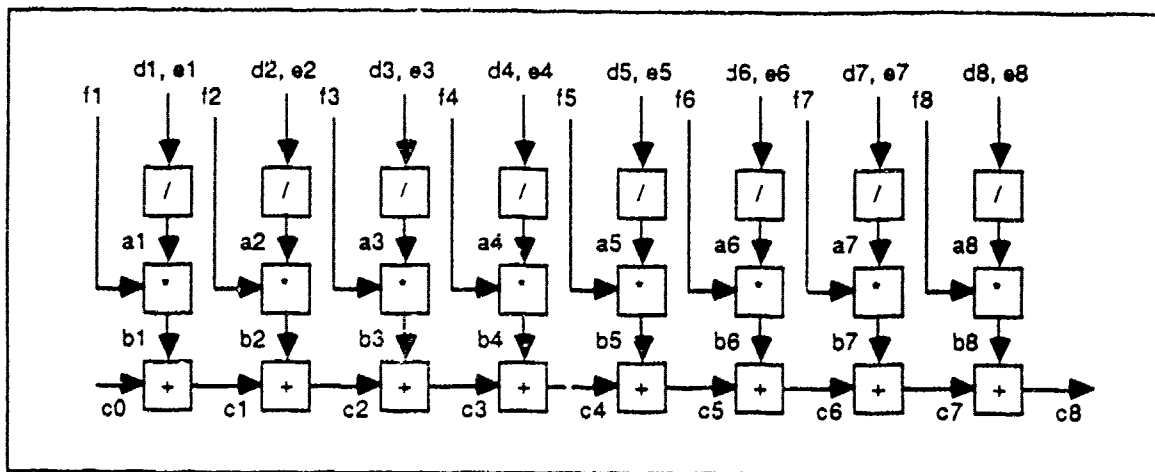


Figure 3. A dependence graph program for a simple numerical computation.

A possible concurrent program implementation would be to assign eight processes to compute the quantities b_1, \dots, b_8 , and a ninth to combine the b_i and output c_1, \dots, c_8 . Such an arrangement maximizes the decomposition of the problem into sub-problems that may run concurrently, while minimizing the communication overhead. For instance, there is no loss in combining the computation of c_1, \dots, c_8 into a single process because of the inherently serial nature of this particular computation.

Another concurrent program might choose a slightly different decomposition and partition the computation of c_1, \dots, c_8 into, say, three processes: $c_1-c_2-c_3$, $c_4-c_5-c_6$, and c_7-c_8 . This arrangement uses 11 processes versus the 9 processes in the previous example. While this leads to no improvement in the lower bound of 13 time units for a single computation with d, e , and f , it shows an improvement with repeated computations with different values of the input arrays, d, e , and f . For instance, this allows one computation to be summing on the c_7-c_8 process while another is summing on the $c_4-c_5-c_6$ process. Depending on the complexity of the computation relative to the overhead costs, it might even be worthwhile to define one process for each of the c_1, \dots, c_8 , giving 16 processes overall. This illustrates two points. First, a strictly sequential computation gives

an opportunity for pipeline concurrency if many such computations are required. Second, given a dependency graph, many possible problem decompositions are possible.

Gajski et. al. also present a different dependence graph program that is optimized to eliminate the "ripple" summation chain by a more efficient summation network. The dependence graph program for this scheme is shown in Figures 4 and 5. Figure 4 is the "top-level" definition of the program. We use the convention of using a single box, optimized summation, in Figure 4 to represent the subgraph that performs the more efficient summation. Figure 5 shows the expansion of that box as a graph. Showing a dependence graph program in this way is merely a convenience; one should envision the subgraphs in their fully expanded form in the top-level dependence graph program definition.

The associative property of addition is used to derive the optimized summation function. For instance, the computation of c_8 is rewritten as follows:

$$\begin{aligned}
 c_8 &= (((((((c_0 + b_1) + b_2) + b_3) + b_4) + b_5) + b_6) + b_7) + b_8) \\
 &= (c_0 + ((b_1 + b_2) + (b_3 + b_4))) + ((b_5 + b_6) + (b_7 + b_8))
 \end{aligned}$$

By regrouping the addition operations, this dependence graph program has more parallelism, and reduces the lower bound on execution time from 13 to 9 execution time units. It is important to realize that the second program is truly different from the first; it cannot be obtained from the first by graph transformations or syntactic manipulations that do not rely on the semantics of the functions on the nodes.

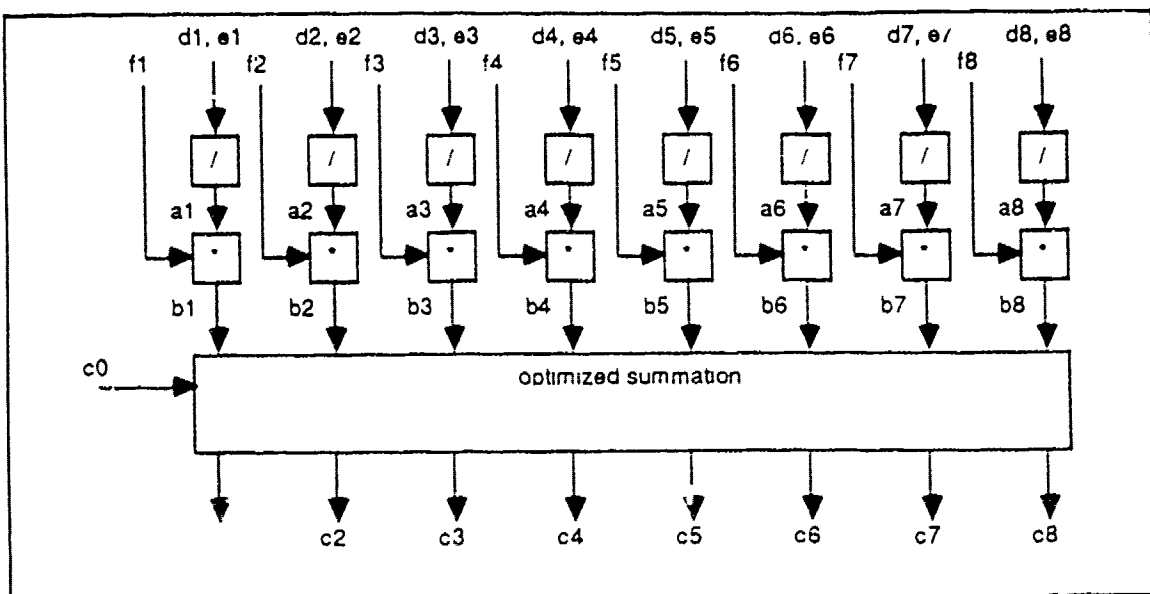


Figure 4. A dependence graph program for the simple numerical computation.

This uses optimization of the recurrence relation using the associative property of addition. This represents the "top-level" definition of the solution. The optimized summation subgraph is shown here a single box, and is shown in expanded form in Figure 5.

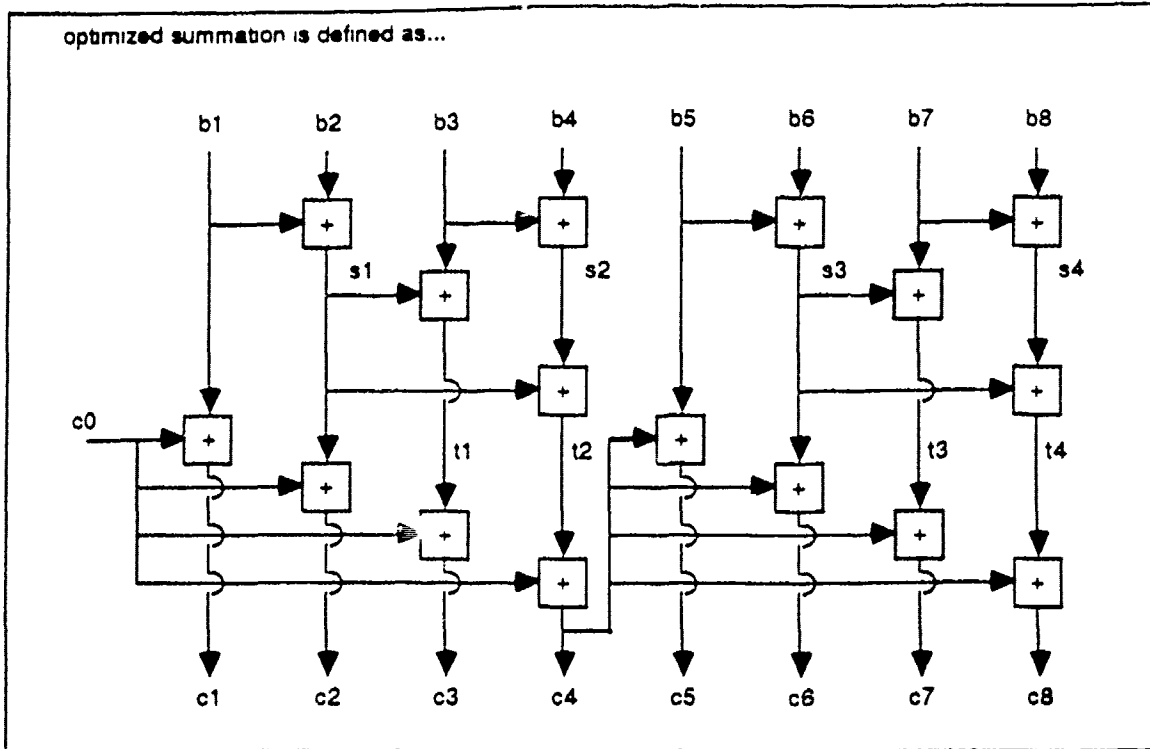


Figure 5. Definition of the "optimized summation" subgraph.

This example highlights several points. First, a given problem may have more than one valid dependence graph program. In the example presented here, the use of knowledge about the underlying semantics of the addition function allows more parallelism. Second, the dependence graph program serves as an intermediate representation from which the solution may be defined for a parallel machine. Third, the dependence graph program does not necessarily make a commitment to the form of the concurrent program. Fourth, for convenience we may describe a dependence graph program as a top-level graph, together with several subgraph definitions.

5. The Airtrac problem

In Airtrac, the problem is to accept radar track data from one or more sensors that are looking for aircraft. Figure 6 depicts a region under surveillance as it might be seen on a display screen at a particular snapshot in time. (Whereas Figure 6 shows many reported sightings, an actual radar would probably show only the most recent sighting.) Locations are designated as either good or bad, where a bad location is illegal or unauthorized, and a good location is legal. The "X" and "Y" symbols represent locations of a good and bad airport, respectively. The locations of radar and acoustic sensors are also shown. The small circles represent track reports that show the location of a moving object in the region of coverage.

Track reports are generated by underlying signal processing and tracking system, and contain the following information:

- location and velocity estimate of object (in x-y plane)

- location and velocity covariance
- the time of the sighting, called the *scan time*
- *track id* for identification purposes.

We would like to answer the following questions in real-time:

- Is an aircraft headed for a bad destination?
- Is it plausible that an aircraft is engaged in smuggling?

By "smuggling" we mean the act of transporting goods from a region or location designated as bad to another bad location. For instance, flying from an illegal airstrip and landing at another illegal airstrip constitutes smuggling.

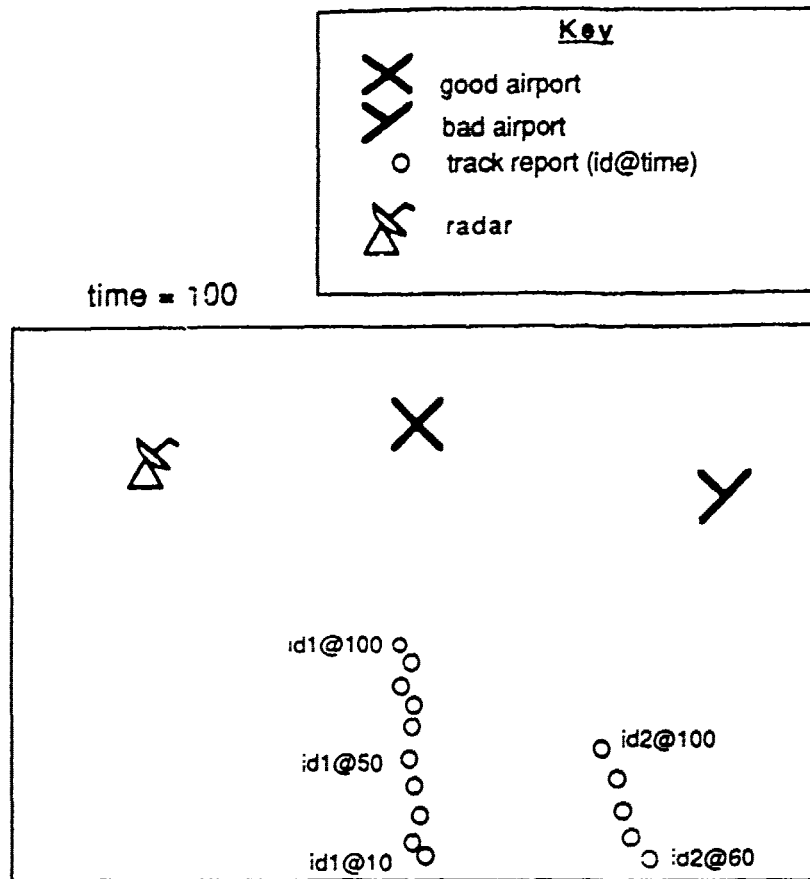


Figure 6. Input to Airtrac.

This shows the inputs that the system receives. The small circles represent estimated positions of objects from radar or acoustic sensors tagged by their identification number and observation time; the goal of the system is to use the time history of those sightings to infer whether an aircraft exists, its possible destinations, and its strategy.

This paper describes our implementation of a solution of a portion of the Airtrac problem. We refer to this portion as the *data association* module. Figure 7 depicts the desired output of the data association step: groupings of reports with the same track id into straight-line, constant-speed sections. These are called *Radar Track Segments*, and have four properties:

- If the Radar Track Segments contains three or more reports, a best-fit line is computed. If the fit is sufficiently good, the segment is declared *confirmed*.
- If a best-fit line has been computed, each subsequent report must fit the line sufficiently closely. If so, the Radar Track Segments remains confirmed. Otherwise, the report that failed to fit (call it the non-fitting report) is treated specially, and the track is declared *broken*.
- A broken track causes the non-fitting report and subsequent reports to be used to form a new Radar Track Segment.

- The last report for a given track id defines that a track is declared *inactive*.

The remaining parts of the Airtrac problem have not yet been implemented as of this writing, but are described more fully elsewhere [Minami 87, Nakano 87].

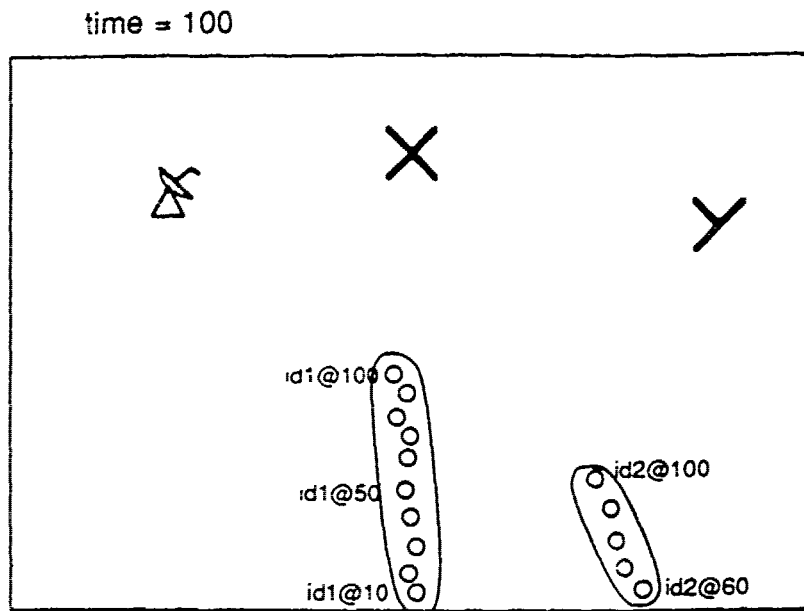


Figure 7. Grouping reports into segments in data association.

This shows the first step in problem solving, grouping the reports into straight-line sections called Radar Track Segments.

5.1. Airtrac data association as dependence graph

Figure 8 shows the Airtrac data association problem as a dependence graph program. On a periodic basis, track reports consisting of position and velocity information for a set of track ids enters the system. Two operations are performed. First, the system checks if a track id is being seen for the first time. If so, a new track-handling subgraph is created. A track-handling subgraph is shown in Figure 8 as a functional box labeled "handle track i," which expands into a graph as shown in Figure 9. Second, the system checks if any track id seen in a previous time has disappeared. If so, it generates an inactivation message for the `handle track` subgraph for the particular track id that disappeared. If the track id has been seen previously, then it is sent to the appropriate `handle track` subgraph.

We distinguish between pure functional nodes, shown as rectangles, and side-effect nodes, shown as rounded rectangles. One use of side-effect nodes is to keep track of which track ids have been seen at the previous time. For instance, by performing set difference operations against the current set of track ids, it is possible to determine the disappeared and new tracks:

```
disappearedTracks = previousTracks - currentTracks
```


$newTracks = currentTracks - previousTracks$

One way to implement this scheme is to have the `ids` `disappeared?` and `id` `previously seen?` nodes update local variables called `previousTracks` and `currentTracks`, as successive track reports arrive.

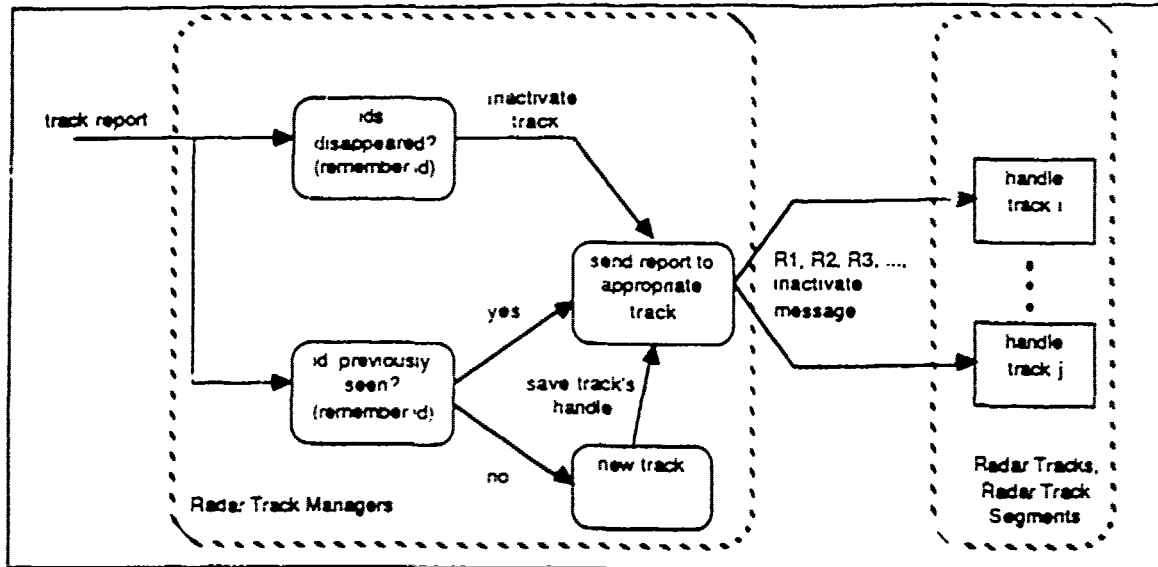


Figure 8. Dependence graph program representation of Airtrac data association.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

Besides detecting new and disappeared tracks, side-effect nodes are used to create a new track-handling subgraph, and maintain the lookup table between track id and the message pathway to each track-handling subgraph. `new track` creates a new track handler subgraph. Whenever a new track is encountered, `send report to appropriate track` is notified, so that subsequent reports will be routed correctly. This arrangement requires that one and only one track handler exist for each track id. `send report to appropriate track` saves the handle⁴ to the track handler created by `new track`, sorts the incoming reports, and sends reports to their proper destinations.

In this abstract program, we implicitly assume that only one track report may be processed at a time by the four side-effect nodes in Figure 8. If we allow more than one track report to be processed concurrently, we may encounter inconsistent situations that allow, for instance, a track id to be seen in one track report, but the `send report to appropriate track` node does not yet have the handle to the required track handler subgraph when the next track report arrives. We define the program semantics to avoid these situations.

`Handle track` receives track reports for a particular id, as well as an inactivation message if one exists. It is further decomposed into a subgraph as shown in Figure 9. The

⁴A handle is analogous to a mail address in a (physical) postal system: a Lamina object may use another object's handle to send messages to that object. Since the message passing system utilizes dynamic routing and we assume that an object remains stationary once created, the handle does not need to encode any information about the particular path messages should follow.

nodes in the handle track subgraph pass a structured value between them, called track segments. A *track segment* has the following internal structure:

- report list (a list of track reports, initially empty)
- best-fit line (a vector of real numbers describing a straight-line constant-velocity path in the x-y plane)

Each node may transform the incoming value and send a different value on an outgoing edge. *add* appends a report to the report list of a track segment. *linefit* computes the best-fit line, and if the confirmation conditions hold, sends the track segment to *confirm*. *confirm* declares the track segment as confirmed, and passes the list to *check fit*. If *linefit* fails to confirm, the earliest report in the list is dropped by *drop*, and another *add*, *linefit* box awaits the arrival of the next report to restart the cycle. The *inactivate* function waits until all reports have arrived before declaring the track inactive. Conceptually, we view the operations of *confirm* and *inactivate* as being monotonic assertions made to the "outside world," rather than value transformations to the track segment.

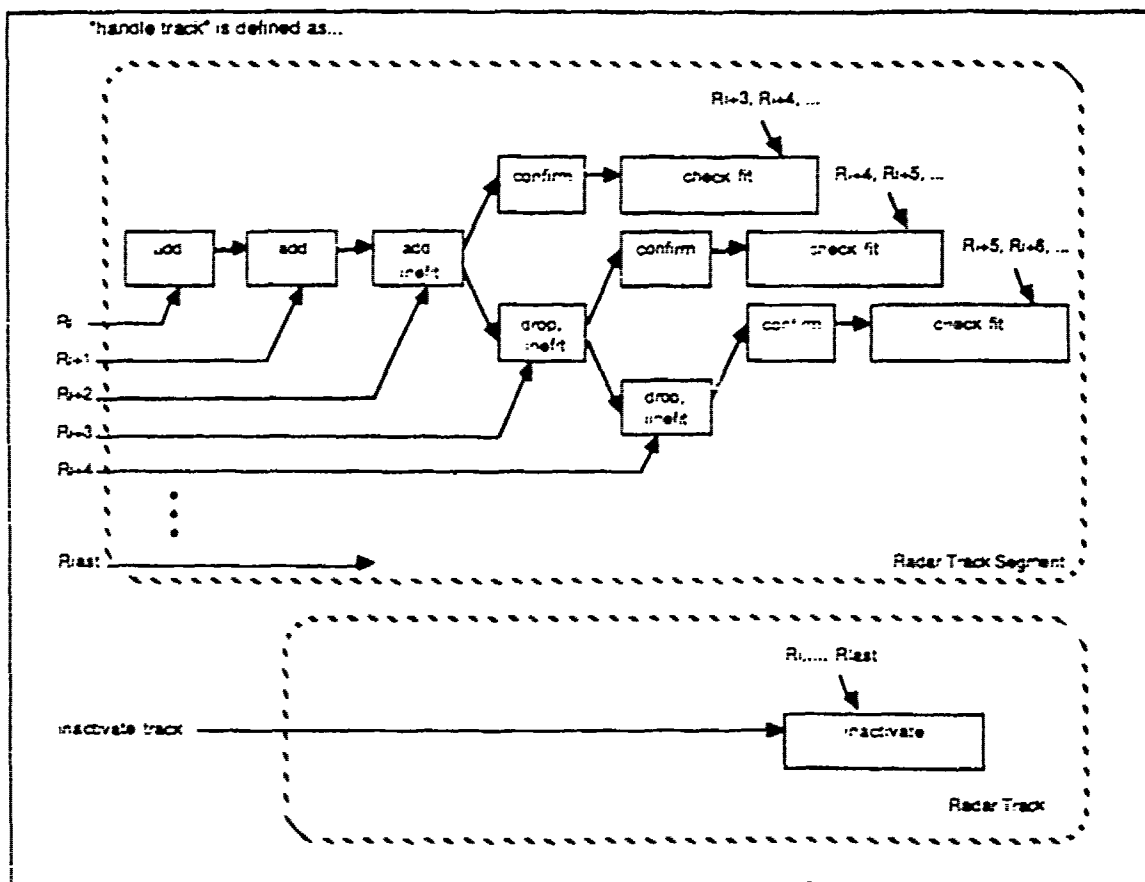


Figure 9. Decomposition of the "handle track" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

check fit itself is further decomposed into more primitive operations, as shown in Figure 10. The *linecheck* operation is similar to the *linefit* function previously

described, except that it compares a new report against the best-fit line computed during the linefit operation: if the new report maintains the fit, the report list is sent to the OK box, and this cycle is repeated for the next report. If the linecheck operation fails, then the track is declared broken, a new track segment is defined. This track segment is sent the report that failed the linecheck operation, in addition to all subsequent reports for this particular track id. The track handling cycle is repeated as before.

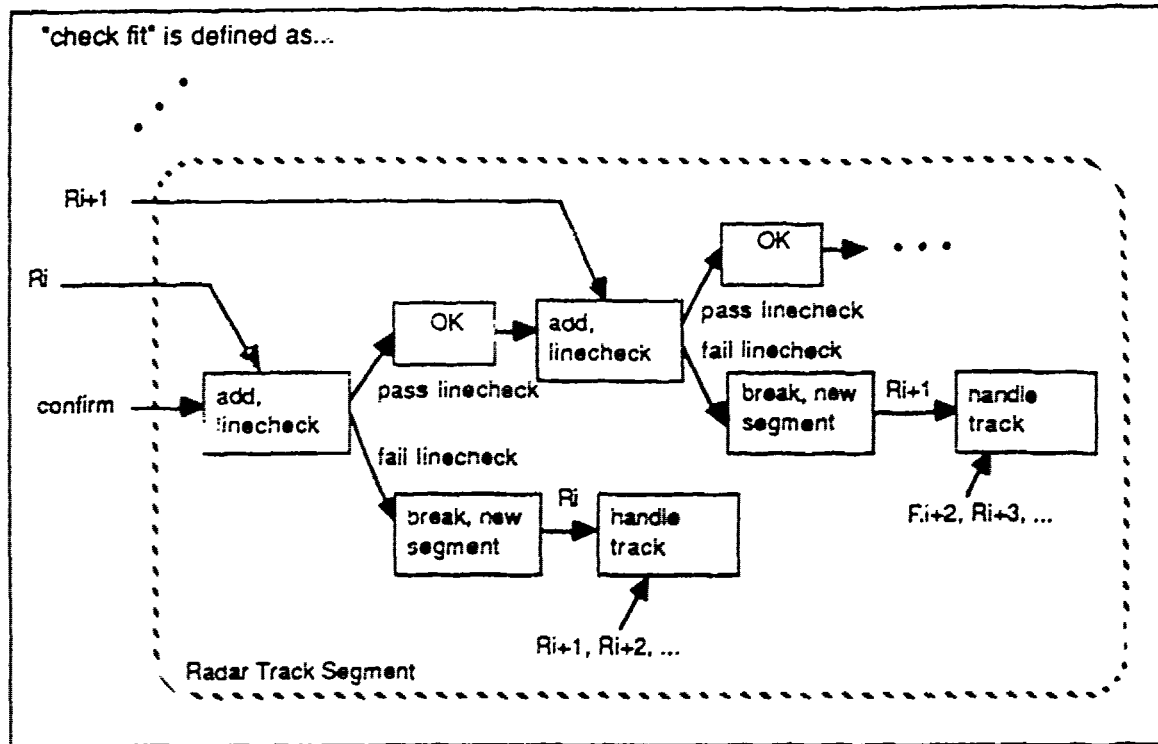


Figure 10. Decomposition of the "check fit" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

A number of observations may be made about the dependence graph program described in this section. First, the sequence of the reports matters. The graph structure clearly depicts the requirement that the incorporation of the R_i -th report into the track segment by the *add* operation must wait until all prior reports, R_1, \dots, R_{i-1} , have been processed. This is true for the *linefit*, *linecheck*, and *inactivate* functions. Second, this program avoids the saving of state information except in the operations that must determine whether a given track id has been previously seen, and in the sorting operation where track reports are routed to the appropriate track handler. Except for these, we find that the problem may be cast in terms of a sequence of value transformations. Third, the program admits the opportunity for a high degree of parallelism. Once the track handler for a given track id has been determined, the processing within that block is completely independent of all other tracks. Fourth, the opportunity for concurrency within the handling of a particular track is quite low, despite the outward appearance of the decompositions shown in Figures 8 and 9. Indeed, an analysis of the dependencies shows that reports must be processed in order of increasing scantime. Fifth, unlike certain portions of the dependence graph that have a structure that is known *a priori*, the track

handler portions of the graph have no prior knowledge of the track ids that will be encountered during processing, implying that new tracks need to be handled dynamically.

5.2. Lamina implementation

In this section, we express the solution to the data association problem as a set of Lamina objects, together with a set of methods on these objects which embody the abstract solution specification presented in the previous section.

Figure 11 shows how we decompose the Airtrac problem for solution by a Lamina concurrent program. We define six classes of objects: Main Manager, Input Simulator, Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment. Some objects, referred to as *static objects*, are created at initialization time, and include the following object classes: Main Manager, Input Simulator, Input Handler, and Radar Track Manager objects. Others are referred to as *dynamic objects*, are created at run-time in response to the particular input data set, and include the following object classes: Radar Track and Radar Track Segment.

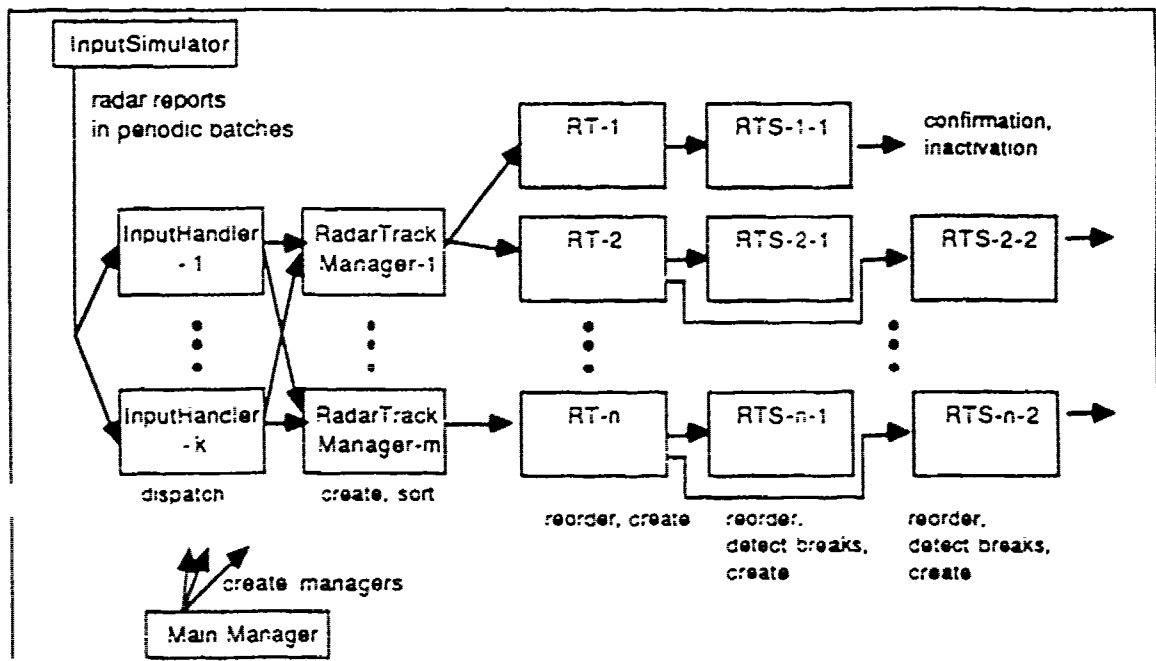


Figure 11. Object structure in the data association module.

Each object is implemented as a Lamina object, which in Figure 11 corresponds to a separate box. The problem decomposition seeks to achieve concurrent processing of independent sub-problems. The Lamina message-sending system provides the sole means of message and value passing between objects. Wherever possible, we pass values between objects to minimize consistency problems, and to minimize the need for protocols that require acknowledgements. For example, we decompose our problem solving so that we require acknowledgements only during initialization where the Main Manager sets up the communication pathways between static objects.

With respect to the dependence graph program, the Lamina implementation takes a straightforward approach. All of the side-effect functions contained in Figure 8, together with some operations to support replication, reside in the Input Handler and Radar Track

Manager object classes. Objects in these two classes are static; we create a predetermined number of them at initialization time to handle the peak load of reports through the system. Replication is supported by partitioning the task of recognizing new and disappeared track ids among Radar Track Managers according to a simple modulo calculation on the track id. Given the partitioning scheme, each Radar Track Manager operates completely independently from the others. Thus, although it needs to maintain a set of objects (e.g. the current tracks, previous tracks), the objects are encapsulated in a Lamina object. Access to and updating of these objects is atomic, providing the mutual exclusion required to assure correctness as specified by the dependence graph program.

Functions in Figures 9 and 10 reside mostly in objects of the Radar Track Segment class, with the inactivation function being performed by objects of the Radar Track class. Objects of these two classes are dynamic: we create objects at run-time in response to the specific track ids that are encountered. For any particular track id, one Radar Track object together with one or more Radar Track Segment objects are created. A new Radar Track Segment is created each time the track is declared broken, which may occur more than once for each track id. Unlike the dependence graph program where we postulate a track segment as a value successively transformed as it passes through the graph, the Lamina implementation defines a Radar Track Segment object with instance variables to represent the evolving state of the track segment. We implement all the major functions on track segments as Lamina methods on Radar Track Segment objects. Again, Lamina objects provide mutual exclusion to assure correctness.

Although nothing in the problem formulation described here indicates why we create multiple Radar Track Segments for a given track, we do so in anticipation of adding functionality in future versions of Airtrac-Lamina. From examination of Figure 10, we see that given any sequence of reports R_i , and any pattern of broken tracks, we obtain no additional concurrency by creating a new Radar Track Segment when a track is declared broken. This is because in the dependency graph program presented here, no activity occurs on one Radar Track Segment after it has created another Radar Track Segment. However, we anticipate that in subsequent versions of Airtrac-Lamina, a Radar Track Segment will continue to perform actions even after a track is declared broken, such as to respond to queries about itself, or to participate in operations that search over existing Radar Track Segments.

Logically, the semantics of the dependency graph program and the Lamina program are equivalent, as they must be. The difference is that the former requires a graph of indefinite size, where its size corresponds to the number of reports comprising the track. The latter requires a quantity of Radar Track Segment objects equal to one plus the number of times the track is declared broken. Although we can easily conceptualize a graph of indefinite size in a dependency graph program, we cannot create such an entity in practice. Because object creation in Lamina takes time, we try to minimize the number of objects that are created dynamically, especially since their creation time impacts the critical path time. A poor solution is to dynamically create the objects corresponding to an indefinite-sized graph as we need them. A better solution is to create a finite network of objects at initialization time, with an implicit "folding" of the infinite graph onto the finite network, thereby avoiding any object-creation cost at run-time. Our Lamina program, in fact, uses a hybrid of these two approaches, folding an indefinite "handle track" graph onto each Radar Track Segment object, and creating a new Radar Track Segment object dynamically when a track is declared broken. By this mechanism, we model transformations of values between graph nodes by changes to instance variables on a Lamina object. The effect on performance is beneficial. Relative to the first solution, we incur less overhead in message sending between objects because we have fewer objects. Relative to the second solution, we create objects that correspond to track ids that appear in the input data stream as they are

needed, which has the effect of bringing more processors to bear on the problem as more tracks become visible.

Both the Radar Track and Radar Track Segment collect reports in increasing scantime sequence. They do so because of the ordering dictated by the dependence graph program, and because the Lamina implementation at the time the experiments were performed did not provide automatic message ordering. Moreover, we know that simply collecting reports in order of receipt leads to severe correctness degradation. For instance, if the scantimes are not contiguous, the scheme by which a Radar Track Segment computes the line-fit leads to nonsensical results because best-fit lines will be computed based on non-consecutive position estimates, leading to erroneous predictions of aircraft movement. To circumvent these problems, we use application-level routines to examine the scantime associated with a report, and queue reports for which all predecessors have not already been handled. These routines effectively insulate the rest of the application from message receipt disorder, and allow the Lamina program to successfully use the knowledge embodied in the dependency graph program.

To indicate the size of the problem, a typical scenario that we experimented with contained approximately 800 radar track reports comprising about 70 radar tracks. At its peak, there is data for approximately 30 radar tracks arriving simultaneously, which roughly corresponds to 30 aircraft flying in the area of coverage.

The correspondence between the Lamina objects in the implementation presented here and the primitive operations embodied in the dependence graph program is shown in the Table 1. The functions described in the dependence graphs are implemented on Radar Track Manager, Radar Track, and Radar Track Segment objects. The Main Manager and Input Simulator perform tasks not mentioned in the dependence graph program. Their tasks may be viewed as overhead: the Main Manager performs initialization, and Input Simulator simulates the input data port. The Input Handler's job is to dispatch incoming reports to the correct Radar Track Manager, thereby supporting the replication of the Radar Track Manager function across several objects. In this way the task of the Input Handler may be viewed as a functional extension of the Radar Track Manager tasks.

Table 1. Correspondence of Lamina objects with functions in the dependence graph program

<u>Lamina object</u>	<u>Corresponding dependence graph program operation</u>
Main Manager	-none- (Create the manager objects in the system at initialization time.)
Input Simulator	-none- (Simulate the input data port that would exist in a real system. This function is an artifact of the simulation.)
Input Handler	-none- (Allows replication of the Radar Track Manager objects; this may be viewed as a functional extension of the Radar Track Manager.)
Radar Track Manager	ids disappeared?, id previously seen?, new track, send report to appropriate track
Radar Track	add, inactivate
Radar Track Segment	add, linefit, confirm, drop, inactivate, linecheck, OK, break, new segment

Table 1 also shows that we decompose the problem to a lesser extent than might be suggested by the dependence graph program, but the overall level of decomposition is still high. We "fold" the dependence graph onto a smaller number of Lamina objects, but we nonetheless obtain a high degree of concurrency from the independent handling of separate tracks. Additional concurrency comes from the pipelining of operations between the following sequence of objects: Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment.

6. Experiment design

Given our experimental test setup, there are a large number of parameter settings, including the number of processors, the choice of the input scenario to use, the rate at which the input data is fed into the system, the number of manager objects to utilize; for a reasonable choice of variations, trying to run all combinations is futile. Instead, based on the hypotheses we attempted to confirm or disconfirm, we made explicit decisions about which experiments to try. We chose to explore the following hypotheses:

- Performance of our concurrent program improves with additional processors, thereby attaining significant levels of speedup.

- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.
- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

Long wall-clock times associated with each experiment and limited resources forced us to be very selective about which experiments to run. We were physically unable to explore the full combinatorial parameter space. Instead, we varied a single experimental parameter at a time, holding the remaining parameters fixed at a base setting. This strategy relied on an intelligent choice of the base settings of the experimental parameters.

We divided our data gathering effort into two phases. First, we took measurements to choose the base set of parameters. Our objective was to run our concurrent program on a system with a large number of processors (e.g. 64), picking an input scenario that feeds data sufficiently quickly into the system to obtain full but not overloaded processing pipelines. We used a realistic scenario that has parallelism in the number of simultaneous aircraft so that nearly all the processors may be utilized. Finally, we chose the numbers of manager objects so the managers themselves do not limit the processing flow. The goal was to prevent the masking of phenomena necessary to confirm or disconfirm our hypotheses. For example, if we failed to set the input data rate high enough, we would not fully utilize the processors, making it impossible that additional processors display speedup. Similarly, if we failed to use enough manager objects, the overall program performance would be strictly limited by the overtaxed manager objects, again negating the effect of additional processors.

Based on measurements in phase one, we chose the following settings for the base set of parameter settings:

- 64 processors,
- Many-aircraft scenario (described more fully below),
- Four input handler objects,
- Four radar track manager objects,
- Input data rate of 200 scans per second.

These settings give system performance that suggests that processing pipelines are full, but not overloaded, where nearly all of the processing resources are utilized (although not at 100 percent efficiency), and the manager objects are not themselves limiting overall performance.

The input data rate governs how quickly track reports are put into the system. As reference, the Airtrac problem domain prescribes an input data rate of 0.1 scan per second (one scan every 10 seconds), where a scan represents a collection of track reports periodically generated by the tracking hardware. For the purpose of imposing a desired processing load on our simulated multiprocessor, our simulator allows us to vary the input data rate. With a data rate of 200 scans per second, we feed data into our simulated multiprocessor 2000 times faster than prescribed by the domain to obtain a processing load where parallelism shows benefits. Equivalently, we can imagine reducing the performance of each processor and message passing hardware in the multiprocessor by a factor of 2000

to achieve the same effect, or with any combination of input data rate increase and hardware speed reduction that results in a net factor of 2000.

In the second phase, we vary a single parameter while holding the other parameters fixed. We perform the following set of three experiments:

- Vary the number of processors from 1 to 100.
- Vary the input scenario to use the one-aircraft scenario.
- Vary the number of manager objects.

Figure 12 shows how the many-aircraft and one-aircraft scenarios differ in the number of simultaneous active tracks. In the many-aircraft scenario, many aircraft are active simultaneously, giving good opportunity to utilize parallel computing resources. In contrast, the one-aircraft scenario reflects the extreme case where only a single aircraft flies through the coverage area at any instant, although the total number of radar track reports is similar between the two scenarios. Although broken tracks in the one-aircraft scenario may give rise to multiple track ids for the single aircraft, the resulting radar tracks are non-overlapping in time.

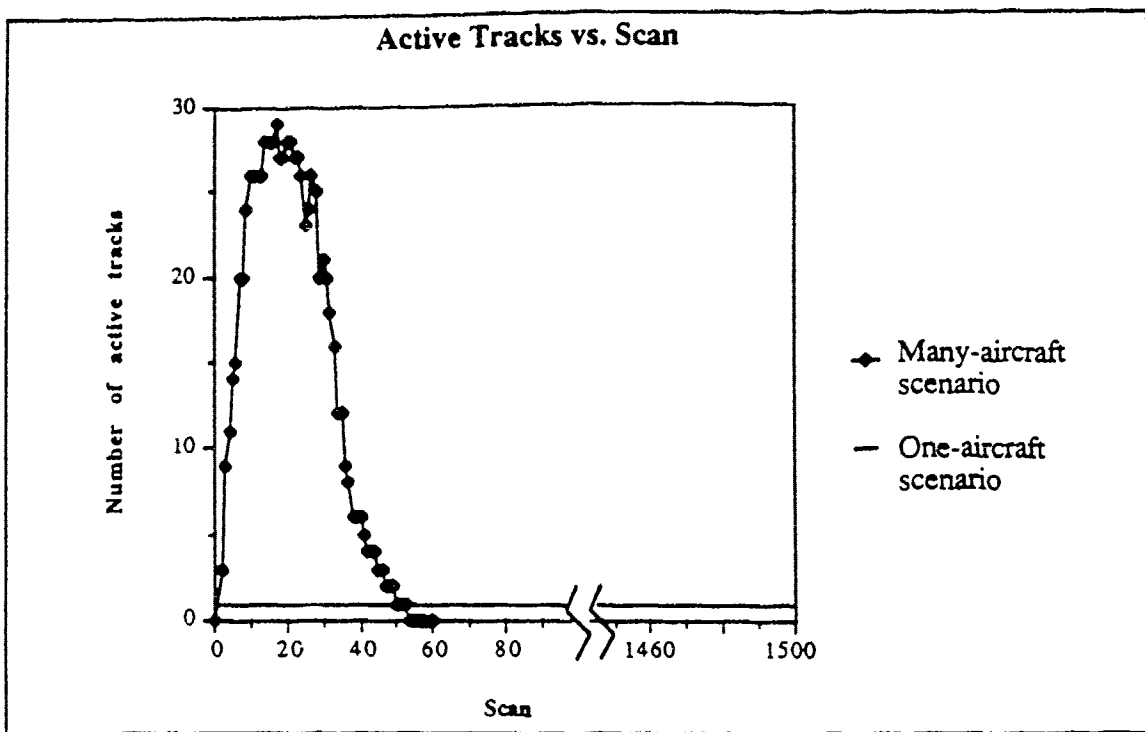


Figure 12. Comparison of the number of active tracks in the many-aircraft and one-aircraft scenarios.

This shows the number of active tracks versus the scan. The scan number corresponds to scenario time in increments of 0.1 seconds.

7. Results

7.1. Speedup

Our performance measure is latency. *Latency* is defined as the duration of time from the point at which the system receives a datum which allows it to make a particular conclusion, to the point at which the concurrent program makes the conclusion. We use latency as our performance measure instead of total running time measures, such as "total time to process all track reports," because we believe that the latter would give undue weight to the reports near the end of the input sequence, rather than weigh performance on all track reports equally.

We focus on two types of latencies: confirmation latency and inactivation latency. *Confirmation latency* measures the duration from the time that the third consecutive report is received for a given track id, to the time that the system has fitted a line through the points, determined that the fit is valid, and it asserts the confirmation. *Inactivation latency* measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts the inactivation. Since a given input scenario contains many track reports with many distinct track ids, our results report the mean together with plus and minus one standard deviation.

Figures 12 and 14 show the effects on confirmation and inactivation latencies, respectively, from varying the number of processors from 1 to 100. Boxes in the graphs

indicate the mean. Error bars indicate one standard deviation. The dashed line indicates the locus of linear speedup relative to the single processor case; its locus is equivalent to an $S_{11/1}$ speedup level of n for n processors.

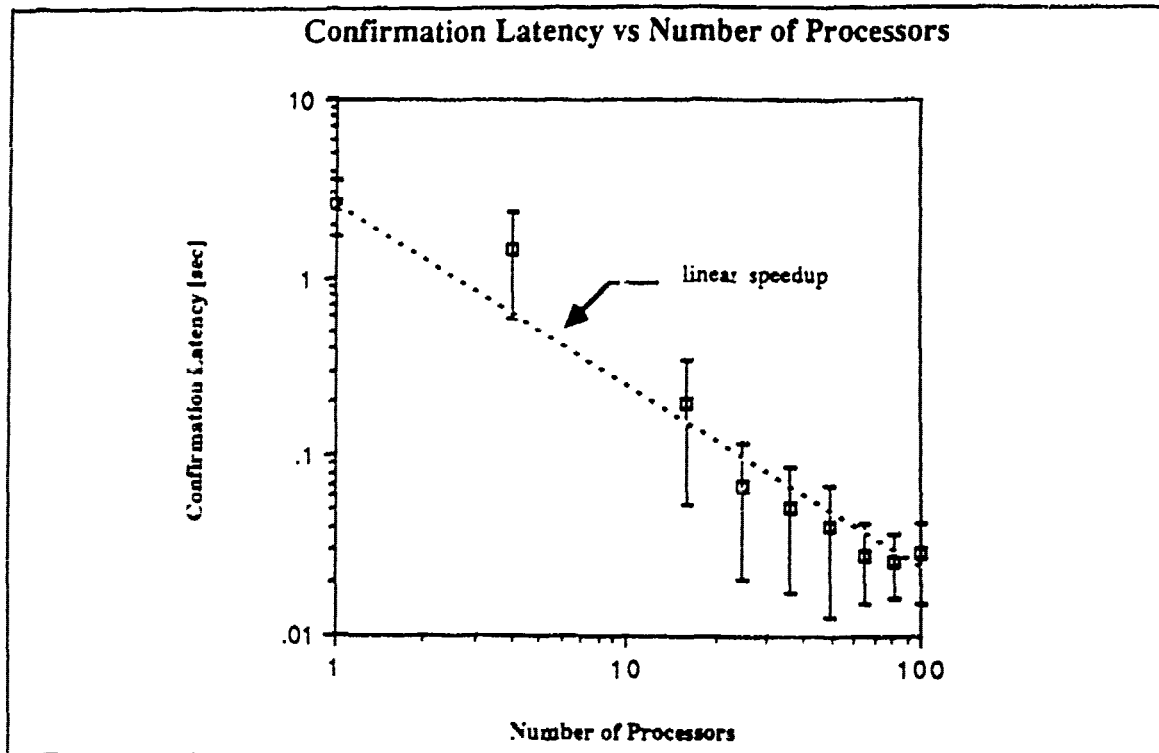


Figure 13. Confirmation latency as a function of the number of processors.

This measures the duration from the time that the third consecutive report is received for a given track id, to the time that the system has fitted a line through the points, and determined that the fit is valid.

The results for both the confirmation and inactivation show a nearly linear decrease in the mean latencies, corresponding to $S_{100/1}$ speedup by a factor of 90 for the confirmation latency, and to $S_{100/1}$ speedup by a factor of 200 for the inactivation latency. The sizes of the error bars make it difficult to pinpoint a leveling off in speedup, if there is any, over the 1 to 100 processor range.

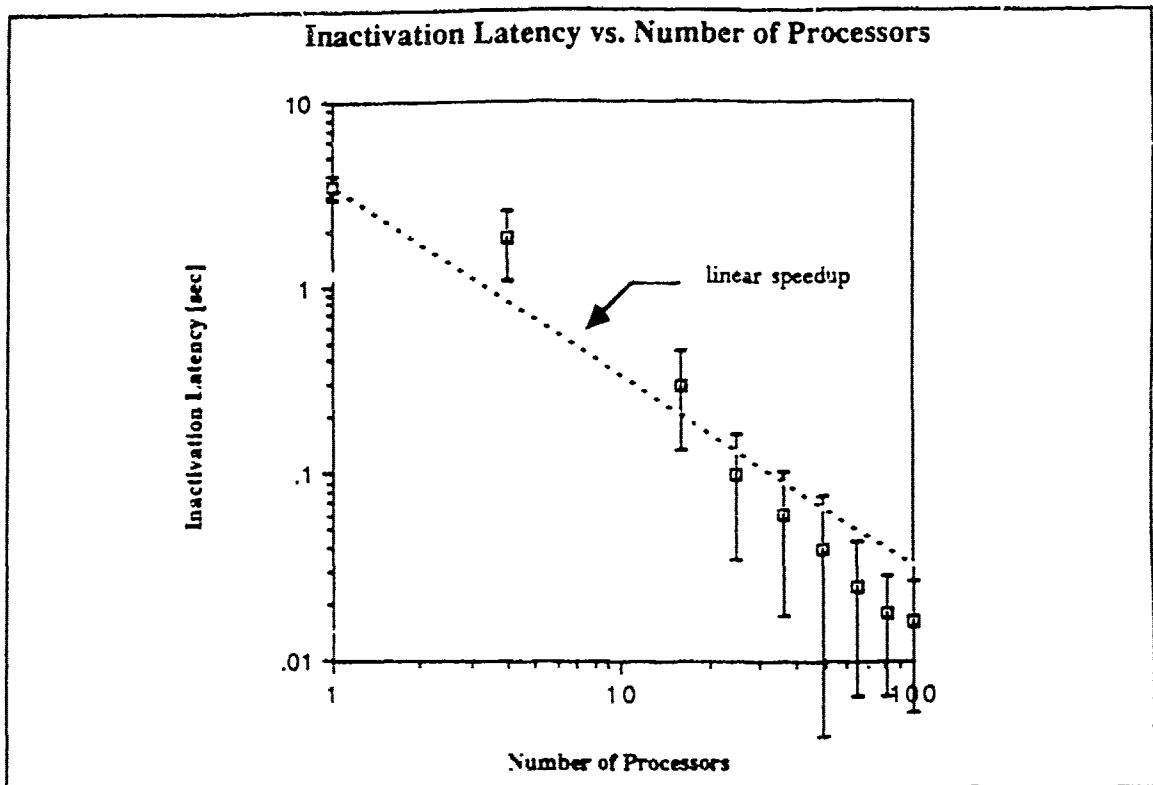


Figure 14. Inactivation latency as a function of the number of processors.

This measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts that conclusion.

7.2. Effects of replication

By replicating manager nodes, we measure the impact of the number of manager objects on performance, as measured by the confirmation latency. In one experiment we fix the number of Radar Track Managers at 4 while we vary the number of Input Handlers. In the other experiment we fix the number of Input Handlers at 4 while we vary the number of Radar Track Managers.

Figures 15 and 16 show the results. We plot the confirmation latency versus the number of managers, instead of against the number of processors as done in Figures 13 and 14.

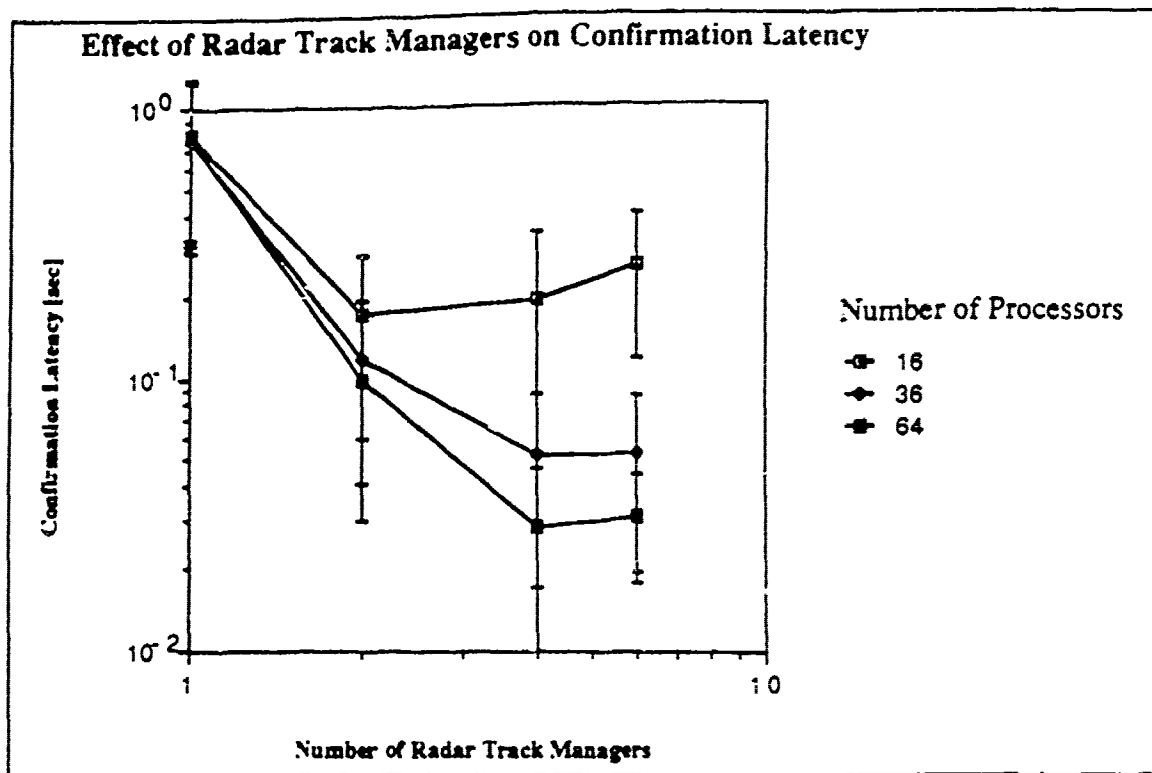


Figure 15. Confirmation latency as a function of the number of radar track managers.

We see that replicating Radar Track Manager objects improves performance: this is because increasing the number of processors does not improve performance in the single Radar Track Manager case, but does in the 4 and 6 Radar Track Managers cases (see Figure 16). Put another way, if we had not used as many as 4 Radar Track Manager objects, then our system performance would have been hampered, and might even have precluded the high degree of speedup displayed in the previous section. Comparing Figures 15 and 16, we also observe that using more Radar Track Managers helps reduce confirmation latency more significantly than using more Input Handlers.

An interesting phenomenon occurs in the 16-processor case. Although the conclusion is not definitive given the size of the error bars, increasing the number of both types of managers from 2 to 4 and 6 increases the mean latency. The likely cause is the current object-to-processor allocation scheme: because each manager object is allocated to a distinct processor, increasing the number of manager objects decreases the number of processors available for other types of objects. Given our allocation scheme (described more fully in Section 8.2), using more managers in the 16-processor case may actually impede speedup.

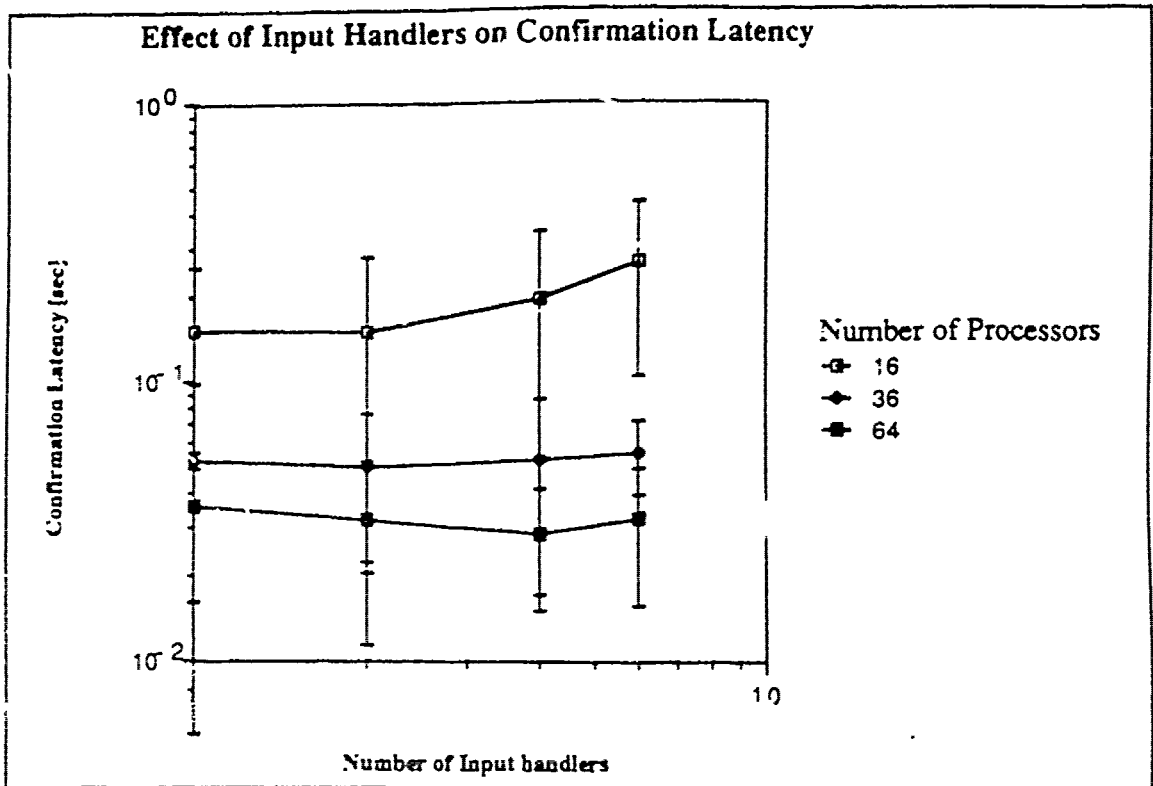


Figure 16. Confirmation latency as a function of the number of input handlers.

The optimal number of manager objects appears to sometimes depend on the number of processors. For Radar Track Managers, 2 or 4 managers is best for the 16-processor array, and 4 or 6 managers is best for the 36 and 64-processor arrays. For Input Handlers, the number of managers does not appear to make much difference, which suggests that Input Handlers are less of a throughput bottleneck than Radar Track Managers. This suggests that in practice it will be necessary to consider the intensity of the managers' tasks relative to the total task in order to make a program work most efficiently. Overall these experiments confirm that replicating objects appropriately can improve performance.

7.3. Less than perfect correctness

Our Lamina program occasionally fails to confirm a track that our reference solution properly confirms. This arises because the concurrent program does not always detect the first occurrence of a report for a given track in the presence of disordered messages. We notice the following failure mechanism. Suppose we have a track consisting of scantimes 100, 110, 120, ..., 150. Suppose that the rate of data arrival is high, causing message order to be scrambled, and that reports for scantimes 110, 120, and 130 are received *before* the report for 100. As implemented, the Radar Track object notices that it has sufficient number of reports (in this case three), and it proceeds to compute a straight line through the reports. When a report for scantime 140 or higher is received, it is tested against the computed line to determine whether a line-check failure has occurred. Unfortunately, when the report for scantime 100 eventually arrives, it is discarded. It is discarded because the track has already been confirmed, and confirmed tracks only grow in the forward direction.

Figure 9 reveals why this error causes discrepancies between the Lamina program and the reference serial program: the handle track operation in the Lamina program is given a different set of reports compared to the reference program, leading to a different best-fit line being computed. To be certified as correct, we require that the reports contained in a confirmed Radar Track Segment must be identical between the Lamina solution and the reference solution.

The lesson here is that message disordering does occur, and that it does disrupt computations that rely on strict ordering of track reports. In our experiments, the incorrectness occurs infrequently. See Figure 17. We believe that with minimal impact on latency, this source of incorrectness can be eliminated without significant change to the experimental results.

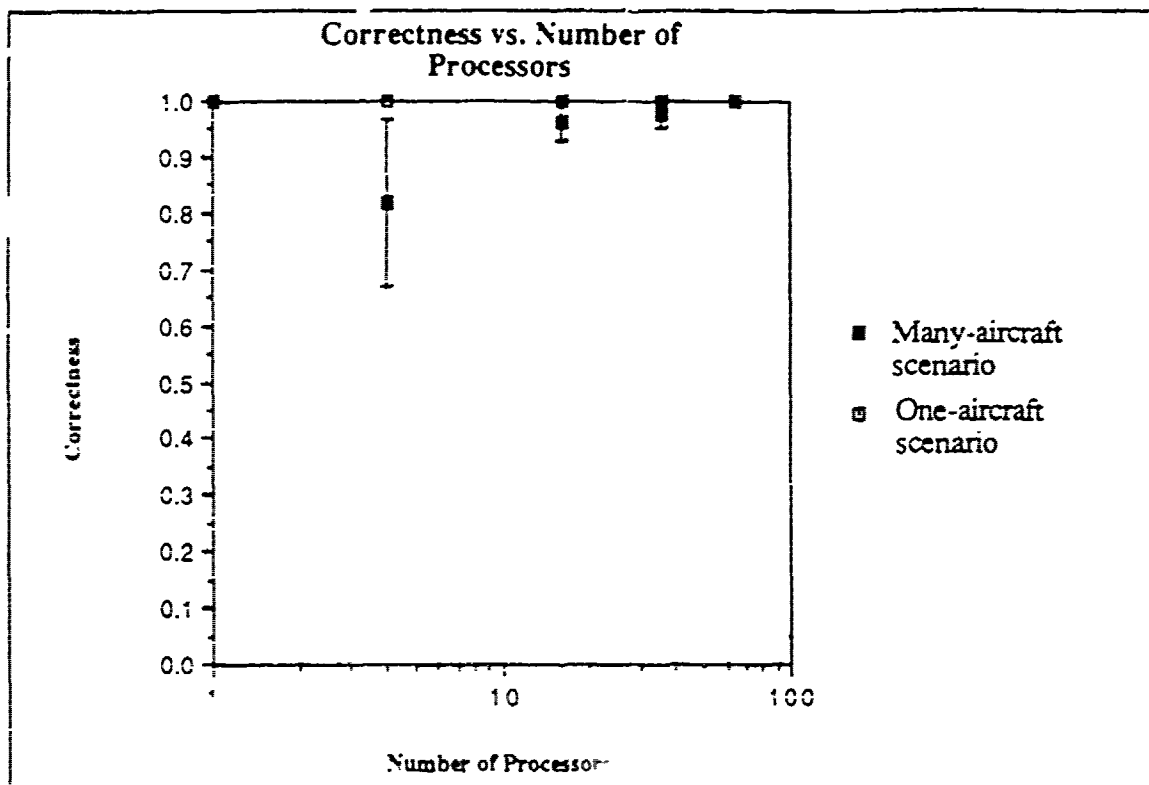


Figure 17. Correctness plotted as a function of the number of processors for the one-aircraft and many-aircraft scenarios.

7.4. Varying the input data set

The results from using the one-aircraft scenario highlight the difficulties in measuring performance of a real-time system where inputs arrive over an interval instead of in a batch. Before experimentation began, we hypothesized that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set. The results relative to this hypothesis are inconclusive. Figure 18 plots the confirmation latency against the number of processors for two input scenarios: the many-aircraft scenario (30 tracks per scan) and the one-aircraft scenario (1 track per scan).

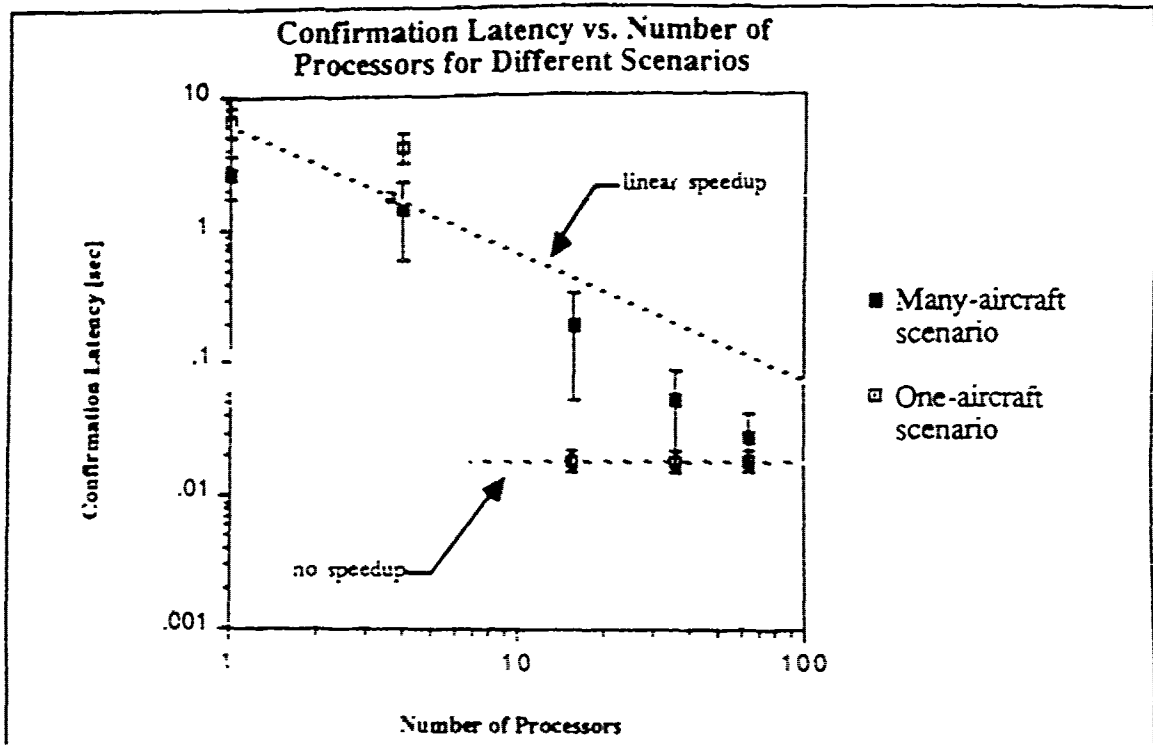


Figure 18. Confirmation latency as a function of the number of processors varies with the input scenario.

The one-aircraft scenario displays two distinct operating modes: one in which processor availability and waiting time determines the latency, and another in which data can be processed with little waiting.

The one-aircraft scenario displays interesting behavior: see Figure 18. While the confirmation latency decreases from the 1-processor to 4-processor case, just as in the many-aircraft scenario, there is distinctly different behavior for 16, 36 and 64 processor cases, where the average latency is constant over this range. The key to understanding this phenomenon is to realize that inputs to the system arrive periodically. The many-aircraft scenario generates approximately 800 reports comprising 70 radar tracks over a 200 millisecond duration. In contrast, the one-aircraft scenario generates approximately 1300 reports comprising 70 radar tracks over an 8 second duration. Thus, although the volume of reports is roughly equivalent (800 versus 1300), the duration over which they enter the system differs by a factor of 40 (0.2 seconds versus 8 seconds). In terms of radar tracks per second, which is a good measure of the object-creation workload, the many-aircraft scenario produces data at a rate of 350 tracks per second, while the one-aircraft scenario produces data at a rate of 8.8 tracks per second. This disparity causes the many-aircraft scenario to keep the system busy, while the one-aircraft scenario meters a comparable inflow of data over a much longer period, during which the system may become quiescent while it awaits additional inputs.

The one-aircraft scenario displays two distinct operating modes: one in which processor availability and waiting time determines the latency, and another in which data can be processed with little waiting. For the 1-processor and 4-processor cases, the system cannot process the input workload as fast as it enters, causing work to back up. This explains why the average confirmation latency for the 70 or so radar tracks is nearly as long as the scenario itself: most of the latency is consumed in tasks waiting to be executed. In

contrast, for the 16-processor, 36-processor and 64-processor cases, there are sufficient computing resources available to allow work to be handled as fast as it enters the system. This explains why the average latency bottoms out at 18 milliseconds, and also tends to explain the small variance.

Recalling that this particular experiment sought to test the hypothesis that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set, we see that these experimental results cannot confirm or disconfirm this hypothesis. The problem lies in the design of the one-aircraft input scenario. The reports should have been arranged to occur over the same 20 millisecond duration as in the many-aircraft scenario, instead of over an 8 second duration. Had that been done, the two scenarios would present to the system comparable workloads in terms of reports per second, but would differ internally in the degree to which sub-parts of the problem can be solved concurrently.

The distinction between the one-aircraft and many-aircraft scenarios can be described in Figure 19. This graph is an abstract representation of Figure 12 presented earlier, and plots the input workload as a function of time. The many-aircraft scenario presents a high input workload over a very short duration, while the one-aircraft scenario presents the same total workload spread out over a much longer interval. If we imagine the dashed lines to represent the workload threshold for which an n-processor system is able to keep up without causing waiting times to increase, we see that the many-aircraft scenario exceeded the ability of the system to keep up even at the 100-processor level, but the one-aircraft scenario caused the system to transition from not-able-to-keep-up to able-to-keep-up somewhere between 4 and 16 processors. A more appropriate one-aircraft scenario, then, is one that has the same input workload profile as the current many-aircraft scenario. Such a scenario would allow an experiment to be performed that fixes the input workload profile, which our experiment inadvertently varied, thereby contaminating its results.

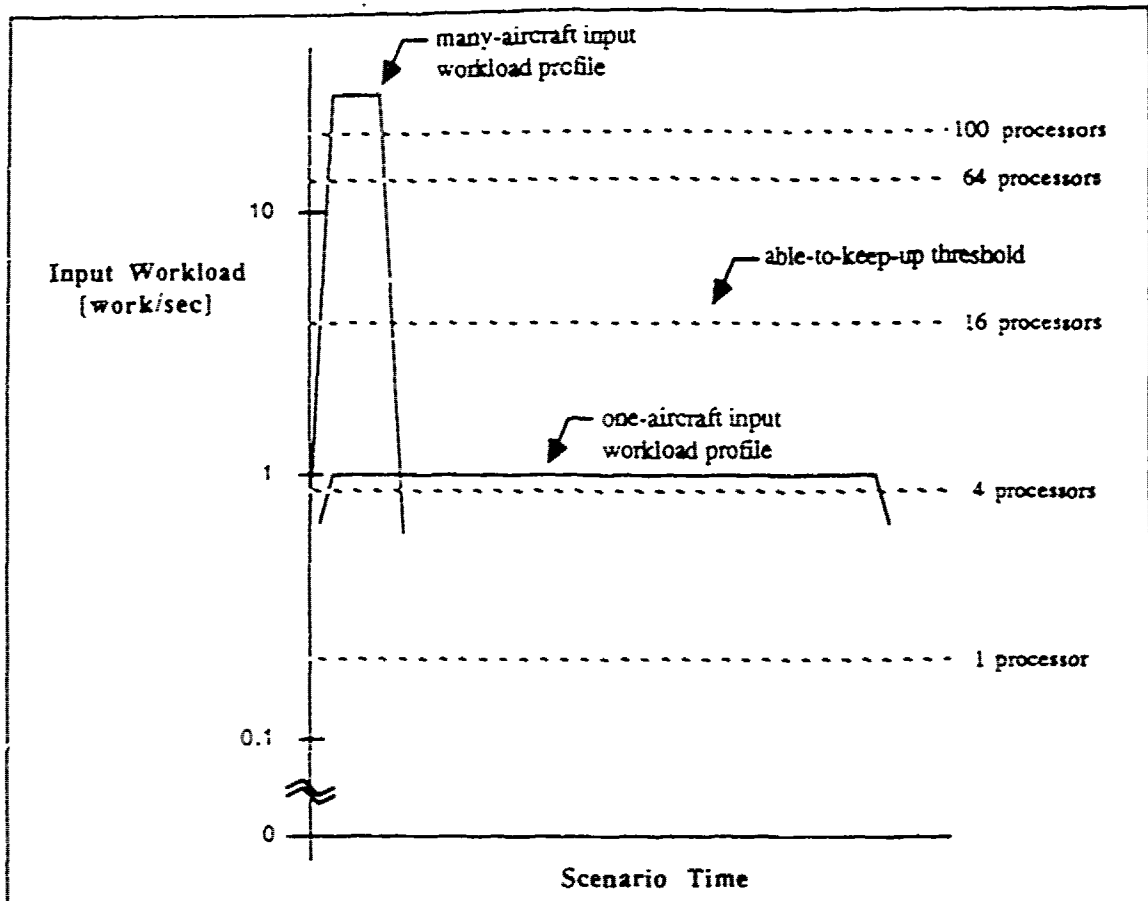


Figure 19. Input workload versus time profiles shown for two possible input scenarios.

The workload threshold above which the work becomes increasingly backlogged varies according to the number of processors.

8. Discussion

This section discusses how we achieved our experimental results using the concepts developed in Section 4. Specifically, we focus on the relationships between problem decomposition, speedup, and achievement of correctness.

8.1. Decomposition and correctness

In this section we analyze the problem solving knowledge embodied in the data association module. We use the dependence graph program to represent inherent dependencies in the problem. This is contrasted with the Lamina implementation to shed light on the rationale behind our design decisions. The goal is to identify the general principles that govern the transition from a dependence graph program to a runnable Lamina implementation.

8.1.1. Assigning functions to objects

We obtained speedup from both independent handling of tracks, and possibly from pipelining within a track, without the necessity to decompose the problem into the small functional pieces suggested in Figures 9 and 10. One might be tempted to believe that a direct translation of the nodes and edges of the dependence graphs into Lamina objects and methods might yield the maximal speedup, but careful study of the dependencies in Figures 9 and 10 reveals that there is very little concurrency to be gained.

In Figure 9, the entire graph is dependent on the arrival of report R_i . For instance, before a track is declared broken, the top-level "handle track" graph requires the arrival of reports $R_1, R_2, \dots, R_{last}$. The leftmost `add` node needs R_1 , and the remainder of the graph is dependent on this node. The `add` node to the right of this one is dependent on the arrival of R_2 , and the remaining right-hand subgraph is dependent on this node. This pattern holds for the entire graph, implying that computation may only proceed as far as consecutive reports beginning with R_1 have arrived. Thus, little concurrency may be gained from the "handle track" operation; in particular, no pipelining is possible because the entire graph receives only one set of reports, R_1, \dots, R_{last} . Figure 10 is similarly dependent on sequential processing of reports. We conclude that lumping all of the functions of Figures 9 and 10 into a small number of objects does not incur a great expense in concurrency. Given the overhead costs associated with message sending and process invocation, we speculate that one or two objects might yield the best possible design. In fact, our design uses $k+2$ objects, where k is the number of times a track is declared broken; k is typically over than three, giving us fewer than five objects for each "handle track" graph.

The dependence graph program provides several useful insights regarding a good problem decomposition. First, it justifies a decomposition that treats the "handle track" function as primitive function, rather than a finer-grained decomposition. Second, it clearly shows the independence between tracks, suggesting a relatively painless problem decomposition along these lines. Third, it shows the need to maintain consistent state about which tracks have been seen, and those which have not, suggesting a decomposition according to track id number, which is the approach that our Lamina program takes.

8.1.2. Why message order matters

A significant part of the Lamina concurrent program implements techniques to allow a Lamina object receiving messages from a single sender to handle them *as if* they were received in the order in which they were originally sent, without gaps in the message sequence. By doing this, we incur a performance cost because the receiver waits for arrival of the next appropriate message, rather than immediately handling whatever has been received.

The dependence graphs help to justify such costs because the dependencies imply ordering. Indeed, in preliminary work in a different framework, one author discovered that when no explicit ordering constraints were imposed during Airtrac data association processing, and neither additional heuristics nor knowledge was used, incorrect conclusions resulted in cases when the input data rate was high. The incorrect conclusions arose from performing the line-fit computation on other reports different from the first three consecutive reports. As such, the incorrectness reflected an interaction between message disordering arising in CARE and the particular Airtrac knowledge, rather than the specific problem solving framework. We believe, for instance, that similar incorrect conclusions would arise in a Lamina program that did not explicitly reorder reports.

We emphasize that although the particular problem that we studied showed strong correctness benefits from imposing a strict ordering of reports, this should not be interpreted as a claim that all problems need or require message ordering. As the dependence graphs make strikingly clear, the very knowledge that we implement dictates ordering. Another problem may not require ordering, but require a strict message tagging protocol, for instance. As a general approach, we believe that the programmer should represent the given problem in dependence graph form, preferably explicitly, to expose the required set of dependencies, and let the overall pattern of dependencies suggest the kinds of decompositions and consistency requirements that might prove best.

8.1.3. Reports as values rather than objects

In the dependence graph program we represent reports as values sent from node to node. Similarly, in the Lamina implementation, we use a design where reports are values sent from object to object. This works well because reports never change, enabling us to treat reports as values. The cost of allowing an object to obtain the value of a report is a fairly inexpensive one-way message, where value-passing is viewed as a monotonic transfer of a predicate. This approach works because we know ahead of time which objects need to read the value of a report, namely the objects that consume the processing pipeline.

Consider a second design where reports are represented as objects. In this scheme, instead of a report being a value passing through a processing pipeline, we arrange for read operations to be applied to an object. Conceptually these are identical problems, the only difference being the frame of reference. In the first case, the datum moves through processing stages requiring its value. In the case being considered here, the datum is stationary, and it responds to requests to read its value. This is attractive when it is not known in advance which objects will need to read its value. The penalty is an additional message required to request the object's value, and the associated message receipt system overhead.

A third design represents reports as objects, but replaces the read message in the previous design with a request to perform a computation, and uses the object's reply message to convey the result of the computation. By arranging a set of reports in a linear pipeline, we can allow the first report to send the results of its computation to the second report, and so forth. This design is the dual of the first design because in this design we send a sequence of computation messages through a pipeline of report objects, whereas in the first design we send a sequence of report value messages through a pipeline of computing objects. The designs differ in the grain-size of the problem decomposition: since our problem has a small number of computations and a large number of reports, the first design yields a small number of computing objects with many reports passing through, whereas the third design yields a large number of objects with a small number of computation messages passing through.

In our design, namely the first design discussed, we choose to represent reports as values sent to successive objects in a processing pipeline because our problem decomposition tells us in advance the objects in a pipeline. Using this design minimizes the number of messages required to accomplish our task, and uses a larger grain-size compared to its dual.

8.1.4. Initialization

Our approach to initialization embodies the correctness conditions of Schlichting and Schneider. Formally, we combine the use of monotonic predicates and predicate transfer with acknowledgement.

During initialization of our application, we create many objects, typically managers. At run-time, these objects communicate among themselves, which requires that we collect handles during creation, and distribute them after all creation is complete. Specifically, the Main Manager collects handles during the creation phase; in essence, each created object sends a monotonic predicate to the Main Manager asserting the value of its handle. The invariant condition may be expressed as follows:

Invariant (asserting own handle): "handle not sent" or "my handle is X"

The Main Manager detects the fact that all creation is complete when each of the predetermined number of objects respond; at this point, it distributes a table containing all the handles to each object. It waits until an acknowledgement is received from each object before initiating subsequent problem solving activity. This is important because if the Main Manager begins too soon, some object might not have the handle to another object that it needs to communicate with. In essence, the table of handles is asserted by a predicate transfer with acknowledgement. The invariant condition is described as follows:

Invariant (distributing table of handles):

"table not sent"

or "problem solving not initiated"

or "all acknowledgements received"

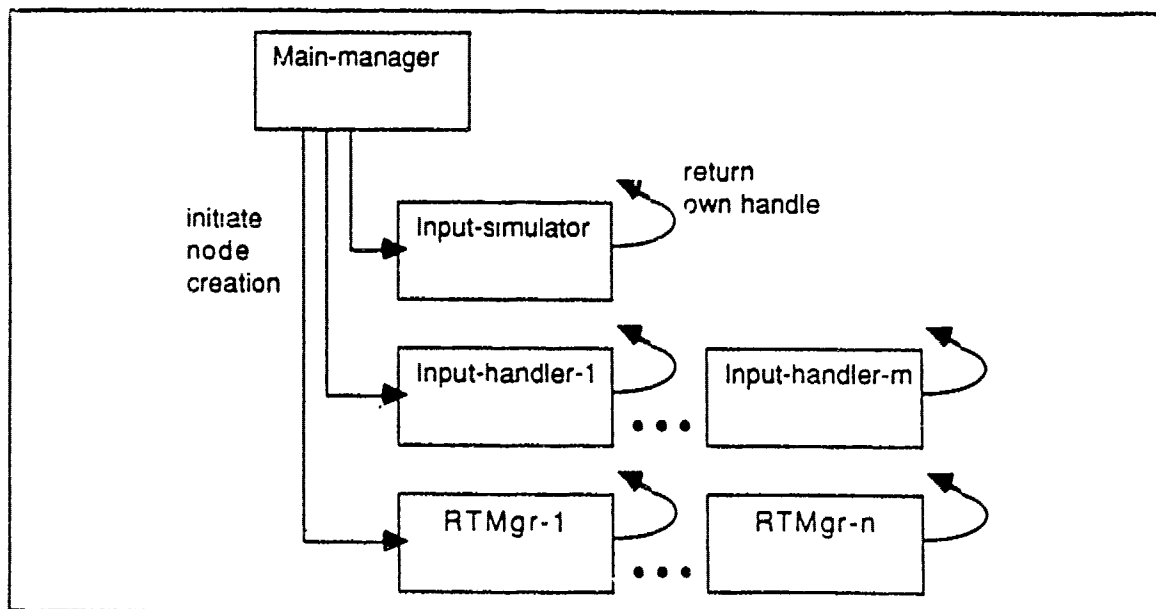


Figure 20. Creating static objects during initialization.

Correctness is crucial during initialization because a missing or incorrect handle, or a missing or improperly created object causes problems at run-time. These problems can compound themselves, causing performance or correctness degradation to propagate. By

using an initialization protocol that is guaranteed to be correct, these problems may be avoided.

8.2. Other issues

8.2.1. Load balance

We define *load balance* as how evenly the actual computational load is distributed over the processors in an array over time. Processing load is balanced when each processor has a mix of processes resident on it that makes all the processors equally busy. If a balanced processing cannot be achieved, the overall performance of a multiprocessor may not reflect the actual number of processors available to perform work due to poor load balance. In our experimentation, we discovered the critical importance of a good load balance algorithm.

We encountered two kinds of problems. The first problem deals with where to place a newly created object. Since we want to allocate objects to processors so as to evenly distribute the load, and because we want to avoid the message overhead associated with a centralized object/processor assignment facility, we focused on the class of algorithms that make object-to-processor assignments based on *local* information available to the processor creating the object. The second problem deals with how objects share limited processor resources. It turns out, for instance, that extremely computation-intensive objects can severely impair the performance of all other objects that share its processor.

At one point in our experimentation, for instance, we observed a disappointing value of unity for the $S_{64/16}$ speedup factor, where we instead expected a factor of 4. Moreover, we noticed an extremely uneven mapping of processes to processors: the approximately 200 objects created during the course of problem solving ended up crowded on only 14 of the 64 available processors! The culprit was the algorithm that decided which neighboring processor should be chosen to place a new object. The algorithm worked as follows. Beginning with the first object created by the system, a process-local data structure, called a *locale*, is created that essentially records how many objects are already located at every other processor in the processing array. When a new process is spawned, the locale data structure is consulted to choose a processor that has the fewest existing processes. This scheme works well when a single object creates all other objects in the system; unfortunately in Airtrac many objects may create new objects.

Given the locale for any given process, when the process spawns a new process, we arranged for the new process to inherit the locale of its parent. The idea is that we want the new process to "know" as much as its parent did about where objects are already placed in the array. This scheme fails because of the tree-like pattern of creations. Beginning with the initial manager object at the root of the tree, any given object has inherited a locale through all of its ancestors between itself and the root. Therefore the locale on a given object will only *know* about other objects that were created by the ancestors of the object *before* the locale was passed down to the next generation. Put another way, the locale on a given object will not reflect creations that were performed on non-ancestor objects, or creations that were performed on ancestor objects after the locale was passed down. This leads to extremely poor load balance.

The same problem occurs even if we define a single locale for each processor that is shared over all processes residing on that processor. Unfortunately, that locale will only know about other objects that were created by objects residing on that processor. That is,

the locale on a given processor will not reflect creations that were performed by objects that reside on *other* processors.

In contrast, ideal load balance occurs when each object knows about all creations that have taken place in the past over the entire processing array. This ideal is extremely difficult to achieve. First, we want to avoid using a single globally-shared data structure. Second, finite message sending time makes it impossible for many objects performing simultaneous object creation to access and update a globally-shared structure in a perfectly consistent manner.

We changed to a "random" load balance scheme which randomly selected a processor in the processing array on which to create a new object [Hailperin 87]. Running the base case on a 64 processor array with approximately 200 objects, we managed to use nearly all the available processors. Processor utilization improved dramatically.

Random processor allocation gave us good performance. In fact, we can argue from theoretical grounds that a random scheme is desirable. First, we deliberately constrain the technique to avoid using global information that would need to be shared. This immediately rules out any cooperative schemes that rely on sharing of information. Second, any scheme that attempts to use local information available from a given number of close neighbors and performs allocations locally faces the risk that some small neighborhood in the processing array might be heavily used, leaving entire sections of the array underutilized. We are left therefore, with the class of schemes that avoids use of shared information but allows any processor to select any other processor in the entire array. Given these constraints, a random scheme fits the criteria quite nicely and in fact performed reasonably well.

Further experimentation revealed more problems. Manager objects have a particularly high processing load because a very small number of objects (typically 5 to 9) handles the entire flow of data. When a non-manager object happens to reside on the same processor as a manager object, its performance suffers. For example, a Radar Track object is responsible for creating a Radar Track Segment object, and the time taken for the create operation affects the confirmation performance. Unfortunately, any Radar Track object that happens to be situated on the same processor as a manager object (e.g. Input Handler, Radar Track Manager) gets very little processor time, and thereby contributes significant creation times to the overall latency measure.

Whereas in the random scheme the probability that a given processor will be chosen for a new object is $\frac{1}{n}$ for n processors, our modified random scheme does the following:

- If there are fewer static objects (e.g. managers) than processors, then place static objects randomly, which can be thought of as sampling a random variable *without replacement*. Place dynamically created objects uniformly on the processors that have no static objects, this time sampling *with replacement*.
- If there are as many or more static objects than processors, then place roughly equal numbers of static objects on each processor in the array. Place dynamically created objects uniformly over the entire array, sampling *with replacement*.

This scheme keeps the high processing load associated with manager objects from degrading the performance of non-manager objects. This scheme performs well for our

cases. We typically had from 5 to 9 static objects, approximately 150 dynamic objects, and from 1 to 100 processors in the array.

There are other considerations that might lead to further improvement in load balance performance that we did not pursue. These are listed below:

- Account for the fact that not all static objects need a dedicated processor. (In our scheme, we gave each static object an entire processor to itself whenever possible.)
- Account for the fact that a processor that hosts one or more static objects may still be a desirable location for a dynamically created object, although less so than a processor without any static objects. (In our scheme, we assumed that any processor with a static object should be avoided if possible.)
- Relocate objects dynamically based on load information gathered at run-time.

8.2.2. Conclusion retraction

This section explores some of the thinking behind our approach toward consistency, which is to make conclusions (e.g. confirmation, inactivation) only when they were true. This is an extremely conservative stance, and possibly incurs a loss in concurrency and speedup. An alternative approach which might allow more concurrency is to make conclusions that are not provably correct: the programmer would allow such conclusions to be asserted, retracted and reasserted freely until a commitment regarding that conclusion is made. Jefferson has explored this computational paradigm, known as *virtual time* [Jefferson 85]. The invariant condition describing the truth value of a conclusion P under such a scheme is shown below:

Invariant: "no commitment made" or "P is true"

In essence, this invariant condition says that the program may *assert* that P is true, but there is no guarantee that P is true unless it is accompanied by a *commitment* to that fact. The benefits of such an approach is that assertions may precede their corresponding commitments by some time interval. This interval may be used 1) by the user of the system in some fashion, or 2) by the program itself to engage in further exploratory computation that may be beneficial, perhaps in reducing computation later. In Airtrac-Lamina, we did not investigate the benefits from exploratory computation.

For the user of the system, he or she must decide how and when to act upon uncommitted assertions rendered by the system. On one hand, the user could view assertions as true statements even before a commitment is made, with the anticipation that a retraction may be forthcoming. On the other hand, the user could view an assertion as true only when accompanied by a commitment; this latter approach places emphasis on the commitment, since only the commitment assures the truth of the conclusion.

We decided against using the scheme outlined here. As a technique to allow concurrent programs to engage in exploratory computations, there might be some merit if the power of such computations can be exploited. As a logical statement to the user of the system, such an uncommitted conclusion is meaningless, since it may later be retracted. As a probabilistic statement to the user of the system, a conclusion without commitment might indicate some likelihood that the conclusion is true. However, we believe that a better way to handle probabilistic knowledge is to state it directly in the problem rather than in the consistency conditions that characterize the solution technique. This unclear separation

between domain knowledge and concurrent programming techniques steered us away from the approach of making assertions with the possibility of subsequent retraction.

9. Summary

Lamina programming is shaped by the target machine architecture. Lamina is designed to run on a distributed-memory multiprocessor consisting of 10 to 1000 processors. Each processor is a computer with its own local memory and instruction stream. There is no global shared memory; all processes communicate by message passing. This target machine environment encourages a programming style that stresses performance gains through problem decomposition, which allows many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

For the programmer, Lamina provides a concurrent object-oriented programming model. Programming within Lamina has fundamental differences with respect to conventional systems:

- Concurrent processes may execute during both object creation and message sending.
- The time required to create an object is visible to the programmer.
- The time required to send a message is visible to the programmer.
- Messages may be received in a different order from which they were sent.

The many processes which must cooperate to accomplish the overall problem-solving goal may execute simultaneously. The programmer-visible time delays are significant within the Lamina paradigm because of the activities that may go on *during* these periods, and they exert a strong influence on the programming style.

This paper developed a set of concepts that allows us to understand and analyze the lessons that we learned in the design, implementation, and execution of a simulated real-time application. We confirmed the following experimental hypotheses:

- Performance of our concurrent program improves with additional processors; we attain significant levels of speedup.
- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

An inappropriate design of our one-aircraft scenario precluded us from confirming or disconfirming the following experimental hypothesis:

- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

In building a simulated real-time application in Lamina, we focused on improving performance of a data-driven problem drawn from the domain of real-time radar track understanding, where the concern is throughput. We learned how to recognize the

symptoms of throughput bottlenecks; our solution replicates objects and thereby improves throughput. We applied concepts of pipelining and replication to decompose our problem to obtain concurrency and speedup. We maintained a high level of correctness by applying concepts of consistency and mutual exclusion to analyze and implement the techniques of monotonic predicate and predicate transfer with acknowledgements. We recognized and repaired load balance problems, discovering in the process that a modified random processor selection scheme does fairly well.

The achievement of linear speedup up to 100 times that obtainable on a single processor serves as an important validation of our concepts and techniques. We hope that the concepts and techniques that we developed, as well as the lessons we learned through our experiments, will be useful to others working in the field of symbolic parallel processing.

Acknowledgements

We would like to thank all the members of the Advanced Architectures Project, who provided a supportive and stimulating research environment, especially to John Delaney who provided valuable guidance and support throughout this project, and to Bruce Delagi, Sayuri Nishimura, Nakul Saraiya, and Greg Byrd, who built and maintained the Lamina/CARE system. Max Hailperin provided the load balance routines, and also provided insightful criticisms. We would also like to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. Special gratitude goes to Edward Feigenbaum for his continued leadership and support of the Knowledge Systems Laboratory and the Advanced Architectures Project, which made it possible to do the reported research. This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and the Workstation Systems Engineering group of Digital Equipment Corporation.

References

- [Andrews 83] G.R. Andrews and Fred B. Schneider, Concepts and notations for concurrent programming *Computing Surveys* 15 (1) (March 1983) 3-43.
- [Arvind 83] Arvind, R.A. Iannucci, Two fundamental issues in multiprocessing: the data flow solution, Technical Report MIT/LCS/TM-241, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1983.
- [Brown 86]. H. Brown, E. Schoen, B. Delagi, An experiment in knowledge-based signal understanding using parallel architectures, Report No. STAN-CS-86-1136 (also numbered KSL 86-69), Department of Computer Science, Stanford University, 1986.
- [Broy 85] M. Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin, 1985).

- [Byrd 87]. G. Byrd, R. Nakano, B. Delagi, A dynamic, cut-through communications protocol with multicast. Technical Report KSL 87-44, Knowledge Systems Laboratory, Stanford University, 1987.
- [Cornañion 85] CORNAFION, *Distributed Computing Systems: Communication, Cooperation, Consistency* (Elsevier Science Publishers, Amsterdam, 1985).
- [Delagi 87a] B. Delagi, N. Saraiya, S. Nishimura, G Byrd, An instrumented architectural simulation system. Technical Report KSL 86-36, Knowledge Systems Laboratory, Stanford University, January 1987.
- [Delagi 87b] B. Delagi and N. Saraiya, Lamina: CARE applications interface, Technical Report KSL 86-57, Working Paper, Knowledge Systems Laboratory, Stanford University, May 1987.
- [Delagi 87c] B. Delagi, private communication, July 1987.
- [Dennis 85] J.B. Dennis, Data flow computation. Manfred Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin, 1985) 345-54.
- [Filman 84] R.E. Filman and D.P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software* (McGraw-Hill Book Co., New York, 1984).
- [Gajski 82] D.D. Gajski, D.A. Padua, D.J. Kuck, R.H. Kuhn, A second opinion on data flow machines and languages, *IEEE Computer* (February 1982) 58-69.
- [Hailperin 87] M. Hailperin, private communication, July 1987.
- [Henderson 80] P. Henderson, *Functional Programming* (Prentice-Hall International, Englewood Cliffs, 1980).
- [Jefferson 85] D.R. Jefferson, Virtual time, *ACM Transactions on Programming Languages and Systems* 7 (3) (July 1985) 404-25.
- [Kruskal 85] C.P. Kruskal, Performance bounds on parallel processors: an optimistic view, Manfred Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin, 1985) 331-44.
- [Kung 82] H.T. Kung, Why systolic architectures?. *IEEE Computer*. (January 1982) 37-46.
- [MacLennan 82] B.J. MacLennan, Values and objects in programming languages. *ACM Sigplan Notices* 17 (12) (December 1982).
- [Minami 87] M. Minami: Experiments with a knowledge-based system on a multiprocessor: preliminary Airtrac-Lamina quantitative results, Working Paper, Technical Report KSL 87-35, Knowledge Systems Laboratory, Stanford University, 1987.

- [Nakano 87] R. Nakano, Experiments with a knowledge-based system on a multiprocessor: preliminary Airtrac-Lamina qualitative results, Working Paper, Technical Report KSL 87-34, Knowledge Systems Laboratory, Stanford University, 1987.
- [Nii 86a] P. Nii, Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures, *Ai Magazine* 7 (2) (1986) 38-53.
- [Nii 86b] P. Nii, Blackboard systems part two: blackboard application systems, blackboard systems from a knowledge engineering perspective, *Ai Magazine* 7 (3) (1986) 82-106.
- [Schlichting 83] R.D. Schlichting and F.B. Schneider, Using message passing for distributed programming: proof rules and disciplines. Technical Report TR 82-491, Department of Computer Science, Cornell University, May 1983.
- [Smith 81] R.G. Smith, *A Framework for Distributed Problem Solving* (UMI Research Press, Ann Arbor, Michigan, 1981).
- [Tanenbaum 81] A. Tanenbaum, *Computer Networks* (Prentice Hall, Englewood Cliffs, New Jersey, 1981).
- [Weihl 85] W. Weihl and B. Liskov, Implementation of resilient atomic data types, *ACM Trans. on Programming Languages and Systems* 7 (2) (April 1985) 244-69.
- [Weinreb 80] D. Weinreb and D. Moon, Flavors: message passing in the Lisp machine, Technical Report, Memo 602, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1980.

CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving

by
H. Penny Nii

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

*To Appear in DARPA Conference on Expert Systems
Technology Base for Strategic Computing Program, Asilomar, April 1986.*

CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving¹

H. Penny Nii

Abstract

This paper is intended to serve as an introduction to two other papers: *User-Directed Control of Parallelism: The CAGE System* [1]; and *POLIGON: A System for Parallel Problem Solving* [6]. Two different skeletal systems, representing two models of concurrent problem solving, are described in those reports. Both systems are designed for parallel execution of application programs built with the systems. This paper describes the context in which these systems are being developed and summarizes the differences between the two systems.

The Context

The POLIGON and the CAGE systems are being developed within the context of two different families of experiments within the Advanced Architectures Project. Each family of experiments consists of a vertically integrated set of programs from each level of system hierarchy outlined in the project proposal (i.e. application, problem-solving framework, knowledge representation and retrieval, implementation language, and hardware/system architecture levels). POLIGON and CAGE are two systems at the problem-solving framework level. The design of both the POLIGON and the CAGE systems are based on the Blackboard problem solving model [5].

The Experiments

Each family of experiments starts with a different set of high-level constraints:

Hardware/system architecture: The POLIGON system is designed for distributed-memory, multi-processor systems. It assumes that the underlying system has a large number (100's to 1000's) of processor memory pairs with very high bandwidth inter-processor communication. The CAGE system, on the other hand, assumes a shared-memory, multi-processor system with tens to hundreds of processors. The underlying system architecture influences the additional constructs at the programming language level needed to support parallel executions. It also has significant affect on the design of blackboard frameworks.

¹This research was supported by DARPA RDC (F30602-85-C-0012), by NASA (NCC 2-220), and by Boeing Computer Services (W-266875).

Control of parallelism: The POLIGON system is designed with an assumption that the underlying problem solving framework on which the application is to be mounted must be intrinsically parallel. The POLIGON system is designed so that predefined constructs in the framework always run in parallel. For example, all rules are evaluated in parallel and all changes to blackboard nodes are made in parallel. The user has some ability to introduce serialization. CAGE, on the other hand, assumes that the user needs control over what is to run in parallel. Thus, everything in CAGE runs serially unless specified otherwise by the user. There are prespecified places where the user can introduce parallelism. For example, the user can specify that the condition parts of rules be evaluated in parallel and the action parts be executed in series.

The family of experiments of which CAGE is a part consists of CAGE (problem solving framework) implemented in Qlisp [3] (implementation language) running on a shared-memory architecture (system architecture) simulated on CARE [2] (system simulator). The other family of experiments consists of POLIGON (problem solving framework) implemented in CAOS [7] and Zetalisp (implementation language) running on a distributed-memory architecture (system architecture) simulated on CARE. Both CAGE and POLIGON run on the same system simulation program and share its software measurement tools. Both skeletal systems will mount the same application problems.

In keeping with the goals of our Project, the primary objective of the two families of experiments is to discover methods that would speed up the execution of knowledge-based application programs. There are, however, additional reasons for the two experiments that relate to the primary objective:

To compare the performance gains between shared versus distributed-memory, multiprocessor systems.

To provide input to the implementation language level (QLisp, CAOS and other concurrent Lisp languages);

To gain some understanding of the differences in 'programmability' between POLIGON and CAGE. More specifically, address the question of whether it is easier/better to let the user have complete control over the parallelism in a program; and as a corollary, to determine the limits of concurrency that can be designed into a framework, and the kinds of concurrencies that are problem specific and need to be expressed by the user.

To determine the extent of control, or serialization, needed in both systems in order to solve a class of problems, and to discover how to apply the needed control.

To determine if multiplicative speed-up can be effected between knowledge sources, rules, and lower level (for example, rule clause evaluation) concurrencies.

To determine what level of process granularity is most appropriate for each hardware/systems architecture.

Comparison of the CAGE and POLIGON Systems

CAGE and POLIGON are concurrent blackboard systems with two different underlying design philosophies. CAGE is an extension of the AGE [4] system with primitives to express parallel execution of knowledge sources, rules, and parts of rules. It is a conservative, incremental approach to building parallel systems. POLIGON is a demon-driven system in which all blackboard nodes are viewed as active agents (and thus each blackboard node can potentially be a processor/memory pair). A change made to a node causes appropriate rules to be evaluated and executed. POLIGON represents a shift in the way we view blackboard systems.

Both systems have programming languages associated with them, the POLIGON language and the CAGE language. The first objective in providing a language at the problem solving level is to facilitate the writing of application programs. This is accomplished by abstracting much of the system detail into language constructs. The second objective is to keep separate the parallelism in the application problem, as expressed by the language, and the parallelism built into the framework that remain invisible to the user. This separation allows us to experiment with parallelism in the application program independent of experiments with parallelism within the framework. Thus, we can for example, keep the application constant and change the parallel constructs within the framework, or keep the framework constant and rewrite the application. In order to facilitate the porting of an application program between POLIGON and CAGE, both languages are syntactically similar. However, the semantics of the languages are very different because the underlying systems are very different. The differences are summarized below.

<u>CAGE</u>	<u>POLIGON</u>
Incremental additions of parallelism to a serial system	Redesigned parallel system
User controlled parallelism	User controlled serial operations
Granularity of parallelism under user control	Granularity of parallelism fixed - rules and actions
Shared memory multi-processor machines	Distributed memory multi-processor machines

Figure 1: Summary of Differences: CAGE and POLIGON

We now describe and discuss some of the issues specific to the CAGE and the POLIGON systems. The discussions should serve as a background to the detailed description of the systems in the separate papers.

CAGE

There are several obvious places for concurrency in blackboard systems, the knowledge sources, rules within the knowledge sources, and the components of the rules.

Knowledge Source concurrency: Knowledge sources are logically independent partitions of domain knowledge. Each knowledge source is event-driven and becomes active when changes relevant to the knowledge source are made to the blackboard. Theoretically, therefore, all knowledge sources can be active at the same time as long as events relevant to each of the knowledge sources occur at the 'same time'. However, knowledge sources are often serially dependent in order to solve a problem. At run time some synchronization (i.e. serialization) must be enforced.

In the class of applications we are considering, the solution generation process characteristically occurs in a pipeline fashion up the blackboard hierarchy. That is, the knowledge source dependencies form a chain from the knowledge sources working on the most detailed level of the blackboard to those working on the most abstract level. When the program is model-driven, the pipeline works in the reverse direction. The task for CAGE in exploring concurrency at this level of granularity is to determine what percentage of the knowledge sources can be active at the same time in the pipe.

Rule concurrency: Each knowledge source is composed of many rules. The condition part of the rules are evaluated for a non-NIL condition (a match) and the action part of those rules that match are executed. The condition-part of all the rules in a knowledge source can be evaluated in parallel. In those cases where the action part of all the rules that match are to be executed, the action part can be executed as soon as the match is completed. However, if only one of the rules is to be fired (single-hit), then the system must wait until all the condition parts are evaluated, and one rule must be chosen whose action part will be executed. (Note that this is very similar to the OPS conflict-resolution phase.) In addition, one can imagine evaluating all of the condition parts in parallel and executing the appropriate action parts in series.

The situation in which all rules are evaluated and fired concurrently will result in the most speed-up, since many rules will be in the state of being evaluated and being executed at the same time. However, if the rules need access to the same blackboard item, memory contentions become a hidden point of serialization. At the same time, the integrity of information on the blackboard cannot be guaranteed. The condition which triggered the action part of the rule may not be the same by the time it is executed. CAGE needs to address these problems, determine the effect on solution quality and overall performance gain of the application program.

Condition-part concurrency: Each condition part of a rule consists of many clauses to be evaluated. These clauses can be computed in parallel. Often these clauses involve relatively

large numeric computation (e.g. calculating a track), making parallel clause evaluation worthwhile. On the other hand, often the clauses refer to the same data item, making the clause evaluation appear to be parallel, but in fact forcing serialization at the data-access level with no gain (and most likely a loss) in speed of computation. The task at this level of granularity is to determine if parallelism at this level is worthwhile. It may be that what is needed at this level is a fast algorithm for matching the condition parts and an appropriate knowledge representation scheme.

Action part concurrency: Often, when a condition part matches, there are many actions to be executed. This is one place where no difficulty is anticipated in parallel execution.

Combining the concurrencies: The action parts of rules generate events, and the knowledge sources are activated by occurrences of these events. In the AGE system events were posted on an event-list and a control monitor invoked the knowledge sources based on those events. In order to eliminate the serialization inherent in this control scheme, a mechanism to activate the knowledge source upon the completion of the action parts of rules is needed. The immediate activation of a knowledge source after action part execution (for example, by broadcasting an 'event message' to all the knowledge sources) results in the loss of global control over knowledge source activation. In some cases, this is acceptable. In other cases, for example when knowledge sources need to be activated on a priority basis (exemplified by the need for the Agenda mechanism in AGE), some control mechanism is needed. The task here is to determine the best (least overhead) control mechanism appropriate to the application.

POLIGON

As mentioned earlier, the application programs are event-driven in blackboard systems. Events are normally defined by the user and expressed as changes to the blackboard nodes. Because a knowledge source is activated by the occurrences of events, and because knowledge sources are collections of rules, one can view the rules as being activated (indirectly) by changes to some blackboard nodes. We can take this line of reasoning one step further and say that a rule is activated by changes to particular slots of blackboard nodes. If we associate a set of rules directly with a slot on a node and evaluate and execute the rules whenever the slot is changed, we have a system with active blackboard nodes.

Conceptually, at least, every blackboard node can be thought of as a processor-memory pair. Each node contains a data structure to store the partial solutions, and the rules are activated whenever a particular slot is changed. Slots with a property that enable rule triggering are called "trigger slots". When the action part of a rule is executed, the changes to the blackboard are made via messages to the nodes to be changed. If the change to is to a trigger slot, then the condition part of the "triggered rules" are evaluated; changes to non-trigger slots do not cause processing.

A major difficulty with this approach is the loss of control, specifically, an ability to control

the order of rule firing. By bypassing the intermediate control step where manipulation of the events and selection of knowledge sources occurs, the system has no global control. The rules will be firing almost indiscriminately all over the blackboard as solution state changes. There is no way to implement problem solving strategies, for example. In addition, rules will not be evaluated in situations when the *non-occurrence* of a change to the blackboard is significant. Such ability is important in signal interpretation programs.

In spite of many anticipated difficulties, we have developed a demon-driven system in hopes of gaining experience with such a system and discovering solutions to the problems. Although there is a substantial shift in the problem solving behavior, POLIGON is being evolved out of the functionalities that were present in AGE. At this point POLIGON is characterized by the following:

Knowledge sources exist only as a conceptual aid in partitioning the problem space.

Levels of in the blackboard data exist as a class hierarchy. A level is a class and a node is an instance of a class. There is also a super-class that knows about the classes. (For clarity, the class will be referred to a more familiar term, the level.)

All nodes are active entities.

Each rule must specify, in addition to the condition and action parts, the level and the node with which it is to be associated, i.e. it must designate a 'trigger'. A trigger consists of a slot name and a trigger-condition, which are to be interpreted as follows: whenever the value of the slot is changed, evaluate the trigger condition. If the trigger condition is non-nil then the rule becomes triggered. A triggered rule is put on a process queue for later evaluation.

The rules can use data futures, and for the time being all bindings are made through lazy evaluation. This means that all bindings are made only when needed. In addition, processing can continue while values are being fetched from other nodes.

The major control problem to be addressed in demon-systems is the serialization of demon activations. Potential for control in POLIGON exists in three places: (1) On the node, where action parts of the rules can be serialized, for example. (2) In the level manager, which knows about all the nodes on the level. (3) In the super-manger which knows about all the level managers. The level manager that can create and garbage collect the nodes, and knows which rules to attach to a newly created node. The level manager is the only agent that knows about all the existing nodes on its level. Thus, to send a message to all the nodes on a particular level, a message is sent to the level manager which forwards it to all its nodes.

In addition to the parallel evaluation of the condition parts of rules, the actions in the action part of the rules are executed in parallel.

Because of POLIGON's uncontrolled parallelism the solution to a problem will be indeterminate. That is, every execution of an application problem can potentially result in different answers. The challenge is to organize the knowledge in such a way that "acceptable" solutions are produced each time.

Most of the same concurrencies made available to the user in CAGE are built into the system

in POLIGON. The major challenge in POLIGON is the serialization of rule execution. For example, the ability to synchronize the execution of actions in CAGE has no counterpart in POLIGON. Since the system is demon-driven at the rule level, there are very few handles available to control the activation of rule evaluation.

Summary

CAGE and POLIGON thus are two very different approaches to the expression of parallelism at the problem solving framework level. As we develop and test applications using these frameworks, we expect to gain a more concrete understanding of their relative strength and weaknesses with respect to usability, application characteristics, and speedup. Each system is discussed in more detail in the papers by Aiello and Rice.

References

- [1] Nelleke Aiello.
User-Directed Control of Parallelism: The CAGE System.
Technical Report KSL Report 86-31, Knowledge Systems Laboratory, Computer Science
Department, Stanford University, April, 1986.
- [2] Bruce Delagi.
CARE Users Manual.
Technical Report KSL-86-36 (working paper), Knowledge Systems Laboratory, 1986.
- [3] Gabriel, R.P. and J. McCarthy.
Queue-Based Multi-Processing Lisp.
In *Proceedings of the 1984 Symposium on Lisp and Functional Programming.* August,
1984.
- [4] H. Penny Nii and Nelleke Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.
- [5] H. Penny Nii.
Blackboard Systems.
Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science
Department, Stanford University, April, 1986.
To appear in *AI Magazine*, Vol. 7, Nos. 2 and 3, 1986.
- [6] James Rice.
Poligon: A System for Parallel Problem Solving.
Technical Report KSL-86-19, Knowledge Systems Laboratory, Computer Science
Department, Stanford University, April, 1986.
- [7] Eric Schoen.
The CAOS System.
Technical Report KSL-86-22, Knowledge Systems Laboratory, Computer Science
Department, Stanford University, April, 1986.

Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon

by
H. Penny Nii, Nelleke Aiello and James Rice
(Nii, Aiello, Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Unisys under PC number V122189; Boeing Computer Services, under contract number W-266875.

Abstract

This paper describes the ways in which blackboard systems can be made to operate in a multi-processor environment. Cage and Poligon, two concurrent problem solving systems based on the blackboard model are described. The factors which motivate and constrain the design of parallel systems in general and parallel problem-solving systems in particular are described.

1. Background

A *Concurrent Problem Solving System* is a network of autonomous, or semi-autonomous, computing agents that solve a single problem. In building concurrent problem solvers, our objectives are twofold: (1) to evolve or invent models of problem solving in a multi-agent environment and (2) to gain significant performance improvement by the use of multi-processor machines. Within the community of researchers in artificial intelligence, there is an interest in understanding and building programs that exhibit cooperative problem-solving behavior among many intelligent agents, independent of computational costs (see [Corkill 83], [Lesser 83], [Smith 81] for some examples). *But*, one of the important pragmatics of using many computers in parallel is to gain computational speed-up.¹ Often, methods useful in a serial (single) problem solver in obtaining a valid solution and coherent problem-solving behavior, usually a centralized control, are not compatible with performance gain in a multi-agent environment. Cage and Poligon attempt to find a balance — to achieve adequate coherence with minimal global control *and* to gain performance with the use of multiple processors.

1.1. Problem Solving and Concurrency

Those problems that have been successfully solved in parallel, such as partial differential equations and finite element analysis, share common characteristics: they frequently used vectors and arrays; solutions to the problems are very regular, using well understood algorithms; and the computational demands, for example, for matrix inversion, are relatively easy to compute. In contrast, the class of applications we are addressing (and AI problems in general) are ill-structured or ill-defined. There is often more than one possible solution; paths to a solution cannot be predefined and must be dynamically generated and tried; generally data cannot be encoded in a regular manner as in arrays — the data structures are often graph structures that must be dynamically created, precluding static allocation and optimization. These differences indicate that to run problem solving programs in parallel, current techniques for parallel programs must be augmented or new ones invented. It is worth reviewing

¹Multiple computers are also used for other reasons besides speed-up — redundancy, mix of specialized hardware, need for physical separation, and so on.

some of the key points to be addressed in building concurrent, problem-solving programs.

1.1.1. Problem Solving Issues

Problem solving has traditionally meant a process of searching a tree of alternative solutions to a problem. Within each generate-and-test cycle, alternatives are generated at a node of a tree and promising alternatives selected for further processing. Knowledge is used to prune the tree of alternatives or to select promising paths through the tree. It is an axiom that the more knowledge there is the less generation and testing has to be done. In the extreme, many knowledge-based systems have large knowledge bases containing pieces of knowledge that recognize intermediate solutions and solution paths, thereby drastically reducing, or even eliminating, search. These two types of problem-solving techniques have been labeled *search* and *recognition* [McDermott 83]. In the search technique the majority of computing time is taken up in generating and testing alternative solutions; in the recognition technique the time is taken up in *matching*, a process of finding the right piece of knowledge to apply. Most applications use a combination of search and recognition techniques. A concurrent problem solving framework must be able to accommodate both styles of problem solving.

In serial systems meta-knowledge, or control knowledge, is often used to reduce computational costs. One common approach decomposes a problem into hierarchically organized sub-problems, and a control module selects an efficient order in which to solve these sub-problems. Closely related is the introduction of contextual information, or domain knowledge, to help in the recognition process. Both approaches enhance performance — reduce the number of alternatives to search or the amount of knowledge to match. In concurrent systems meta-knowledge and control modules become fan-in points, or hot-spots. A *hot-spot* is a physical location in the hardware where a shared resource is competed for, forcing an unintended serialization. Does this imply that problem solving systems that rely heavily on centralized control are doomed to failure in a concurrent environment? Can control be distributed? If so, to what extent? If more knowledge results in less search, can a similar trade-off be made between knowledge and control? In concurrent systems where control, especially global control, is a serializing process, can knowledge be brought to bear to alleviate the need for control?

1.1.2. Concurrency Issues

The biggest problem in concurrent processing was first described by Amdahl [Amdahl 67]. Simply stated, it is as follows: The length of time it takes to complete a run with parallel processes is the length of time it takes to run the longest process plus some overhead associated with running things in parallel. Take a problem that can be decomposed into a collection of independent sub-problems that can run concurrently, but which internally must run serially. If all of these components are run

concurrently, then the run-time for the whole problem will be equal to the run-time for the longest running component, plus any overhead needed to execute the sub-problems in parallel. Thus, if the longest process takes 10% of the total run time that the parallel processes would have taken if run end-to-end (serially), then the maximum speed-up possible is a factor of 10. Even if only one percent of the processing must be done sequentially this limits the maximum speed-up to one hundred, however hard one tries and however many processors are used. This is a very depressing result, since it means that many orders of magnitude of speed-up are only available in very special circumstances.

This raises the issue of *granularity*, the size of the components to be run in parallel. Amdahl's argument indicates the need for as small a granularity as possible. For example, is a rule a good candidate grain size for computation? On the other hand, if the process creation and process switching time is expensive, we want to do as much computation as possible once a process is running, that is, favor a larger granularity. In addition, in a multi-computer architecture a balance must be achieved between the load on the communication network and on the processors. It is often the case that as process granularity decreases, the processes become more tightly coupled — that is, there is a need for more communication between them. The communication cost is of course a function of the hardware-level architecture, including bandwidth, distance, topology, and so on. Finding an optimal grain size at the problem solving level is a multi-faceted problem.

Even if one is able to find an optimal granularity, there are forces that inhibit the processes from running arbitrarily fast in parallel. Some of the more common problems are:

- *Hot-Spots and Bottlenecks*: It is frequently the case that a piece of data must be shared. In any real machine multiple, simultaneous requests to access the same piece of data cause *memory contention*. The act of a number of processes competing for a shared resource — memory or processors — causes a degradation in performance. These processor and memory hot-spots cause bottlenecks in the processing of data: they restrict the flow of data and reduce parallelism.
- *Communications*: Multi-computer machines do not have a shared address space in which to have memory bottlenecks of the kind mentioned above. However the communications network over which the processing elements communicate still represents a shared resource which can be overloaded. It has a finite bandwidth. Similarly, multiple, asynchronous messages to a single processing element will cause that element to become a hot-spot.
- *Process Creation*: Execution of the sub-problems mentioned above require that they run as processes. The cost of the creation and management of such processes is non-trivial. There is a process

grain size at which it does not pay to run in parallel, because executing it sequentially is faster than executing it in parallel.

Having introduced some issues and constraints associated with parallelizing programs, we now introduce some other concepts that are important in writing concurrent programs, an understanding of which is useful to appreciate the discussions later in this paper fully.

- *Atomic operation*: This refers to a piece of code which is executed without interruption. In order to have consistent results (data) it is important to have well defined atomic operations. For instance, an update to a slot in a node might be defined to be atomic. Primitive atomic actions are usually defined at the system level.
- *Critical sections*: Critical sections are usually programmer-defined and refer to those parts of the program which are uninterruptible, that is, atomic. The term is usually used to describe large, complex operations that must be performed without interruption.
- *Synchronization*: This term is used to describe that event which brings asynchronous, parallel processes together synchronously. Synchronization primitives are used to enforce serialization.
- *Locks*: Locks are mechanisms for the implementation of critical sections. Under some computational models, a process that executes a critical section must acquire a lock. If another process has the lock, then it is required to wait until that lock is released.
- *Pipeline*: A pipeline is a series of distinct operations which can be executed in parallel but which are sequentially dependent; for instance, an automobile assembly line. The speed-up that can be gained from a pipeline is proportional to the number of stages, assuming that each stage takes the same amount of time, that is, if the pipe is "well balanced." Pipeline parallelism is a very important source of parallelism.

1.2. Background Motivation

In experiments conducted at CMU [Gupta 86], Gupta showed that applications written in OPS [Forgy 77] achieved speed-up in the range of eight to ten, the best case being about a factor of twenty. The experiments ran rules in parallel, with pipelining between the condition evaluation, conflict resolution, and action execution. The overhead for rule matching was reduced with the use of a parallelized Rete algorithm. (In programs written in OPS, roughly 90% of the time is spent in the match phase.) The speed-up factors seem to reflect the amount of relevant knowledge chunks (rules) available for processing a given problem solving state; this number appears to be rather small. Although the applications were not written specifically for a parallel architecture, the results are closely tied to the nature of the OPS system itself, which uses a monolithic and homogeneous

rule set and an unstructured working memory to represent problem solving states.

The premise underlying the design of Cage and Poligon is that this discouraging result could be overcome by dividing and conquering. It is hoped that by partitioning an application into loosely-coupled sub-problems (thus partitioning the rule set into many subsets of rules), and by keeping multiple states (for the different sub-problems), multiplicative speed-up, with respect to Gupta's experimental results, can be achieved. If, for example, a factor of seven speed-up could be achieved for each sub-problem, the simultaneous execution of rule sets could result in a speed-up of seven times the number of sub-problems. We are looking for methods that can provide at least a two orders-of-magnitude speed up. The challenge, of course, is to coordinate the resulting asynchronous, concurrent, problem solving processes toward a meaningful solution with minimal overheads.

1.3. The Blackboard Model and Concurrency

The foundation for most knowledge-based systems is the problem-solving framework in which an application is formulated. The problem-solving framework implements a computational model of problem solving and provides a language in which an application problem can be expressed. We begin with the blackboard model of problem solving [Nii 86], which is a problem-solving framework for partitioning problems into many loosely coupled sub-problems. Both Cage and Poligon have their roots in the blackboard model of problem solving. The blackboard framework seems, at first glance, to admit the natural exploitation of concurrency. Some of the possible parallelism that can be exploited are:

- knowledge parallelism — the knowledge sources and rules within each knowledge source can run concurrently;
- pipeline parallelism — transfer of information from one level to another allows pipelining; and
- data parallelism — the blackboard can be partitioned into solution components that can be operated on concurrently.

In addition, the dynamic and flexible control structure can be extended to control parallelism.

These characteristics of blackboard systems have prompted investigators, for example Lesser and Corkill [Lesser 83] and Ensor and Gabbe [Ensor 85], to build distributed and/or parallel blackboard systems. The study of parallelism in blackboard systems goes back to Hearsay-II [Fennell 77].

The blackboard problem-solving metaphor itself is very simple; it entails a collection of intelligent agents gathered around a blackboard, looking at pieces of information written on it, thinking about them and writing their conclusions up as they come to them. This is shown in Figure 1.

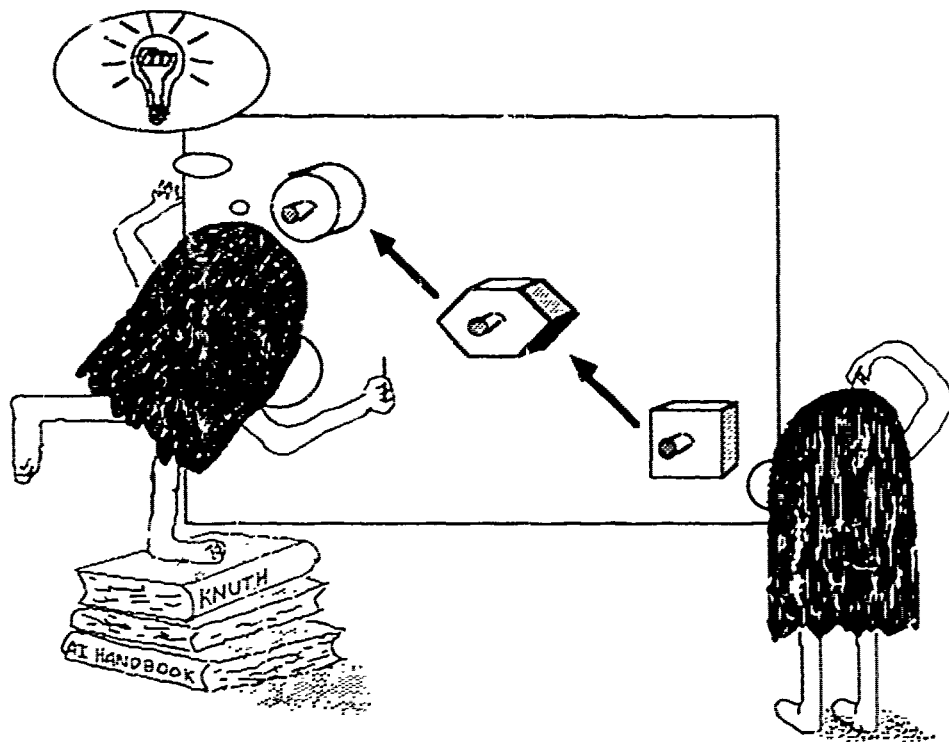


Fig. 1. The Blackboard Metaphor.

There are some assumptions made in this model that are so obvious that they might be missed. An understanding of the implications of these assumptions is vital to an understanding of the problem of achieving parallelism in blackboard systems.

- All of the agents can see all of the blackboard all of the time, and what they see represents the current state of the solution.
- Any agent can write his conclusions on the blackboard at any time, without getting in anyone else's way.
- The act of an agent writing on the blackboard will not confuse any of the other agents as they work.

The implications of these assumptions are that a single problem is being solved asynchronously and in parallel. However, the problem solving behavior, if it were to be emulated in a computer, would result in very inefficient computation. For example, for every agent to "see" everything would entail stopping everything until every agent has looked at everything.

Existing, serial blackboard systems make a number of modifications to the pure blackboard metaphor in order to make a reasonable implementation on conventional hardware. In effect, they modify the blackboard metaphor so that it *cannot* be executed in parallel. Some of these modifications are shown in Figure 2 and are described below.

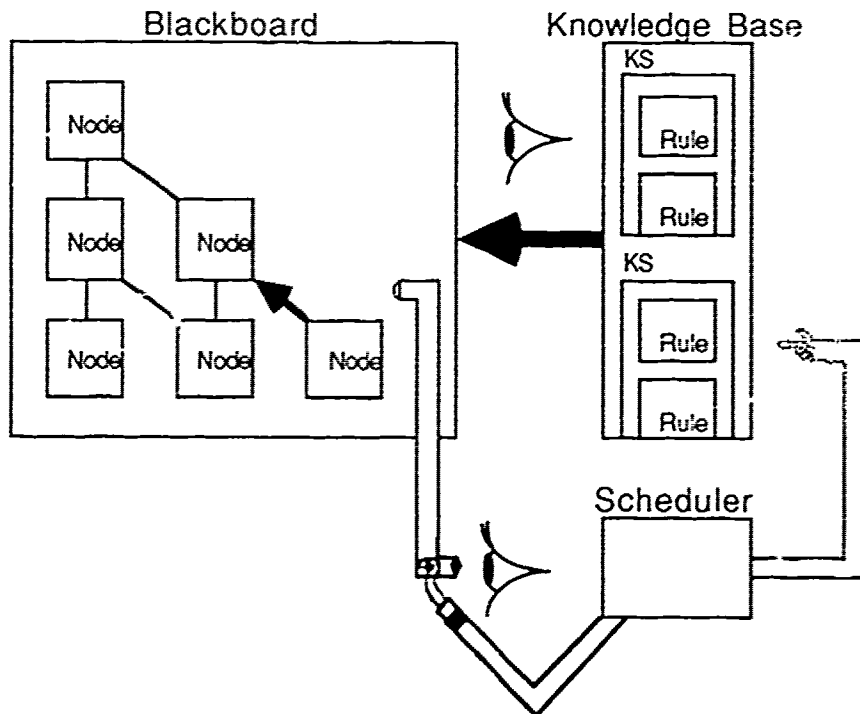


Fig. 2. The Serial Blackboard System.

- Agents are represented as *knowledge sources*. These knowledge sources are schedulable entities and only one can be running at any time. It will be shown later that one of the possible sizes for computational grains is the knowledge source.
- To coordinate the execution of knowledge sources, a scheduling or control mechanism is implemented. This is, in many ways, an efficiency gaining mechanism, which uses control knowledge to select only the most "valuable" knowledge source at any given moment to work on the problem.
- The blackboard is not truly "globally visible" in the sense prescribed by the blackboard metaphor. Instead, the blackboard is implemented as a data structure, which is sufficiently interconnected that it is possible for a knowledge source to find its way from one data item to a related one easily. Knowledge sources can only work on a limited area of the blackboard — knowledge sources and their context of invocation are, in fact, treated as self-contained subproblems.
- An implicit assumption is made that a knowledge source operates within a valid, or consistent, context and that the "ordered" execution of knowledge sources, even when the ordering is done dynamically, preserve the consistency of the blackboard data.

Trying to parallelize serial blackboard systems characterized above directly has certain limitations. First, only a modest speed-up can be achieved by a central scheduler determining the knowledge sources to be run in parallel.

The performance levels off very quickly at a very low number (a gain of less than a factor of three in our experiments no matter how many knowledge sources are run in parallel and no matter how many processors are used. Second, one of the most difficult problems in parallel computation is to maintain consistent data values. In concurrent blackboard systems, the data consistency problems occur in three different contexts: (1) on the entire blackboard, maintaining consistent solution states; (2) in the contents of the nodes, assuring that all slot values are from the same problem solving state; and (3) in the slots, keeping the value being evaluated from changing before the evaluation is completed.

2. The Advanced Architectures Project

Cage [Aiello 86] and Poligon [Rice 86], two frameworks for concurrent problem solving, are being developed within the Advanced Architectures Project (AAP) at the Knowledge Systems Laboratory of Stanford University. The objective of the AAP is the development of broad system architectures that exploit parallelism at different levels of a system's hierarchical construction. To exploit concurrency one must begin by looking for parallelism at the application level and be able to formulate, express, and utilize that parallelism within a problem-solving framework, which, in turn, must be supported by an appropriate language and software/hardware system. The system levels chosen and some issues for study are:

- Application level: How can concurrency be recognized and exploited?
- Problem solving level: Is there a need for a new problem-solving metaphor to deal with concurrency? What is the best process and data granularity? What is the trade-off between knowledge and control?
- Programming language level: What is the best process and data granularity at this level? What are the implications of choices at the language level for the hardware and system architecture?
- System hardware level: Should the address spaces be common or disjoint? What should the processor and memory characteristics and granularity be? What is the best communication topology and mechanisms? What should the memory-processor organization be?

At each system level one or more specific methods and approaches have been implemented in an attempt to address the problems at that level. These programs are then vertically integrated to form a family of experimental systems — an application is implemented using a problem-solving framework using a particular knowledge representation and retrieval method, all of which use a specific programming language, which in turn runs on a specific system/hardware architecture simulated in detail on the Lisp-based CARE simulator [Delagi 86]. Each family of experiments is designed to evaluate, for example, the system's

performance with respect to the number of processors, the effects of different computational granularity on the quality of solution and on execution speed-up, ease of programming, and so on. The results of one such family of experiments have been reported by Brown and Schoen [Brown 86, Schoen 86].

Within the context of this AAP organization, Cage and Poligon are two systems that are implemented to study the problem-solving level. Both Cage and Poligon use frames and condition-action rules to represent knowledge. The target system architecture for Cage is a shared-memory multi-processor; the target architecture for Poligon is a distributed-memory multi-processor, or multi-computer.

Both Cage and Poligon aim to solve a particular, but broad, class of applications: real-time interpretation of continuous streams of errorful data, using many diverse sources of knowledge. Each source of knowledge contributes pieces of a solution which are integrated into a meaningful description of the situation. Applications in this class include a variety of signal understanding, information fusion, and situation assessment problems. The utility of blackboard formulations has been successfully demonstrated by programs written to solve problems in our target application class [Brown 82, Mccune 83, Nii 82, Shafer 86, Spain 83, Williams 84].

Most of the systems in this class use the recognition style of problem solving with knowledge bases of facts and heuristics; numerical algorithms are also included as a part of the knowledge. Some search methods are employed but are generally confined to a few of the sub-problems.

In designing a concurrent blackboard system for the AAP, two distinct approaches seemed possible — one, to extend a serial blackboard system, and the other, to devise a new architecture to exploit the event-driven nature of blackboard systems. Each has its own problems and its own advantages, which will be described in the following sections.

3. Extending the Serial System - Cage

Cage is a concurrent blackboard framework system, based on the (serial) AGE [Nii 79] blackboard system. AGE uses a set of rules as a representation for its knowledge sources; it uses a set of event tokens as preconditions (a trigger) for the knowledge sources, and each significant change to the blackboard posts an event in a global data structure. The controller selects an event and executes a knowledge source whose precondition matches the selected event.¹ In addition to the basic functionality found in AGE, Cage allows user-directed control over the concurrent execution of many of its constructs (see Figure 3). Otherwise, the two systems are functionally identical.

¹There are more elaborate constructs in AGE, but this description suffices for the current purpose.

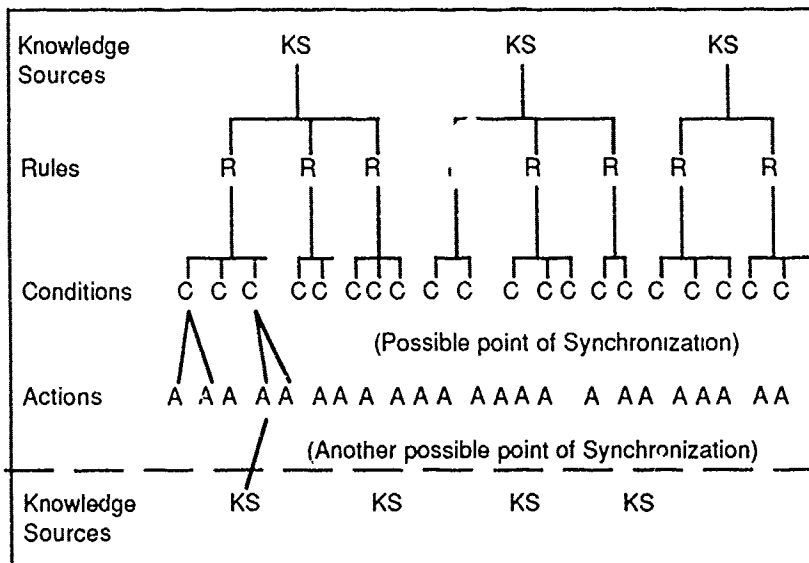


Fig. 3. Parallel Components of Cage

3.1. The Cage Architecture

The basic components of a system built with Cage are:

- A global data store (the blackboard) on which emerging solutions are posted. Objects on the blackboard are organized into hierarchical levels, and each object is described with a set of attribute-value pairs.
- Globally accessible lists on which control information is posted (for example, lists of events, expectations, and so on).
- An arbitrary number of knowledge sources, each consisting of an arbitrary number of rules.
- Control information that can help to determine (1) which blackboard elements are to be the focus of attention and (2) which knowledge sources are to be used at any given point in the problem solving process.
- Declarations that specify which components are to be executed in parallel (knowledge sources, rules, condition and action parts of rules), and at what points synchronization is to occur.

The user can run Cage serially (at which point Cage behavior is identical to that of AGE), or can run with one or more of the components running concurrently. In the serial mode, the basic control cycle begins with the selection and execution of a knowledge source. A resulting change to the blackboard may cause several knowledge sources to become relevant and candidates for execution. Cage uses a global list structure to record the changes to the blackboard, called events. The controller selects one of the events. The user can specify how the event is to be selected, such as FIFO, LIFO, or any user defined best-first method. The event in focus is then

matched against the knowledge source preconditions. The knowledge sources, whose preconditions match the focus events, are then executed in some predetermined order. The rules within each knowledge source are evaluated, and the action part of the rule is executed for those rules whose condition parts are satisfied. The user may choose to allow only one rule to fire per knowledge source activation or many rules to fire. Each action part may cause one or more changes on the blackboard and a corresponding number of events is recorded on the event list. Figure 4 shows the serial Cage control cycle.

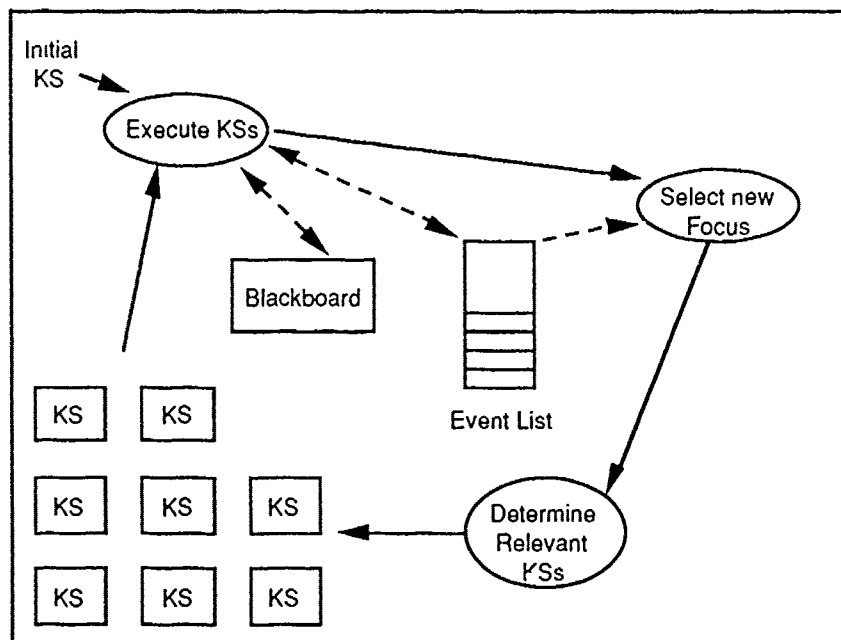


Fig. 4. Cage Serial Control Cycle

Using the concurrency control specifications, the user can alter the simple, serial control loop of Cage by requesting the concurrent execution of application components. Cage allows for a range of granularity for these concurrent processes; from knowledge sources all the way down to predicates in the condition parts of rules. The various concurrency operations that can be specified, together with the serial version, are summarized below and shown in Figure 4.

Knowledge Source Control

Serial:

Pick an event and execute the associated knowledge sources.

Parallel:

1. As each event is generated execute the associated knowledge sources in parallel, OR
2. Wait until all active knowledge sources complete execution, generating a number of events, and then execute the knowledge sources relevant to those events concurrently, OR

3. Wait until several events are generated then select a subset and execute the relevant knowledge sources for all the subset events in parallel.

Within Each Knowledge Source

Serial:

1. Perform context evaluation.
2. Evaluate the condition parts, then execute the action part of one rule whose condition side matched, OR
3. Evaluate all the condition parts then execute all the actions of those rules whose condition side matched, serially.

Parallel:

1. Perform context evaluation in parallel.
2. Evaluate all condition parts in parallel, then
 - a. synchronize (that is, wait for all the condition side evaluations to complete) and choose one action part, OR
 - b. synchronize and execute the actions serially (in lexical order), OR
 - c. execute the actions in parallel as the condition parts match.

Within Rules

Serial:

Evaluate each clause then execute each action.

Parallel:

Evaluate the condition-part clauses in parallel then execute the actions of the action part in parallel.

3.2. Discussion of the Concurrent Components

Each of the potential concurrent components are discussed below.

3.2.1. Knowledge Source concurrency:

Knowledge sources are logically independent partitions of the domain knowledge. A knowledge source is selected and executed when changes made to the blackboard are relevant to that knowledge source. Theoretically, many different knowledge sources can be executed at the same time as long as the relevant blackboard changes occur close to each other. But, the knowledge sources are often serially dependent and some synchronization must be introduced.

In the class of applications under consideration, the solution is built up in a pipeline-like fashion up the blackboard hierarchy. That is, the knowledge source dependencies form a chain from the knowledge sources working on the most detailed level of the blackboard to those working on the most abstract level. (When the program is model-driven, this pipeline works in the reverse direction.) Knowledge sources can be running in parallel, processing the data along the pipe.

Thus, there are two potential sources of knowledge source parallelism: (1) knowledge sources working on different regions (partial solutions) of the blackboard asynchronously, that is, "data parallelism," and (2) knowledge sources working in a pipelined fashion exploiting the flow of information up (or down) the data hierarchy.

3.5.2. Rule cone concurrency:

Each knowledge source is composed of a number of rules. The condition parts of these rules are evaluated for a match with the current state of the solution, and the action parts of those rules that match the state are executed. The condition parts of all the rules in a knowledge source, being side-effect-free, can be evaluated concurrently. In cases where all the matched rules are to be executed, the action parts can be executed as soon as the condition part is matched successfully. If only one of the rules is to be selected for execution, the system must wait until all the condition parts are evaluated, and one rule, whose action part is to be executed, must be chosen.¹ The situation in which all rules are evaluated and executed concurrently potentially has the most parallelism. However, if the rules access the same blackboard data item, memory contention becomes a hidden point of serialization. At the same time, the integrity of the information on the blackboard cannot be guaranteed. The problem is of two types: timeliness and consistency. First, the state which triggered the rule may be modified by the time the action part is executed. The question is then; is the action still relevant and correct? Second, if a rule accesses attributes from different blackboard objects, there is no guarantee that the values from the objects are consistent with respect to each other.

Condition-part concurrency: Each condition part of a rule may consist of a number of clauses to be evaluated. These clauses can often be evaluated concurrently. In the chosen class of applications, these clauses frequently involve relatively large numeric computations, making parallel evaluation worthwhile. However, as discussed above, if the clauses refer to the same data item, memory contention would force a serialization.

Action-part concurrency: Often, when a condition part matches, more than one potentially independent action is called for, and these can often be executed in parallel.

This problem of data consistency occurs both in Cage and in Poligon. It can be partially alleviated by defining an atomic operation that includes both read and write. This ensures that between the time that an item of data is read, processed, and the result stored, there is no change in the state of the node.² However, this makes a commitment to a certain level of granularity, for example, read the data for the condition part of a rule and execute the

¹Note that this is very similar to the OPS conflict-resolution phase. Refer to [Gupta 86] for the results of running OPS rules in parallel.

²In Lamina [Delagi 86], another programming framework developed for the AAP project, the atomic action is read-process-write.

rule. In order to enable experimentation with granularity, atomic actions are kept small and locks, block reads, and block writes are provided in Cage. Although an atomic read/write operation does not solve the problems of timeliness or of global coherence, it does assure that the data within the nodes are consistent. And, although locks have a potential for causing deadlocks, they are provided for the user to construct larger critical sections.

3.2.3. Concurrency Control

The action parts of rules generate events, and knowledge sources are activated by the occurrences of these events. In the (serial) AGE system events are posted on a global event list and, working on these events, a control monitor activates one or more knowledge sources. In order to eliminate the serialization inherent in this control scheme, a mechanism to activate the knowledge source immediately upon event generation is needed. This immediate activation of knowledge sources bypasses the control module and effectively eliminates global control. In some cases, this is acceptable. In other cases where knowledge sources are serially dependent, some control mechanism is needed. Centralized control mechanisms, such as selecting many events to be processed in parallel, causing many knowledge sources to run concurrently, are also provided.

Some answers to the many questions raised about Cage's architecture are embedded in the system. However, much of the burden is passed on to the application programmer. Some useful programming techniques that were discovered are discussed below.

3.3. Programming with Cage

There are a number of problems that crop up during concurrent execution that do not appear during serial execution. The solutions to some of these problems involved reformulating the application problem; some involved the use of programming techniques not commonly used in serial systems. Both Cage and Poligon have been used to implement a signal understanding system called Elint [Brown 86]. It is described briefly below.

3.3.1. The Elint Application

The problem is one of receiving multiple streams of reports from radar systems, abstracting these into hypothetical radar emitting aircrafts and tracking them as they travel through the monitored airspace. These aircraft are themselves abstracted into clusters — perhaps formations — which are themselves tracked. Sometimes an aircraft in a cluster would split off, forcing the splitting of the cluster node and rationalization of the supporting evidence. The nature of the radar emissions from the aircraft(s) are interpreted in order to determine the intentions and degree of threat of each of the clusters of emitters.

The Elint application has a number of characteristics which are of significance.

- The system must be able to deal with a continuous data stream. It is not acceptable to wait until all of the data has been read in and then figure out what was going on.
- The application domain is potentially very data parallel. The ability to reason about a large number of aircraft simultaneously is very important.
- The aircraft themselves, as objects in the solution space, are quite loosely coupled.

3.3.2. Pitfalls, Problems and Solutions

The following programming techniques arose while implementing Elint in Cage.

1. When the computational grain size is limited to a knowledge source, it is possible to read all the slots of a node that are referenced in the knowledge source by locking the node once and reading all of the slots at once. This is in contrast to locking the node every time a slot is read by the rules. This is equivalent to reading all of the blackboard data accessed from a knowledge source before any rules are evaluated. This approach accomplishes two important things: (1) It reduces the number of references to the blackboard, thereby reducing the opportunities for memory contention, and (2) it ensures that all the rules are looking at data from the same point in the evolving solution.

2. In a serial blackboard system one precondition may serve to describe several changes to the blackboard adequately. For example, suppose one rule firing causes three changes to be made serially. The last change, or event, is generally a sufficient precondition for the selection of the next knowledge source. In a concurrent system, all three events must be included in a knowledge source's precondition. This is to ensure that all three changes have actually occurred before the knowledge source is executed.

In general, a simple precondition consisting of an event token is not sufficient for Cage. Either a sophisticated scheduler with detailed specification of the activation requirements of the knowledge sources, or a complex, knowledge-source precondition that contain the same requirements is needed.

3. It is important when writing the conditions of rules for a Cage application to keep in mind the feasibility of running the condition clauses concurrently, that is, keeping them independent of each other in the sense of not accessing the same data.

4. Occasionally two knowledge sources running in parallel may attempt to change a slot at almost the same time. It is possible that the first change would invalidate the firing of the second rule. To overcome this type of race condition, a conditional action — an action which checks the value of a slot before making a change — was added. It allows the action to check the most recent updates before making further changes. The alternative would have been to lock a node for an entire knowledge source execution which would seriously limit parallelism.

3.3.3. A Problem with Continuous Input Streams

Since Elint is a real-time system, it is time dependent. Processing a continuous stream of data can lead to *out-of-order events* caused by delay of one kind or another; an example might be a knowledge source stuck in a memory queue delaying its changes to the blackboard. This means that new data at time t may have to be analyzed before all the ramifications of data from an earlier time ($t - n$) have been executed — at any point the data can be out of order. The Elint application had to be reformulated to address this problem. Time tags had to be associated with each event and blackboard value, and the rules had to be re-written to use the time tags to reason about unordered events.

3.3.4. Incremental Introduction of Parallelism

Experiments with Cage indicate that it is much more difficult to program a parallel system than a serial one. It lends subjective support to our supposition that an incremental approach to parallelism is easier to program than an all-at-once approach. We began with a serial version of Elint and turned on clause level concurrency first and debugged it, then experimented with rule level, and finally knowledge source level concurrency. Only after Elint was working correctly with each of these concurrent operations, were they combined.

As discussed earlier, Cage can execute multiple sets of rules, in the form of knowledge sources, concurrently. If the rule parallelism within each knowledge source can provide a speed-up in the neighborhood cited by Gupta, and if many knowledge sources can run concurrently without getting in each other's way, we can hope to get a speed up in the tens. The extra parallelism comes from working on many parts of the blackboard, in other words, by solving many sub-problems in parallel. It was found, however, that the use of a central controller to determine which knowledge sources to run in parallel drastically limits speed-up, no matter how many knowledge sources are executed in parallel. Amdahl's limit and synchronization come strongly into play. The implication for Cage is that knowledge-source invocation should be distributed, without synchronization. This will eliminate two major bottlenecks — a data-hot spot at the event list, and waiting for the slowest process to finish during synchronization. Still, within a shared-memory, multi-processor system, the interface to the blackboard is a bottleneck. One solution to this is to

distribute the blackboard, which is one of the main characteristics of Poligon.

4. Pursuing a Daemon-driven Blackboard System — Poligon

Control in the blackboard model could be summarized as follows: *knowledge sources respond opportunistically to changes in the blackboard.* As discussed earlier, in reality, and especially in serial systems, the blackboard changes are recorded and a control module decides which change to pursue next. In other words, the knowledge sources do not respond directly to changes on the blackboard. A control module generally dictates the problem-solving behavior. This is a serializing process.

The basic question that led to the design of Poligon is: *What if we attach the knowledge sources to the data elements in the blackboard which, when changed, would result in the activation of those knowledge source?* Instead of waiting until a control module activates a knowledge source, why not immediately execute the knowledge source as the relevant data are changed, and get rid of the control module? A blackboard change would serve as a direct trigger for knowledge source activations. Next, assign a processor-memory pair for each blackboard node, and have the knowledge sources (now on the blackboard processing element) communicate changes to other nodes by passing messages via a communication network. (see Figure 5)

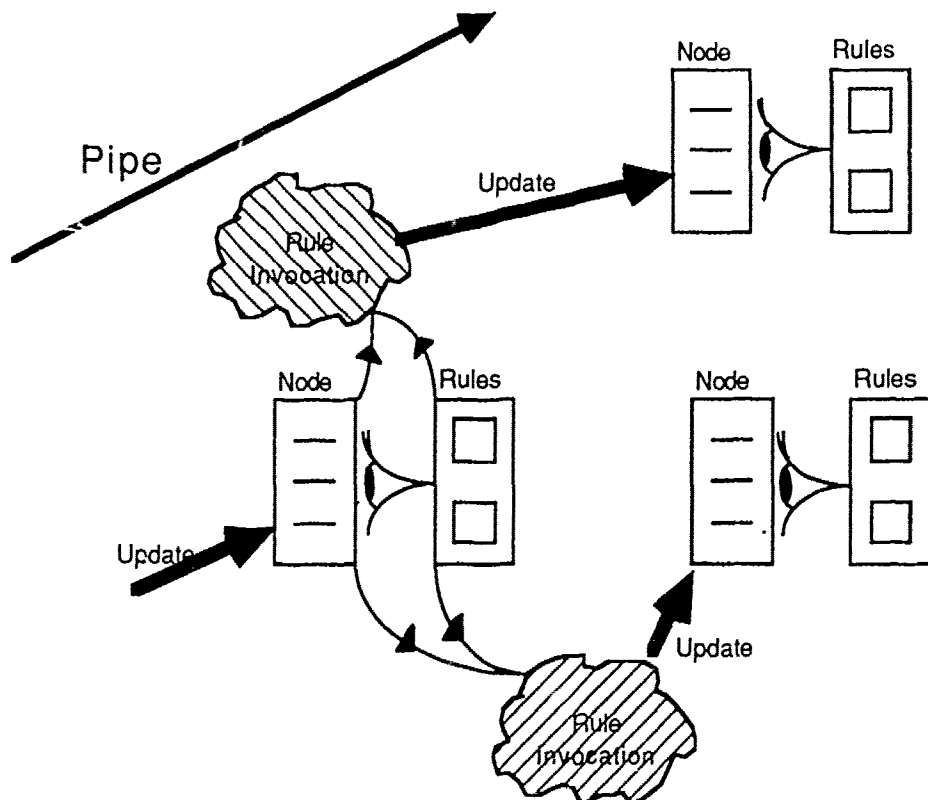


Fig. 5. Organization of Poligon.

Because a knowledge source is activated by a blackboard change, and because a knowledge source is a collection of rules, one can view the rules as being activated (indirectly, to be sure) by a change to some blackboard node. A rule could be activated by a change to a particular slot on a blackboard node. Slots with a property that trigger rules are called "trigger slots". When the action part of a rule is executed, the changes to the blackboard are communicated to the nodes to be changed. If a change is made to a trigger slot, then the condition parts of the "triggered rules" are evaluated; changes to non-trigger slots do not directly cause any processing.

Poligon was designed from the start to exploit "fine"-grained parallelism — "fine" grain here referring to parts of rules. It is generally thought that a shared-memory hardware architecture is not able to deliver increasing performance as more processors are added. This is a result of memory contention and of physical limits in the bandwidths of the busses and switches used to connect the processors to the memory. Thus, Poligon was designed from the start to be run on a form of distributed-memory multiprocessor, the elements of which communicate by sending messages to one another. Its match to the hardware will be seen clearly in the next section where we discuss the structure of Poligon and what makes it different from existing, serial implementations of blackboard systems.

4.1. The Structure of Poligon

In this section we describe the key features of Poligon. Instead of a detailed description of the implementation, a number of points which are central to Poligon's computational model are highlighted and contrasted with conventional blackboard implementations.

As has been mentioned above, Poligon is designed to run on hardware which provides message-passing primitives as the mechanism for communication between processing elements. It is important to note that the way in which information flows on the blackboard can be viewed, at an implementation level, as a message-passing process. This allows a tight coupling between the implementation of a system such as Poligon and the underlying hardware.

Poligon has no centralized scheduler. This was motivated by a desire to remove any bottlenecks that might be caused by the serial execution of such a scheduler and by multiple, asynchronous processes trying to put events onto the scheduler queue, causing memory contention. (The problems were clearly manifested in Cage.) This required the definition of a different knowledge invocation mechanism. Not only was a centralized scheduler eliminated but all global synchronization was eliminated as well. This means that it is likely that different parts of a Poligon program will run at different speeds and will have different ideas of how the solution is progressing.

Having eliminated the scheduler, there is clearly no need for any — presumably serializing — separation of the knowledge sources from the blackboard. The Poligon programmer, therefore, specifies at compile-time the classes of blackboard node that a particular piece of knowledge is interested in. At compile-time and at system initialization time, *knowledge is associated directly with the nodes on the blackboard that might invoke it.* This eliminates any communication delay and memory contention that might be caused by having to find a matching rule in a remote knowledge base.

In conventional blackboard systems, knowledge sources are taken to denote both units of knowledge and units of scheduling. If all that a system attempted to execute in parallel was its knowledge sources then a great deal of potential parallelism might be lost by the failure to exploit parallelism at a finer grain. In Poligon, therefore, *knowledge sources are not scheduling units*, they are simply collections of knowledge. *All of the rules in a knowledge source can, in principle, be invoked in parallel and parallelism at a finer grain than this can also be exploited during the execution of rules.*

Having eliminated the scheduler a new mechanism was needed that would cause the application's knowledge to be executed. It was decided to go for a very simple mechanism. *Poligon's rules are triggered as daemons by updates to slots in nodes.* The association between rules and the slots that trigger their invocation is made at compile-time, allowing efficient, concurrent invocation of all eligible rules after an event on the blackboard.

The message-passing metaphor for the implementation and the distribution of the knowledge base over the blackboard mentioned above, allowed the development of a *computational model which views a blackboard node as a process, responsible for its own housekeeping and for processing messages*, for instance, for slot updates and slot read operations.

Serial blackboard systems generally don't have a significant problem with the creation of new blackboard nodes. This is because of the atomic execution of knowledge sources. Such systems can usually be confident that, when a new node is created, no other node has been created that represents the same object. In parallel systems multiple, asynchronous attempts can be made to create nodes which are really intended to represent the same real-world object. Poligon provides mechanisms to allow the user to prevent this from happening.

It was found necessary occasionally to share data between a number of nodes. Poligon allows no global variables at all so it was necessary to find a suitable way of defining sharable, mutable data, whilst still trying to reduce the bottlenecks that can be caused by shared data structures. Poligon, like many frame systems, has a generalized class hierarchy with the classes themselves being represented as blackboard nodes. *Poligon uses class nodes as managers*, not only for node creation, as mentioned above, but also

to store data to be shared between all of the instances of a class and to support operations which apply to all members of a class.

Most blackboard systems represent the slots in nodes simply as value lists associated with the name of the slot. The serial operation of such systems allows the programmer to make assumptions about the order of elements in the value list. This assumption allows operations on all of the elements of the value list in the knowledge that no modification will have happened to the value list since it was read, because knowledge source executions are atomic. In Poligon, because a large number of rules can asynchronously be attempting to perform operations on a slot simultaneously, it was imperative to find mechanisms that would help to keep the operation of the system coherent without slowing down the access to slots too much, causing large critical sections and reducing parallelism. *Poligon, therefore, provides "smart" slots.* They can keep their values in the correct order and index them for flexible and focused data retrieval. They can also have user defined behavior which allows them to make sure that operations performed on them leave them consistent.

4.2. Shifting the Metaphor

Poligon's design looks very much like a frame-based program specialized for a particular implementation of the blackboard model. The expected behavior of the system is much closer than the serial systems to the blackboard problem-solving metaphor in one respect — the knowledge sources respond to changes in the blackboard *directly*.¹ As in Cage there are two major sources of concurrency in this scheme: (1) Each blackboard node can be active simultaneously to reflect data parallelism — the more blackboard nodes, the more potential parallelism. (2) Rules attached to a node can be running on many different processing elements simultaneously providing knowledge parallelism. This daemon-driven system with a facility for exploiting both data and knowledge parallelism poses some serious problems, however. First, it is easy to keep the processors and communication network busy, but the trick is to keep them busy converging toward a solution. Second, solutions to a problem will be non-deterministic — that is, each run will most likely produce different answers. Worse, a solution is not guaranteed since individual nodes cannot determine if the system is on the right path to an overall solution — that is, there is no global control module to steer the problem solving. Within the AI paradigm that looks for satisficing answers, non-determinism, per se, is not a cause for alarm; however, non-convergence or an incorrect solution is. One remedy to these problems is to introduce some global control mechanisms. Another solution is to develop a problem-solving scheme that can operate without a global view or global control. We have focussed our efforts in Poligon on the latter approach.

¹As an historical note, this takes us back to Selfridge's Pandemonium [Selfridge 59], which influenced Newell's ideas of blackboard-like programs [Newell 62]. It also has some of the flavor of the actor formalism [Hewitt 73].

4.2.1. Distributed, Hierarchical Control

A hierarchical control mechanism is introduced that exploits the structure of the blackboard data. The levels, in the AGE sense, of the blackboard are organized as a class hierarchy. Each level is a class and a blackboard node is an instance of that class. Class nodes contain information about their instances (number of instances, their address, and so on), and knowledges sources can be attached to class nodes to control their instance nodes. To minimize confusion, class nodes will be referred to using a more concrete term, *level manager*. Similarly, a super-manager node can control the class nodes.

1. Within Poligon, the potential for control is located in three types of places: Within each node, where action parts of the rules can be executed serially, for example.
2. In the level manager which can, for example, be used to monitor the activities of its nodes. Since the level manager is the only agent that knows about the nodes on its level, a message that is to be sent to all the instance nodes must be routed through their manager node. The level manager also controls the creation and garbage collection of the nodes, and attaches the relevant rules to newly created nodes.
3. In the super-manager, whose span of control includes the creation of level managers and their activities, and indirectly their offspring.

The introduction of control mechanisms solves some of the difficulties, but it also introduces bottlenecks at points of control, for example, at the level manager nodes. One solution to this type of bottleneck is to replicate the nodes, that is, create many copies of the manager nodes. The CAOS experiments, mentioned earlier, took this approach [Brown 86]. Although Poligon supports this strategy, our research is leading us to try a different tactic.

4.2.2. A New Rôle for Expectation-driven Reasoning

It was initially conjectured that model-driven and expectation-driven processing would not play a significant role in concurrent systems — at least not from the standpoint of helping with performance. One view of top-down processing is that it is a means of gaining efficiency in serial systems in the following way: In the class of applications under consideration, the interpretation of data proceeds from the input data up an abstraction hierarchy — the amount of information being processed is reduced as it goes up the hierarchy. Expectations, posted from a higher level to a lower level, indicate data needed to support an existing hypothesis; data expected from predictions; and so on. Thus, when an expected event does occur, the bottom-up analysis need not continue up — the higher level node is merely notified of the event and it does the necessary processing, for example, increases the confidence in its hypothesis. When the analysis involves a large search space, this expectation-driven approach can save a substantial amount of processing time in serial systems.

In Poligon hot-spots often occur at a node to which many lower level nodes communicate their results (a fan-in). The upward message traffic can be reduced by posting expectations on the lower level nodes and having them report back only when *unexpected* events occur. This approach, currently under investigation, is one way for a node to distribute parts of the work to lower level nodes, and hopefully relieves the type of bottlenecks caused by fan-ins at a node without resorting to node replication.

It is generally expected that, within the abstraction hierarchy of the blackboard, information volume is reduced as one goes up the hierarchy. This translates into the following desiderata for concurrent systems: For an arbitrary node to avoid being a hot-spot, there must be a decrease in the rate of communication proportional to the number of nodes communicating to it. That is, the wider the fan, the less communication is allowable from each node. It was found while re-implementing the serial ELINT application in Poligon, that the highest level nodes had to be updated for almost every new data item. Such a formulation of the problem, while posing no problem in serial systems, reduces parallelism in concurrent problem solvers.

4.2.3. A New Form of Rule

If, for any given data item, there are many rules that check its state then the system must ensure that this data item does not change until all of those rules have checked it. A typical example is as follows: Suppose there are two rules that are mutually exclusive, one performs some action if a data value is "on" and the other performs some other action if the value is "off." How can we ensure that between the time the first rule accesses the data and the second does so, there is not some other action that changes the data? It was found in Poligon (and also in Cage) that these mutually exclusive rules need to be written in the form of case-like conditionals to assure data consistency of the form described above. Since the need for process creation, and subsequent maintenance, is reduced through combining rules, this form of rule also aids in speeding up the overall rule execution. It does mean, however, that the grain size of some of the rules has been made bigger, at least at the source code level, and the programmers must think differently about rules than they do in current expert systems.

4.2.4. Agents with Objectives

At any given point in the computation, the data at different nodes can be mutually inconsistent or out of date. There are many causes for this, but one cause is that blackboard changes are communicated by messages and the message transit time is unpredictable. In the applications under consideration, where there are one or more streams of continuous input data, the problem appears as scrambled data arrival — the data may be out of temporal sequence or there may be holes in the data. Waiting for earlier data does not help, since there is no way to predict when that data might appear. Instead, the node must do the best it can with the information it

has. At the same time, it must avoid propagating changes to other nodes if its confidence in its output data or inferences is low.

Put another way, *each node must be able to compute with incomplete or incorrect data, and it must 'know' its objectives to enable it to evaluate the resulting computation. A result is passed on only if it is known to be an improvement on a past result.* This represents a change from the problem-solving strategies generally employed in blackboard systems where the control/scheduling module evaluates and directs the problem solving. With no global control module to evaluate the overall solution state and with asynchronous problem-solving nodes, a reasonable alternative is to make each node evaluate its own local state. Of course, there is no guarantee that the sum total of local correctness will yield global correctness. However, the way that blackboard systems are generally organized — each blackboard level representing a class of solution islands, the span of knowledge sources being limited to a few levels, and having functionally independent knowledge sources — appears at this point, to provide an appropriate methodology for creating loosely-coupled nodes that can be provided with local objectives and a capability for self-evaluation.¹ The "smart" slots mentioned earlier are used to implement this strategy.

The design of Poligon poses an interesting question — is it still a blackboard system? There is a substantial shift in the problem-solving behavior and in the way the knowledge sources need to be formulated. The structure of the solution is not globally accessible. There is no control module to guide the problem solving at run time. The metaphor shifts to one in which each "blackboard" node is assigned a narrow objective to achieve, doing the best it can with the data passed to it, and passing on information only when the new solution is better than the last one. The collective action of the "smart" agents results in a satisficing solution to a problem.²

Although there is a substantial shift away from the conventional problem solving metaphor, Poligon evolved out of the mechanisms that were present in AGE. Most of the same opportunities for concurrency made available to the user in Cage are built into the system in Poligon. The Poligon language forces the user to think in terms of blackboard levels and knowledge sources. But the underlying system has no global data. Whether such a formulation makes the job of constructing concurrent, knowledge-based

¹It is interesting to note that the need for local goals does not seem to change with process granularity. Although the methods used to generate the goals are very different, Lesser's group has found that each node in its distributed system needs to have local goals [Durfee 85]. In this system each node contains a complete blackboard system; each system (node) monitors the activities in a region of a geographic area which is monitored collectively by the system as a whole.

²In retrospect, these characteristics for concurrent problem solving seem obvious. When a group of humans solve a problem collectively by subdividing a task, we assume each person has the ability to evaluate his or her own performance relative to the assigned task. When there are "uncaring" people, the overall performance is bad, both in terms of speed and solution quality.

systems easier or more difficult for the knowledge engineer still remains to be seen. A difficulty might arise because the semantics of the Poligon language, that is, the mapping of the blackboard model to the underlying software and hardware architecture, is hidden from the user. For example, there is no notion of message-passing or of a distributed blackboard reflected in the Poligon language. In contrast, the choice of what, and how, to run concurrently is completely under user control in Cage.

5. Conclusions

In this paper we discussed the relationship between the blackboard model, its existing serial implementations, and the degree to which the intuitively inherent parallelism is really present.

Cage and Poligon, two implementations of the blackboard model designed to operate on two different parallel hardware architectures, were described briefly, both in terms of their structure and the motivation behind their design.

Our framework development, application implementations on these frameworks, and initial performance experiments to date has taught us that: (1) it is difficult to write a real-time, data interpretation programs in a multi-processor environment, and (2) performance gains are sensitive to the ways in which applications are formulated and programmed. In this class of application, performance is also sensitive to data characteristics.

The "obvious" sources of parallelism in the blackboard model, such as the concurrent processing of knowledge sources, do not provide much gain in speed-up if control remains centralized. On the other hand, decentralizing the control, or removing the control entirely, creates a computational environment in which it is very difficult to control the problem-solving behavior and to obtain a reasonable solution to a problem. As granularity is decreased, to obtain more potential parallel components, the interdependence among the computational units tends to increase, making it more difficult to obtain a coherent solution *and* to achieve a performance gain at the same time. We described some of the methods employed to overcome these difficulties.

In the application class under investigation, much of the parallelism came from data parallelism — both from the temporal data sequence and from multiple objects (aircrafts, for example) — and from pipe-lining up the blackboard hierarchy. The ELINT application was unfortunately knowledge poor, so that we were unable to explore knowledge parallelism, except as a by-product of data and pipeline parallelism. ELINT has been implemented in both Cage and Poligon, and experiments are now being performed. The experiments are designed to measure and to compare performance by varying different parameters: process granularity, number of processors, data rate, data arrival characteristics, and so on.

It is clear that much more research is needed in this area before a combination of a computational and problem-solving model can be developed that is easy to use, that produces valid solutions reliably, and that can increase performance by a significant amount.

6. Bibliography

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism: The Cage System*. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.
- [Amdahl 67] Gene M. Amdahl. *Validity of a Single Processor Approach to Achieving Large Scale Computing Capabilities*. Proceedings of AFIPS Computing Conference 30, 1967.
- [Brown 82] Harold Brown, Jack Buckman, et al. *Final Report on Hannibal*. Technical Report, ESL, Inc., 1982. Internal Document.
- [Brown 86] Harold D. Brown, Eric Schoen and Bruce Delagi. *An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures*. Technical Report KSL-86-69, Knowledge Systems Laboratory, Computer Science Department, Stanford University, October 1986.
- [Corkill 83] Daniel D. Corkill and Victor R. Lesser. *The Use of Meta Level Control for Coordination in Distributed Problem Solving*. Proceedings of the 7th International Conference on Artificial Intelligence: 748-755, 1983.
- [Delagi 86a] Bruce Delagi. *CARE Users Manual*. Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 86b] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-76, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 88] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Durfee 85] Edmund Durfee, Victor Lesser and Daniel Corkill. *Coherent Cooperation Among Communicating Problem Solvers*. Technical Report, Department of Computer and Information Sciences, September, 1985.

- [Ensor 85] J. Robert Ensor and John D. Gabbe. *Transactional Blackboards*. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.
- [Fennell 77] Richard D. Fennell and Victor R. Lesser. *Parallelism in AI Problem Solving: A case Study of Hearsay-II*. IEEE Transactions on Computers: 98-111, February, 1977.
- [Forgy 77] Charles Forgy and John McDermott. *OPS, A Domain-Independent Production System Language*. Proceedings of the 5th International Joint Conference on Artificial Intelligence 1:933-939, 1977.
- [Gupta 86] Anoop Gupta. *Parallelism in Productions Systems*. Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1986. Ph. D. dissertation.
- [Hewitt 73] Hewitt, C., P. Bishop and R. Steiger. *A Universal, Modular Actor Formalism for Artificial Intelligence*. Proceedings of the 3rd International Joint Conference on Artificial Intelligence: 235-245, 1973.
- [Lesser 83] Victor R. Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks*. The AI Magazine, Fall:15-33, 1983.
- [McCune 83] Brian P. McCune and Robert J. Drazovich. *Radar with Sight and Knowledge*. Defense Electronics, August 1983.
- [McDermott 83] John McDermott and Allen Newell. *Estimating the Computational Requirements for Future Expert Systems*. Technical Report, Internal Memo, Computer Science Department, Carnegie-Mellon University, 1983.
- [Newell 62] Allen Newell. *Some Problems of Basic Organization in Problem-Solving Programs*. In M. C. Yovits, G. T. Jacobi and G. D. Goldstein (editors), Conference on Self-Organizing Systems, pp 393-423. Spartan Books, Washington, D.C., 1962.
- [Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 82] H. Penny Nii, Edward A. Feigenbaum, John J. Anton and A. Joseph Rockmore. *Signal-to-Symbol Transformation: HASP/SIAP Case Study*. AI Magazine vol 3-2, 23-35, 1982.

- [Nii 86] H. Penny Nii. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in *AI Magazine*, vol. 7-2 and vol. 7-3, 1986.
- [Rice 86] James Rice. *Poligon: A System for Parallel Problem Solving*. Technical Report KSL-86-19, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
- [Schoen 86] Eric Schoen. *The CAOS System*. Technical Report KSL-86-22, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
- [Selfridge 59] Oliver G. Selfridge. *Pandemonium: A Paradigm for Learning*. Proceedings of the Symposium on the Mechanization of Thought Processes: 511-529, 1959.
- [Shafer 86] Steven A. Shafer, Anthony Stentz and Charles Thorpe. *An Architecture for Sensor Fusion in a Mobile Robot*. Proceedings of the 1986 IEEE International Conference on Robotics and Automation, 1986.
- [Smith 81] Smith, B.J. *Architecture and Applications of the HEP Multiprocessor Computer System*. Proceedings of the International Society for Optical Engineering. San Diego, California, August 25-28, 1981.
- [Spain 83] David S. Spain. *Application of Artificial Intelligence to Tactical Situation Assessment*. Proceedings of the 16th EASCON83: 457-464, September, 1983.
- [Williams 84] Mark Williams, Harold Brown and Terry Barnes. *TRICERO Design Description*. Technical Report ESL-NS539, ESL, Inc., May, 1984.

Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems

by

H. Penny Nii, Nelleke Aiello and James Rice
(Nii, Aiello, Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

To appear in *Distributed Artificial Intelligence II* edited by L. Gasser and M. N. Huhns. Pitman Publishing Ltd. and Morgan Kaufmann, 1989.

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

Some ways in which blackboard systems can be made to operate in a multi-processor environment are described in this paper. Cage and Poligon are two concurrent problem solving systems based on the blackboard model. The factors which motivate and constrain the design of parallel systems in general and parallel problem-solving systems in particular are discussed. Experiments performed on these two software architectures are described and their results and implications enumerated and explained.

1. Introduction

In this paper we introduce two software systems, Cage and Poligon. Cage and Poligon are two blackboard systems designed to exploit multiprocessor hardware with the intent of achieving computational speed-up. Blackboard systems, although architecturally well suited for problems requiring the interpretation of multiple streams of signal and symbolic data, are often computationally too expensive to perform reasonably in a real-time environment. Cage and Poligon, the results of two sub-projects of the Advanced Architectures project at the Knowledge Systems Laboratory of Stanford University, are attempts to produce high performance parallel blackboard systems.

The Cage system is a conservative attempt to introduce parallelism into the existing, serial blackboard architecture AGE. The Cage architecture represents an experiment into what could reasonably be achieved given the current state of commercially available multiprocessors, most of which are shared-memory machines with from several to a few tens of processors. An example of such a machine might be the BBN ButterflyTM machine¹.

Poligon, which makes a radical shift from conventional blackboard systems, anticipates future developments in parallel hardware architectures. It is designed to work on the next generation of distributed-memory machines using hundreds or thousands of processors.

A general background and the general motivations for the development of Cage and Poligon are discussed in Sections 2 and 3. The rationales for the design of Cage and Poligon are discussed in Sections 4 and 5 respectively. Since both systems run on simulated machines, the simulation system, CARE, is discussed briefly in Section 6.

Experiments have been performed on these two systems, some of which are described in Section 8 together with their results. The application problem, Elint, which drove the experiments, is described in Section 7. Since Cage and Poligon are very different systems, both from the standpoint of software design and hardware requirements, it is difficult to compare the per-

¹Butterfly is a registered trade mark of Bolt Beranek and Newman Corporation.

formance of the two systems. In addition, since the research goals for the two system architectures are different, the set of experiments performed on them are different. To facilitate some form of comparison between the two types of system, however, one particular experiment was designed to be performed, as closely as reasonably possible, on both systems. The relative performance of the two systems will be discussed in Section 9 in the context of these particular experimental results.

2. Background

A *Concurrent Problem Solving System* is a network of autonomous, or semi-autonomous, computing agents that solve a single problem. In building concurrent problem solvers, our objectives are twofold: (1) to evolve or invent models of problem solving in a multi-agent environment and (2) to gain significant performance improvement by the use of multiprocessor machines. One of the important practical concerns of using many computers in parallel is to gain computational speed-up¹. Centralized control is useful in a serial (single) problem solver for obtaining a valid solution and coherent problem-solving behavior, but it is not compatible with performance gain in a multi-agent environment. Cage and Poligon attempt to find a balance; to achieve adequate coherence with minimal global control *and* to gain performance with the use of multiple processors.

2.1. Problem Solving and Concurrency

Those problems that have been successfully solved in parallel, such as partial differential equations and finite element analysis, share common characteristics. They frequently use vectors and arrays; solutions to the problems are very regular, using well understood algorithms; and the computational demands, for example, for matrix inversion, are relatively easy to compute. In contrast, the class of applications we are addressing (and AI problems in general) are ill-structured and/or ill-defined. There is often more than one possible solution. Paths to a solution cannot be predefined and must be dynamically generated and tried, and generally, data cannot be encoded in a regular manner in array-like structures. The data structures for the solution states are often graph structures that must be dynamically created, precluding static allocation and optimization. These differences indicate that to run problem solving programs in parallel, current techniques for parallel programs must be augmented or new ones invented. It is worth reviewing some of the key points to be addressed in building concurrent, problem-solving programs.

2.1.1. Problem Solving Issues

Problem solving has traditionally meant a process of searching a tree of alternative solutions to a problem. Within each generate-and-test cycle, alternatives are generated at a node of a tree and promising ones selected for

¹Although multiple computers can be used because of the need for redundancy, a mix of specialized hardware or a need for physical separation, and so on.

further processing. Knowledge is used to prune the tree of alternatives or to select promising paths through the tree. It is an axiom that the more knowledge there is, the less generation and testing has to be done. In most expert systems pieces of knowledge recognize intermediate solutions and solution paths, thereby eliminating search. These two types of problem-solving techniques have been labeled *search* and *recognition* [McDermott 83]. In the search technique the majority of computing time is taken up in generating and testing alternative solutions; in the recognition technique the time is taken up in *matching*, a process of finding the right piece of knowledge to apply. Most applications use a combination of search and recognition techniques. A concurrent problem solving framework must be able to accommodate both styles of problem solving.

In serial systems meta-knowledge, or control knowledge, is often used to reduce computational costs. One common approach decomposes a problem into hierarchically organized sub-problems, and a control component selects an efficient order in which to solve these sub-problems. This approach enhances the performance of search and recognition problem solving by reducing the number of alternatives to search or the amount of knowledge to match. In concurrent systems meta-knowledge and controllers become fan-in points, or hot-spots. A *hot-spot* is a physical location in the hardware where a shared resource is competed for, forcing an unintended serialization. Does this mean that problem solving systems that rely on centralized control are doomed to failure in a concurrent environment? Can control be distributed? If so, to what extent? If more knowledge results in less search, can a similar trade-off be made between knowledge and control? That is, in concurrent systems where control, especially global control, is a serializing process, can knowledge be brought to bear to alleviate the need for control? These are some of the basic questions that studies in concurrent problem solving need to address.

2.1.2. Concurrency Issues

The biggest problem in concurrent processing was first described by Amdahl [Amdahl 67]. Simply stated, it is as follows: The length of time it takes to complete a run with parallel processes is the length of time it takes to run the longest serial process plus some overhead associated with running things in parallel. Take a problem that can be decomposed into a collection of independent sub-problems that can run concurrently, but which internally must run serially. If all of these components are run concurrently, then the run-time for the whole problem will be equal to the run-time for the longest running component, plus any overhead needed to execute the sub-problems in parallel. Thus, if the longest process takes 10% of the total run time if the processes were run end-to-end (serially), then the maximum speed-up possible is a factor of 10. Even if only one percent of the processing must be done sequentially this limits the maximum speed-up to one hundred. However hard one tries and however many processors are used. This is a very depressing result, since it means that many orders of magnitude of speed-up are only available in very special circumstances.

This raises the issue of *granularity*, the size of the components to be run in parallel. Amdahl's argument indicates the need for as small a granularity as possible. But, if the overhead cost of process creation and process switching is expensive, we want to do as much computation as possible once a process is running, that is, favor a larger granularity. In addition, in a multi-computer architecture a balance must be achieved between the load on the communication network and on the processors. It is often the case that as process granularity decreases, the processes become more tightly coupled, that is, there is a need for more communication between them. The communication cost is, of course, a function of the hardware-level architecture, including bandwidth, distance, topology, and so on. Finding an optimal grain size at the problem solving level is a multi-faceted problem.

Even if one is able to find an optimal granularity, there are forces that inhibit the processes from running arbitrarily fast in parallel. Some of the more common problems are discussed below.

- *Hot-Spots and Bottlenecks*: It is frequently the case that a piece of data must be shared. Multiple, simultaneous requests to access the same piece of data cause *memory contention*. A number of processes competing for a shared resource — memory or processors — causes a degradation in performance. These processor and memory hot-spots restrict the flow of data and reduce parallelism.
- *Communications*: Multi-computer machines do not have a shared address space in which to have memory bottlenecks of the kind mentioned above. However, the communications network over which the processing elements communicate represents a shared resource which can be overloaded. It has a finite bandwidth. Similarly, multiple, asynchronous messages to a single processing element will cause that element to become a hot-spot.
- *Process Creation*: Execution of the sub-problems, into which the overall problem is divided, requires that they run as processes. The cost of the creation and management of such processes is non-trivial. There is a process grain size at which it is faster to run many sub-processes sequentially than to execute them in parallel.

Some issues and constraints associated with parallelizing programs were introduced above. We now introduce some concepts that are important in writing concurrent programs, an understanding of which is useful to subsequent discussions.

- *Atomic operation*: This refers to a piece of code which is executed without interruption. In order to have consistent results (data) it is important to define appropriate atomic operations. For instance, an update to a slot in an object might be defined to be atomic. Primitive atomic actions are usually defined at the system level.

- *Critical sections*: Critical sections are usually programmer-defined and refer to those parts of the program which are uninterruptible, that is, atomic. The term is usually used to describe large, complex operations that must be performed without interruption.
- *Synchronization*: This term is used to describe that event which brings asynchronous, parallel processes together synchronously. Synchronization primitives are used to enforce serialization.
- *Locks*: Locks are mechanisms for the implementation of critical sections. Under some computational models, a process that executes a critical section must acquire a lock. If another process has the lock, then it must wait until that lock is released.
- *Pipeline*: A pipeline is a series of distinct operations which can be executed in parallel but which are sequentially dependent; for instance, an automobile assembly line. The speed-up that can be gained from a pipeline is proportional to the number of pipeline stages, assuming that each stage takes the same amount of time. Such a pipe is "well balanced." Because reasoning consists of sequentially dependent inference steps, pipeline parallelism is a very important source of parallelism in problem solving programs.

2.1.3. Background Motivation

In experiments conducted at CMU [Gupta 86], Gupta showed that applications written in OPS [Forgy 77] achieved speed-up in the range of eight to ten, the best case being about a factor of twenty. The experiments ran rules in parallel, with pipelining between the condition evaluation, conflict resolution, and action executions. The overhead for rule matching was reduced with the use of a parallelized Rete algorithm. (In programs written in OPS, roughly 90% of the time is spent in the match phase.) The speed-up factors seem to reflect the amount of relevant knowledge chunks (rules) available for processing a given problem solving state, and this number appears to be rather small. Although the applications were not written specifically for a parallel architecture, the results are closely tied to the nature of the OPS system itself, which uses a monolithic and homogeneous rule set and an unstructured working memory to represent problem solving states.

The premise underlying the design of Cage and Polygon is that this discouraging result could be overcome by dividing and conquering. It is hoped that by partitioning an application into loosely-coupled sub-problems (thus partitioning the rule set into many subsets of rules), and by keeping multiple states (for the different sub-problems), multiplicative speed-up, with respect to Gupta's experimental results, can be achieved. If, for example, a factor of seven speed-up could be achieved for each sub-problem by the simultaneous execution of its rules, it is possible to obtain an overall speed-up of seven times the number of sub-problems. The challenge, of course, is to coordinate the resulting asynchronous, concurrent, problem-solving pro-

cesses toward a meaningful solution with minimal overheads. The focus of Cage and Poligon has been on the methods and techniques required to obtain coherent solutions from many independent sub-problem solvers.

2.2. The Blackboard Model and Concurrency

The foundation for most knowledge-based systems is the problem-solving framework in which an application is formulated. The problem-solving framework implements a computational model of problem solving and provides a language in which an application problem can be expressed. We begin with the Blackboard Model [Nii 86a], which is a problem-solving framework for partitioning problems into many loosely coupled sub-problems. Both Cage and Poligon have their roots in the blackboard model of problem solving. The blackboard approach seems, at first glance, to admit the natural exploitation of concurrency, such as:

- Knowledge parallelism, in which the knowledge sources and rules within each knowledge source can run concurrently;
- Pipeline parallelism, in which transfer of information from one level to another (one method of implementing a reasoning chain) forms pipelines; and
- Data parallelism, in which the blackboard is partitioned into solution components that can be operated on concurrently.

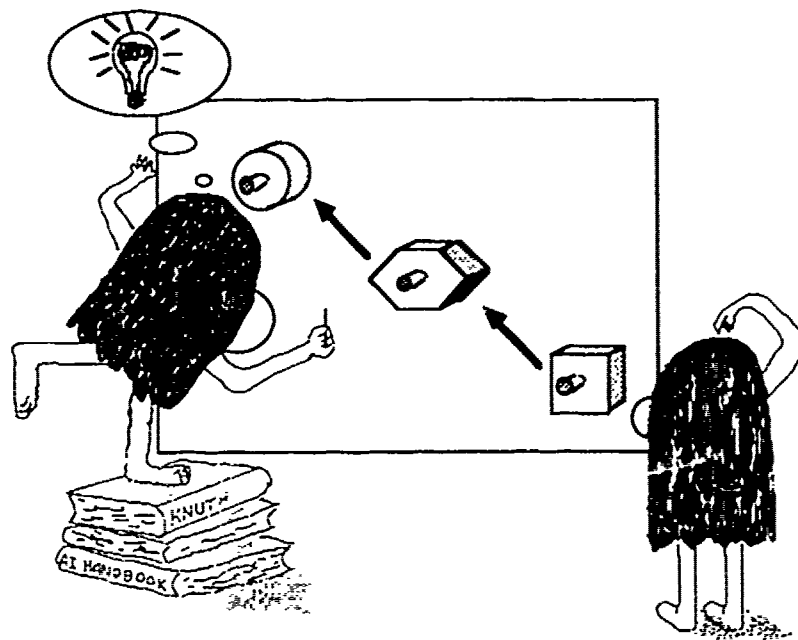


Figure 2.1. The Blackboard Metaphor

In addition, the dynamic and flexible control component can be extended to control the parallel execution of different components of the system.

These characteristics of blackboard systems have prompted investigators, for example Lesser and Corkill [Lesser 83] and Ensor and Gabbe [Ensor 85], to build distributed and/or parallel blackboard systems. The study of parallelism in blackboard systems goes back to Hearsay-II [Fennell 77].

The blackboard problem solving metaphor is very simple: A collection of intelligent agents gather around a blackboard, look at pieces of information written on it, think about them, and add their conclusions as they come to them. This is shown in Figure 2.1.

There are some basic assumptions made in this model, an understanding of the implications of which is vital to an understanding of the difficulties of achieving parallelism in blackboard systems.

- All of the agents can see all of the blackboard all of the time, and what they see represents the current state of the solution.
- Any agent can write his conclusions on the blackboard at any time without getting in anyone else's way.
- The act of an agent writing on the blackboard will not confuse any of the other agents as they work.

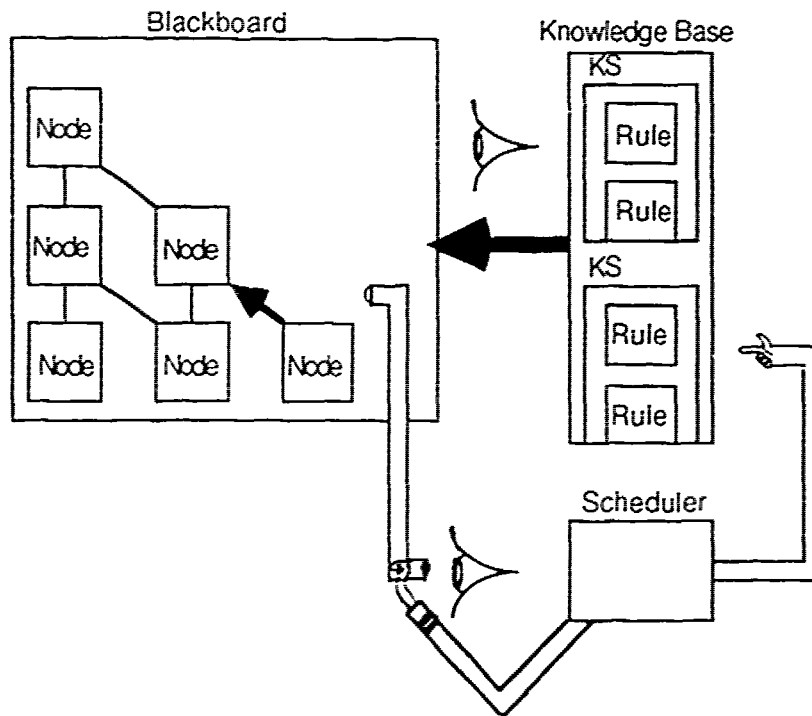


Figure 2.2. The Serial Blackboard Model

These assumptions imply that a single problem is being solved asynchronously and in parallel. However, the problem solving behavior, if it were to be emulated in a computer, would result in very inefficient compu-

tation. For example, for every agent to "see" everything "simultaneously" requires stopping everything until every agent has looked at everything.

Existing, serial blackboard systems make a number of modifications to this blackboard metaphor in order to make a reasonable implementation on conventional, serial hardware. In effect, the blackboard metaphor is modified so that it *cannot* be executed in parallel. Some of these modifications are shown in Figure 2.2 and are described below.

- Agents are represented as *knowledge sources*. These knowledge sources are schedulable entities and only one can be running at any time. It will be shown later that one of the possible sizes for computational grains is the knowledge source.
- To coordinate the execution of knowledge sources, a scheduler, or some other central control mechanism, is implemented. In many ways, this is an efficiency gaining mechanism in which control knowledge selects only the most productive knowledge source at any given moment to work on the problem.
- The blackboard is not "globally visible" in the sense prescribed by the blackboard metaphor. Instead, the blackboard is implemented as a data structure, which is sufficiently interconnected to enable a knowledge source to find its way from one data item to a related one easily. Knowledge sources generally work on a limited area of the blackboard, known as the knowledge source's context. Often knowledge sources and their contexts are treated as self-contained sub-problems.
- An implicit assumption is made that a knowledge source operates within a valid, or *consistent* context. That is, the values of the different properties of an objects on the blackboard are consistent with respect to each other and are the same everywhere they are mentioned. To assure this, for example, a knowledge source is never interrupted while it is making changes to the blackboard.

Trying to parallelize serial blackboard systems characterized above directly has certain limitations. First, the granularity of the central scheduler that decides what knowledge sources to run in parallel is one of the main factors that limit speed-up. The results of a Cage experiment described later indicates that the main reason for its modest speed-up was the scheduler. To attain significant speed-up, the notion of centralized control must be abandoned. Second, one of the most difficult problems in parallel computation is to maintain consistent data values and to achieve a coherent solution state. In concurrent blackboard systems, consistency and coherence problems occur in three different contexts: (1) on the entire blackboard — maintaining relationships that "make sense" among the objects (global coherence of the solution); (2) in the contents of the nodes — assuring that all slot values are from the same problem solving state (mutual data consistency); and (3) in

the slots — keeping the value being evaluated from changing before the evaluation is completed (data consistency).

3. The Advanced Architectures Project

Cage and Poligon, two frameworks for concurrent problem solving, are being developed within the Advanced Architectures Project (AAP) [Rice 88] at the Knowledge Systems Laboratory of Stanford University. The objective of the AAP is the development of broad system architectures that exploit parallelism at different levels of a system's hierarchical construction. To exploit concurrency one must begin by looking for parallelism at the application level and be able to formulate, express, and utilize that parallelism within a problem-solving framework. A framework must, in turn, be supported by an appropriate language and software/hardware system. The system levels chosen and some issues for study are:

- Application level: How can concurrency be recognized and exploited?
- Problem solving level: Is there a need for a new problem-solving metaphor to deal with concurrency? What is the best process and data granularity? What is the trade-off between knowledge and control?
- Programming language level: What is the best process and data granularity at this level? What are the implications of choices at the language level for the hardware and system architecture?
- System/hardware level: Should the address spaces be common or disjoint? What should the processor and memory characteristics and granularity be? What is the best communication topology and mechanisms? What should the memory-processor organization be?

At each system level one or more specific methods and approaches have been implemented in an attempt to address the problems at that level. These programs are then vertically integrated to form a family of experimental systems — an application is implemented using a problem-solving framework using a particular knowledge representation method, all of which use a specific programming language, which in turn runs on a specific system/hardware architecture simulated in detail on the Lisp-based CARE simulator [Delagi 86a] (see Section 6). Each family of experiments is designed to evaluate, for example, the system's performance with respect to the number of processors, the effects of different computational granularity on the quality of solution and on execution speed-up, ease of programming, and so on. The results of one such family of experiments have been reported by Brown and Schoen [Brown 86, Schoen 86].

Within the context of this AAP organization, Cage and Poligon are two frameworks (or shells) implemented to study the problem-solving level.

Both Cage and Poligon use frames and condition-action rules to represent knowledge. The target system architecture for Cage is shared-memory multiprocessors; the target architecture for Poligon is distributed-memory multiprocessors, or multi-computers.

Both Cage and Poligon aim to solve a particular, but broad, class of applications: the interpretation of continuous streams of errorful data, using many diverse sources of knowledge. Each source of knowledge contributes pieces of a solution which are integrated into a meaningful description of the situation. Applications in this class include a variety of signal understanding, information fusion, and situation assessment problems. The utility of blackboard formulations has been successfully demonstrated by programs written to solve problems in our target application class [Brown 82, McCune 83, Nii 82, Shafer 86, Spain , Williams 84].

Most of the systems in this class use the recognition style of problem solving with knowledge bases of facts and heuristics; numerical algorithms are also included as a part of the knowledge. Some search methods are employed, but they are generally confined to a few of the knowledge sources.

An example problem in this class, called Elint (described in Section 7), was implemented in both Cage and Poligon.

In designing a concurrent blackboard system for the AAP, two distinct approaches seemed possible; one, to extend a serial blackboard system, and the other, to devise a new architecture to exploit the event-driven nature of blackboard systems. Each has its advantages and problems; they will be described in the following sections.

4. Extending the Serial System: Cage

In this section we discuss the Cage system, its origins and its architecture. In order to put this into a proper perspective, we first give a brief description of the (serial) AGE system [Nii 79], upon which Cage (Cage \equiv Concurrent AGE) is closely modelled. The AGE and Cage systems are functionally identical other than that Cage allows parts of the system to be executed in parallel.

4.1. The AGE system

The AGE system is one implementation of the blackboard problem-solving model [Nii 86] mentioned in Section 2.2. The knowledge in an AGE application is expressed both in the structure of the blackboard — the declaration of the blackboard *levels* — and in the knowledge base itself. An AGE knowledge base is composed of a number of *knowledge sources*, each of which contains a number of *rules*. Rules are condition-action pairs, as is the case in most blackboard systems.

Knowledge sources are invoked by the scheduler, which is user programmable. The selection of applicable knowledge sources is performed by the use of *Events*.

An event is a symbolic token, which is posted by AGE after a knowledge source makes a significant modification to the blackboard. For instance, a chess playing blackboard system might, after placing the opponent in check, post an event indicating that the opponent was in check. This event is recorded by the system on a global event queue along with information about the posting agent and the cause of the event. This allows the system to focus its attention on the parts of the blackboard which are active and provides the appropriate context in which to invoke any appropriate knowledge sources. The event tokens are defined by the user and posted automatically by the AGE system any time a node on the blackboard is changed.

The knowledge sources are labelled with the event tokens in which they are interested. This allows the user specified scheduling mechanism to invoke only those knowledge sources whose label matches the event token. The label on the knowledge source is referred to as the knowledge source *precondition*.

Within the knowledge sources rules can be invoked in two ways:

- The condition parts of the rules are evaluated until a match is found. This search for an applicable rule is performed serially in the lexical order of the rules. This mechanism is referred to as *Single-Hit*.
- The condition parts of all of the rules are evaluated and all rules that match are executed. The execution of the action parts of the matched rules is performed serially in the lexical order of the rules. This mechanism is referred to as *Multiple-Hit*.

These rule invocation strategies are peculiar to the AGE system and its derivatives.

4.2. The Cage Architecture

The basic components of the Cage system are:

- A global data store (the blackboard) on which emerging solutions are posted as object, attribute, and value triples. Objects on the blackboard are organized into levels of abstraction.
- Globally accessible lists on which control information is posted (for example, a list of events, a list of expectations, and so on).
- An arbitrary number of knowledge sources, each consisting of an arbitrary number of rules.

- Control information that can help to determine at any given point in the problem-solving process which blackboard node is to be in focus and which knowledge sources are to be executed.
- Declarations that specify which components are to be executed in parallel and at what points synchronization is to occur. The components for potential concurrency are knowledge sources, rules, condition parts of rules, and action parts of rules.

The user can run Cage serially (at which point Cage behavior is identical to that of AGE), or with pre-specified components running concurrently. In the serial mode, the basic control cycle begins with the selection and execution of a knowledge source. Cage uses a global list structure, called the *event list*, to record the changes to the blackboard. The scheduler selects one of the events (the user can specify how the event is to be selected, such as FIFO, LIFO, or any user-defined best-first method). The resulting event in focus is then matched against the knowledge source preconditions. The knowledge sources, whose preconditions match the focus event, are then executed in some predetermined order. The condition parts of the rules within each knowledge source are evaluated, and the action parts of the rules, whose conditions are satisfied, are executed. Each action part may cause one or more changes on the blackboard which are recorded on the event list. Figure 4.1 shows the Cage control cycle in the serial mode.

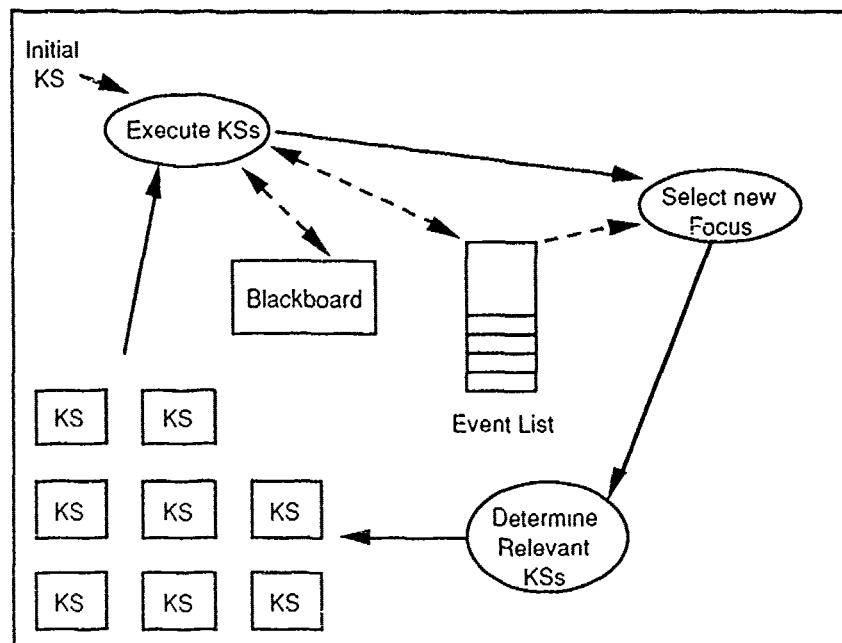


Figure 4.1. Cage Serial Control Cycle

By selecting one of the concurrency control options, the user can alter the simple, serial execution of knowledge sources and their parts to execute in parallel. The various concurrency options shown in Figure 4.2 are summarized below.

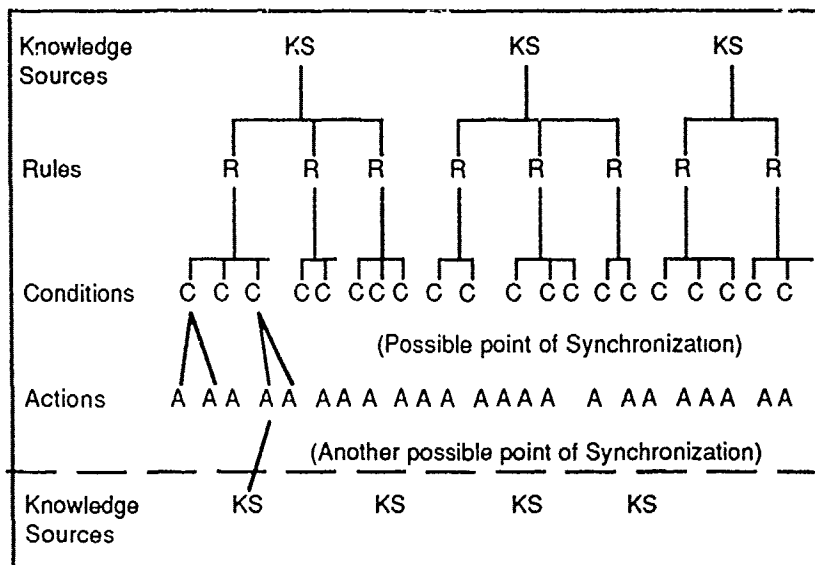


Figure 4.2. Parallel Components of Cage

Knowledge Source Control

Serial:

Pick an event and execute the associated knowledge sources.

Parallel:

As each event is generated execute the associated knowledge sources in parallel

Or Wait until all the active knowledge sources complete execution and invoke the knowledge sources relevant to all the resulting events concurrently.

Within Each Knowledge Source

Serial:

Perform context evaluation and then,

Evaluate the condition parts, then execute the action part of the first rule whose condition side matched (Single-Hit)

Or Evaluate all the condition parts then execute serially all the action parts of those rules whose conditions side matched (Multiple-Hit).

Parallel:

Perform context evaluation in parallel.

Evaluate all condition parts in parallel, and then, synchronize (that is, wait for all the condition side evaluations to complete) and choose one action part

Or synchronize and execute the actions serially (in lexical order)

Or execute the actions in parallel as the condition parts match.

Within Rules

Serial:

Evaluate predicate in the condition part, then execute each action.

Parallel:

Evaluate the predicates in the condition parts in parallel, then execute the actions in the action part in parallel.

4.3. Discussion of the Concurrent Components

We described the mechanisms for concurrency in Cage. We now discuss where and how these mechanisms can be used to gain speed-up.

4.3.1. Knowledge Source concurrency

Knowledge sources are logically independent partitions of the domain knowledge. A knowledge source is selected and executed when changes made to the blackboard are relevant to that knowledge source. Theoretically, many different knowledge sources can be executing at the same time. But, knowledge sources are often serially dependent, reflecting the reasoning process.

In the class of applications under consideration, the solution is built up in a pipeline-like fashion up the blackboard hierarchy. That is, the knowledge source dependencies form a chain from the knowledge sources working on the most detailed level of the blackboard to those working on the most abstract level. The implication is that knowledge sources can run in parallel along pipes formed by the blackboard data. (When the program is model-driven, this pipeline works in the reverse direction.)

There are two potential ways for knowledge sources to run in parallel: (1) knowledge sources working on different regions of the blackboard asynchronously (working on sub-problems in parallel) and (2) knowledge sources working in a pipelined fashion exploiting the flow of information up, or down, the data hierarchy (pipeline the reasoning). Both sources of parallelism are possible due to data parallelism inherent in the application.

4.3.2. Rule concurrency

Each knowledge source is composed of a number of rules. The condition parts of these rules are evaluated for a match with the current state of the solution, and the action parts of those rules that match the state are executed. The condition parts of all the rules in a knowledge source, being side-effect-free by design, can be evaluated concurrently without fear of unpleasant interactions. In cases where all the matched rules are to be executed (Multiple-Hit), the action parts can be executed as soon as the condition part is matched successfully. If only one of the rules is to be selected for execution (Single-Hit), the system must wait until all the condition parts are evaluated, and one rule, whose action part is to be executed, must be chosen.¹ The situation in which all of the rules are evaluated and executed concurrently potentially has the most parallelism. However, if the rules access the same blackboard data item, memory contention becomes a hidden point of serialization.

The asynchronous firing of rules is associated with two types of problem: timeliness and coherence. First, the state which triggered the rule may be

¹Refer to [Gupta 86] for the results of running OPS rules in parallel.

modified by the time the action part is executed. The question is then: is the action still relevant and correct? Second, if a rule accesses attributes from different blackboard objects, there is no guarantee that the values from the objects are consistent with respect to one another.

Condition-part concurrency: Each condition part of a rule may consist of a number of predicates to be evaluated. These predicates can often be evaluated concurrently. In the chosen class of applications these predicates frequently involve relatively large numeric computations, making parallel evaluation worthwhile. However, as discussed above, if the clauses refer to the same data item, memory contention would force a serialization, nullifying the apparent benefits of concurrent execution.

Action-part concurrency: Often, when a condition part matches, more than one potentially independent action is called for, and these can be executed in parallel.

The problem of data consistency occurs both in Cage and in Poligon. It can be partially alleviated by defining an atomic operation that includes both read and write on an object. This ensures that between the time that an item of data is read, processed, and the result stored, there is no change in the state of the object¹. For this to be possible there are two requirements: (1) all the data needed by the knowledge source is stored in an object and (2) a commitment is made about the granularity of the critical section — for example, "*read the data for the condition part of a rule and execute the action part.*" However, for most applications a knowledge source needs data stored in more than one node; and given the goal of the research, it is undesirable to commit to any particular process grain size. In order to enable experimentation with granularity, atomic actions in Cage are kept small and locks, block reads, and block writes are provided. Although an atomic block-read or -write operation does not solve the problems of timeliness or of global coherence, it does ensure that the data within each node is consistent. And, although locks have a potential for causing deadlocks, they are provided for the user to construct larger critical sections, for example, the object creation process is made atomic using locks.

4.3.3. Concurrency Control

The action parts of rules generate events, and knowledge sources are activated by the occurrences of these events. In the (serial) AGE system events are posted on a global event list and, based on the type of these events, a scheduler invokes one or more knowledge sources. In order to eliminate this serial control scheme, a mechanism to activate the relevant knowledge sources immediately upon event generation is needed. This immediate activation of knowledge sources still requires a scheduler in Cage, but it is very small, and, from a problem solving perspective, effectively eliminates global control. In some cases this is acceptable, but for those cases where a

¹In Lamina [Delagi 86b], another programming framework developed for the AAP project, the comparable atomic action is read-process-write.

more elaborate control is needed a centralized scheduler/control mechanism is provided. For instance, one mechanism allows the accumulation of events, after which all knowledge sources relevant to a subset of the events can be invoked in parallel.

Some answers to the many questions raised about concurrent problem solving are embedded in Cage's architecture. However, much of the burden is passed on to the applications programmer. Some useful programming techniques that were discovered are discussed below.

4.4. Programming with Cage

There are a number of problems that crop up in concurrent systems that do not appear in serial ones. The solutions to some of these problems involve reformulating the application problem; some involve the use of programming techniques not commonly used in serial systems. The techniques discussed below fall into the second category.

4.4.1. Pitfalls, Problems and Solutions

A need for the following programming techniques arose while implementing Elint (see Section 7) in Cage.

- When the only things to run in parallel are the knowledge sources, it is possible to read all the attributes of an object that are referenced in a knowledge source by locking the object once and reading all of the attributes. This is in contrast to locking the object every time an attribute is read by the rules. In other words, all necessary blackboard data is collected into local variables in the knowledge source's activation context before any rules are evaluated. This ensures that all the rules are looking at data from the same time.
- In a serial blackboard system one precondition may serve to describe several changes to the blackboard adequately. For example, suppose the firing of one rule causes three changes to be made serially. The last change, or event, is generally a sufficient precondition for the selection of the next knowledge source. In a concurrent system, however, all three events must be included in a knowledge source's precondition to ensure that all three changes have actually occurred before the knowledge source is executed.

In general, a simple precondition consisting of an event token is not sufficient as it would be in a serial system. A detailed specification of the activation requirements of the knowledge sources must be available, either in their preconditions or in the global scheduler.

- It is important for the programmer, when writing the condition parts of rules, to keep in mind the possibility of running the predicates concurrently. This involves keeping predicates from accessing the same data.

- Occasionally two knowledge sources running in parallel may attempt to change an attribute at almost the same time. It is possible that the first change would invalidate the later changes. To overcome this race condition, a conditional action — an action which checks the value of a slot before making a change — was added. An alternative solution to the race condition is to lock a node for an entire knowledge source execution, which would seriously limit parallelism.

5. Pursuing a Daemon-driven Blackboard System: Poligon

Control in the blackboard model can be summarized as follows: *knowledge sources respond opportunistically to changes in the blackboard.* As discussed earlier, in reality, and especially in serial systems, the blackboard changes are recorded and a control component decides which change to pursue next. In other words, the knowledge sources do not respond directly to changes on the blackboard. A central scheduler generally dictates the problem-solving behavior. This is a serial process.

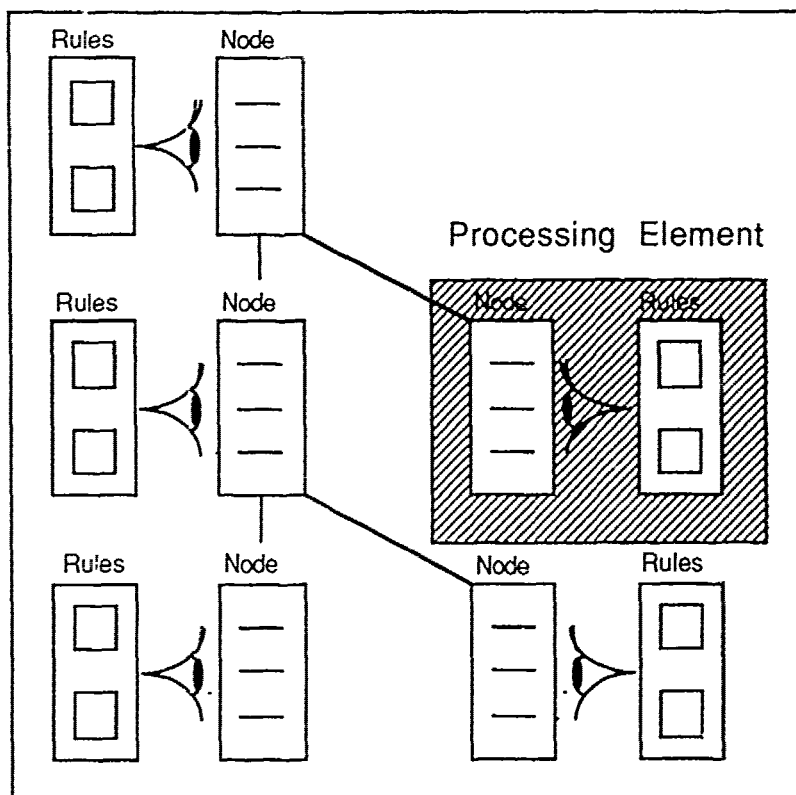


Figure 5.1. The Organization of Poligon

The basic question that led to the design of Poligon is: *What happens if you get rid of the scheduler?* Instead of waiting until a scheduler activates a knowledge source, why not execute the knowledge source immediately as the relevant data is changed by attaching the knowledge source to the data? A blackboard change can then serve as a direct trigger for knowledge source activations. To accomplish this, assign a processor-memory pair for

each blackboard object (called a *node* in Poligon), and have the knowledge sources (now on the blackboard processing element) communicate changes to other nodes by passing messages via a communication network. (see Figure 5.1).

Because a knowledge source is now activated directly by a blackboard change, and because a knowledge source is a collection of rules, one can view the rules as also being activated by that change. As a further refinement, a rule can be activated by a change to a particular slot (an attribute) of a blackboard node. If a change is made to a slot to which rules are attached, the condition parts of the triggered rules are evaluated; changes to other slots do not initiate any processing. Rule activation under these conditions are reminiscent of active values in object-oriented programming.

Poligon was designed from the start to exploit "medium"-grained parallelism; "medium" grain here referring to parts of rules, but not to small expressions. It is generally thought that in a shared-memory architecture performance gain levels off rather quickly as a result of physical limits in the bandwidths of the busses and switches connecting the processors and the memory. Thus, Poligon was designed from the start to be run on a form of distributed-memory multiprocessor. Because Poligon was designed for this form of hardware architecture, it differs considerably from existing serial implementations of blackboard systems.

5.1. The Structure of Poligon

In this section we describe the key features of Poligon. Instead of a detailed description of the implementation, a number of points which are central to Poligon's computational model are highlighted and contrasted with conventional blackboard implementations.

It should be noted that the user's cognitive model of the Poligon system and the system's implementation model are not necessarily closely connected. For instance, the Poligon system is implemented on an object-oriented substrate, which uses message passing to invoke methods. No sign of this message-passing behavior is visible to the user, who views the Poligon system very much like a conventional blackboard system.

As has been mentioned above, Poligon is designed to run on distributed-memory machines — hardware which provides message-passing primitives as the mechanism for communication between processing elements. It is important to note that the way in which information flows on the blackboard can be viewed, at an implementation level, as a message-passing process. This allows a tight coupling between the implementation of a system such as Poligon and the underlying hardware. It also allows the development of a computational model which views a blackboard node as a process, responsible for its own housekeeping and for processing messages.

- *Poligon has no centralized scheduler.* This was motivated by a desire to remove any bottlenecks that might be caused by multiple, asyn-

chronous processes trying to put events onto a single scheduler queue. (This problem is clearly manifested in Cage.) The elimination of a central scheduler requires a new knowledge invocation mechanism. *Poligon's rules are triggered as daemons by updates to slots in nodes.* The association between rules and the slots that trigger their invocation is made at compile-time, allowing efficient, concurrent invocation of all eligible rules after a blackboard event.

- When the centralized scheduler is eliminated, it also eliminates all global synchronization and any mechanism for focus of attention. This means that different parts of a Poligon program will run at different speeds, and each part will have a different idea of how the solution is progressing. The application writer is required not to make any assumptions concerning the global coherence or state of the solution.
- Having eliminated the scheduler, there is clearly no need for the separation of the knowledge sources from the blackboard. The Poligon programmer, therefore, specifies at compile-time the types of blackboard node with which a particular piece of knowledge is to be associated. At compile-time and at system initialization time, *knowledge is associated directly with the nodes on the blackboard that might invoke it.* In fact, when Poligon is running in its most optimized state the knowledge base is block-compiled and the rules are wired directly to the slots of the nodes, in which they are interested, eliminating all knowledge search.
- In conventional blackboard systems, knowledge sources are units of scheduling. If a system attempts to execute only its knowledge sources in parallel a great deal of potential parallelism will be lost by the failure to exploit parallelism at a finer grain. In Poligon, therefore, knowledge sources are simply collections of smaller pieces of knowledge in the form of rules. *All of the rules in a knowledge source can be executed in parallel.* Indeed, knowledge sources are compiled out by the Poligon compiler.
- Serial blackboard systems generally don't have a significant problem with the creation of new blackboard nodes. This is because of the atomic execution of knowledge sources. Such systems can usually be confident that, when a new node is created, no other node has been created that represents the same object. In parallel systems multiple, asynchronous attempts can be made to create nodes which are really intended to represent the same real-world object. Poligon provides mechanisms to allow the user to prevent this from happening.
- It is occasionally necessary to share data between a number of nodes. Since Poligon allows no global variables, it is necessary to find a way to define sharable, mutable data, while still trying to reduce the bottlenecks that can be caused by shared data structures. Poligon, like many frame systems, has a generalized class hierarchy with the

classes themselves being represented as user defined blackboard nodes. One way to view this is that the level structure on the blackboard is replaced by a class structure. The nodes belonging to a level are instances of a class. In addition, the classes nodes are active, serving as managers that create instance nodes. This level manager also stores data shared between all of the instances to support operations which apply to all members of a class. Shared data can therefore be implemented in a distributed manner by using slots on the level-manager (class) nodes.

- Most blackboard systems represent the slots of nodes simply as lists of values associated with the name of the slots. Because knowledge source executions are atomic in serial systems, programs can assume that no modification will have happened to the value list since it was read. In Poligon, because a large number of asynchronously running rules can be attempting to perform operations on the same slot simultaneously, a mechanism is needed to assure data consistency without slowing down the access to slots (a large critical section would reduce parallelism). *Poligon provides "smart" slots.* They are smart in the sense that they can have associated with them user defined behavior which can make sure that operations performed on the data leave the data consistent.

The problem of data consistency within a slot is reduced by the slot being able to determine cheaply and locally whether a modification is reasonable. Global solution coherency can be enhanced by the same process — slots can evaluate whether a modification will lead to a more precise solution. This causes a sort of distributed hill-climbing which helps the system evolve towards a coherent solution.

5.2. Shifting the Metaphor

Poligon's design looks very much like a frame-based program specialized for a particular implementation of the blackboard model. The expected behavior of the system is much closer to the blackboard problem-solving metaphor than serial systems, in one respect: the knowledge sources respond to changes in the blackboard *directly*¹. There are two major sources of concurrency in this scheme, which are similar to those in Cage:

- Each blackboard node can be active simultaneously to reflect data parallelism — the more blackboard nodes, the more potential parallelism.
- Rules attached to a node can be running on many different processing elements simultaneously providing knowledge parallelism.

¹As an historical note, this takes us back to Selfridge's Pandemonium [Selfridge 59], which influenced Newell's ideas of blackboard-like programs [Newell 62]. It also has some of the flavor of the actor formalism [Hewitt 73].

This daemon-driven system with a facility for exploiting both data and knowledge parallelism poses some serious problems, however. First, it is easy to keep the processors and communication network busy; the trick is to keep them busy converging toward a solution by doing useful work. Second, solutions to a problem will be non-deterministic, that is, each run will most likely produce different answers. Worse, a solution is not guaranteed since individual nodes cannot determine if the system is on the right path to an overall solution. That is, there is no global control to steer the problem solving. Within the AI field where we look for satisficing answers, non-determinism, *per se*, is not a cause for alarm. However, non-convergence to a solution or an incorrect solution is not acceptable.

One remedy to these problems is to introduce some global control mechanisms. Another solution is to develop a problem-solving scheme that can operate without a global view or global control. We have focussed our efforts in Poligon on the latter approach.

5.2.1. Distributed, Hierarchical Control

In Poligon, an hierarchical control mechanism is introduced that exploits the structure of the blackboard data. The level structure, in the AGE sense, of the blackboard are, as mentioned earlier, organized as a class hierarchy. Each level is a class and a blackboard object is an instance of that class. Class nodes, or level managers, contain information about their instances (number of instances, their addresses, and so on), and knowledges sources can be attached to level managers to control their instance nodes. Similarly, a super-manager node can control the level managers.

Within Poligon, the potential for control is located in three types of places:

1. Within each node, where action parts of the rules can be, though are generally not, executed serially. This is the only point at which the user can explicitly request serialization.
2. In the level manager which can, for example, be used to monitor the activities of its nodes. Since the level manager is the only agent that knows about the nodes on its level, a message that is to be sent to all the nodes on that level must be routed through their manager node. The level manager also controls the creation and garbage collection of the nodes and it attaches the relevant rules to newly created nodes.
3. In the super-manager, whose span of control includes the creation of level managers and their activities, and indirectly their offspring.

The introduction of these control mechanisms solves some of the difficulties, but it also introduces bottlenecks at points of control, for example, at the level manager nodes. One solution to this type of bottleneck is to replicate the nodes, that is, create many copies of the manager nodes. The CAOS experiments, mentioned earlier, took this approach [Brown 86]. Although Poligon supports this strategy, our research is leading us to try a different tactic.

5.2.2. A New Role for Expectation-driven Reasoning

It was initially conjectured that model-driven and expectation-driven processing would not play a significant role in concurrent systems, at least not from the standpoint of helping with speed-up. One view of top-down processing in serial systems is as a means of gaining efficiency in the following way: In the class of applications under consideration, the interpretation of data proceeds from the input data up an abstraction hierarchy. The amount of information being processed is reduced as it goes up the hierarchy. Expectations, posted from a higher level to a lower level, indicate data needed to support an existing hypothesis, data expected from predictions, and so on. Thus, when an expected event does occur, the bottom-up analysis need not continue up — the higher level node is merely notified of the event, and it does the necessary processing, for example, increases the confidence in its hypothesis. When the analysis involves a large search space, this expectation-driven approach can save a substantial amount of processing time in serial systems.

In Polygon hot-spots often occur at a node to which many lower level nodes communicate their results (a fan-in). The upward message traffic can be reduced by posting expectations on the lower level nodes and having them report back only when *unexpected* events occur. This approach, currently under investigation, is one way for a node to distribute parts of the work to lower level nodes, and hopefully relieves the type of bottlenecks caused by fan-ins at a node without resorting to node replication (which has its own management problems). This top-down expectation driven approach may turn out to be useful in relieving hot-spots.

It is generally expected that within the abstraction hierarchy of the blackboard, information volume is reduced as one goes up the hierarchy. This translates into the following desideratum for concurrent systems: For an arbitrary node to avoid being a hot-spot, there must be a decrease in the rate of communication proportional to the number of nodes communicating with it. That is, the wider the fan, the less communication is allowable from each node to the fan-in point. It was found while re-implementing the serial Elint application in Polygon, that the highest level nodes had to be updated for every new data item. Such a formulation of the problem, while posing no problem in serial systems, produces hot-spots and reduces parallelism in concurrent problem solvers.

5.2.3. A New Form of Rules

If, for any given data item, there are many rules that check its state, then the system must ensure that this data item does not change until all the rules have checked it. A typical example is as follows: Suppose there are two rules that are mutually exclusive, one performs some action if a data value is "on" and the other performs some other action if the value is "off." How can we ensure that between the time the first rule accesses the data and the second does so, there is no other action that changes the data? It was found in Polygon (and also in Cage) that these mutually exclusive rules

need to be written in the form of case-like conditionals (with the condition checking being atomic) to ensure a consistency. In these rules, at most one of the selectable action parts will be executed. Since the need for process creation and its maintenance is reduced by combining rules, this form of rule also helps to speed up overall rule execution. It does mean, however, that the grain size of the rules is generally bigger, at least at the source code level, and the programmers must think differently about rules than they do in current expert systems.

5.2.4. Agents with Objectives

At any given point in the computation, the data at different nodes can be mutually inconsistent or out of date. There are many causes for this, but one cause is that blackboard changes are communicated by messages and the message transit time is unpredictable. In the applications under consideration, where there are one or more streams of continuous input data, the problem appears as scrambled data arrival — the data may be out of temporal sequence or there may be holes in the data. Waiting for earlier data does not help, since there is no way to predict when that data might appear. Instead, the node must do the best it can with the information it has. At the same time, it must avoid propagating changes to other nodes if it has a low confidence in its output data or in its inferences.

Put another way, *each node must be able to compute with incomplete or incorrect data, and it must 'know' its objectives to enable it to evaluate the resulting computation. A result is passed on only if it is known to be an improvement over a past result.* This represents a change from the problem-solving strategies generally employed in blackboard systems where the controller/scheduler evaluates and directs the problem solving. With no global controller to evaluate the overall solution state and with asynchronous problem-solving nodes, a reasonable alternative is to make each node evaluate its own local state. Of course, there is no guarantee that the sum total of local correctness will yield global correctness. However, the organizations of blackboard applications seem to help in this matter. Blackboard systems are generally organized into sub-problems, and each blackboard level represents a class of intermediate solutions. The knowledge sources are functionally independent, and their span of knowledge is limited to a few levels. This type of problem decomposition creates subproblem nodes (with relevant knowledge sources) which can have local objectives and a capability for self-evaluation¹. The "smart" plots mentioned in Section 5.1 are used to implement this strategy.

¹It is interesting to note that the need for local goals does not seem to change with process granularity. Although the methods used to generate the goals are very different, Lesser's group has found that each node in its distributed system needs to have local goals [Durfee 85]. In this system each node contains a complete blackboard system; each system (node) monitors the activities in a region of a geographic area which is monitored collectively by the system as a whole.

The design of Poligon poses an interesting question — is it still a blackboard system? There is a substantial shift in the problem-solving behavior and in the way the knowledge sources need to be formulated. The structure of the solution is not globally accessible. There is no control mechanism to guide the problem solving at run time. The metaphor shifts to one in which each "blackboard" node is assigned a narrow objective to achieve, doing the best it can with the data passed to it, and passing on information only when the new solution is better than the last one. The collective action of the "smart" agents results in a satisficing solution to a problem¹.

Although there is a substantial shift away from the conventional implementation of the blackboard metaphor, Poligon evolved out of the mechanisms that were present in AGE. Most of the same opportunities for user-defined concurrency available in Cage are built into the system in Poligon. The Poligon language forces the user to think in terms of blackboard levels and knowledge sources. But the underlying system has no global data. Whether the divergence between the problem solving model (in the form of the Poligon language) and the computational model (in the form of its implementation in a particular form of hardware) makes the job of constructing concurrent, knowledge-based systems easier or more difficult for the knowledge engineer still remains to be seen. A difficulty might arise because the semantics of the Poligon language, that is, the mapping of the blackboard model to the underlying software and hardware architecture, is hidden from the user. For example, there is no notion of message-passing or of a distributed blackboard reflected in the Poligon language. In contrast, the choice of what, and how, to run concurrently is completely under user control in Cage.

6. The CARE Simulation System and Machine Architecture

CARE [Delagi 86a] is the name given both to the simulator used on the Advanced Architectures Project and to the hardware designs being developed on that simulator.

The CARE software system consists of a kit of components with which the user can construct simulated multiprocessors. The processor components and their behavior and interconnection topology are easily defined and specialized by the user. CARE allows experimentation with a large number of simulated machines each with a differing numbers of processors. In addition, a number of system parameters can be used to investigate the performance of different hardware variants. For the purposes of the Cage and Poligon experiments, system parameters were held constant while the number of processors was varied for each experiment.

¹In retrospect, these characteristics for concurrent problem solving seem obvious. When a group of humans solve a problem collectively by subdividing a task, we assume each person has the ability to evaluate his or her own performance relative to the assigned task. When there are "uncaring" people, the overall performance is bad, both in terms of speed and solution quality.

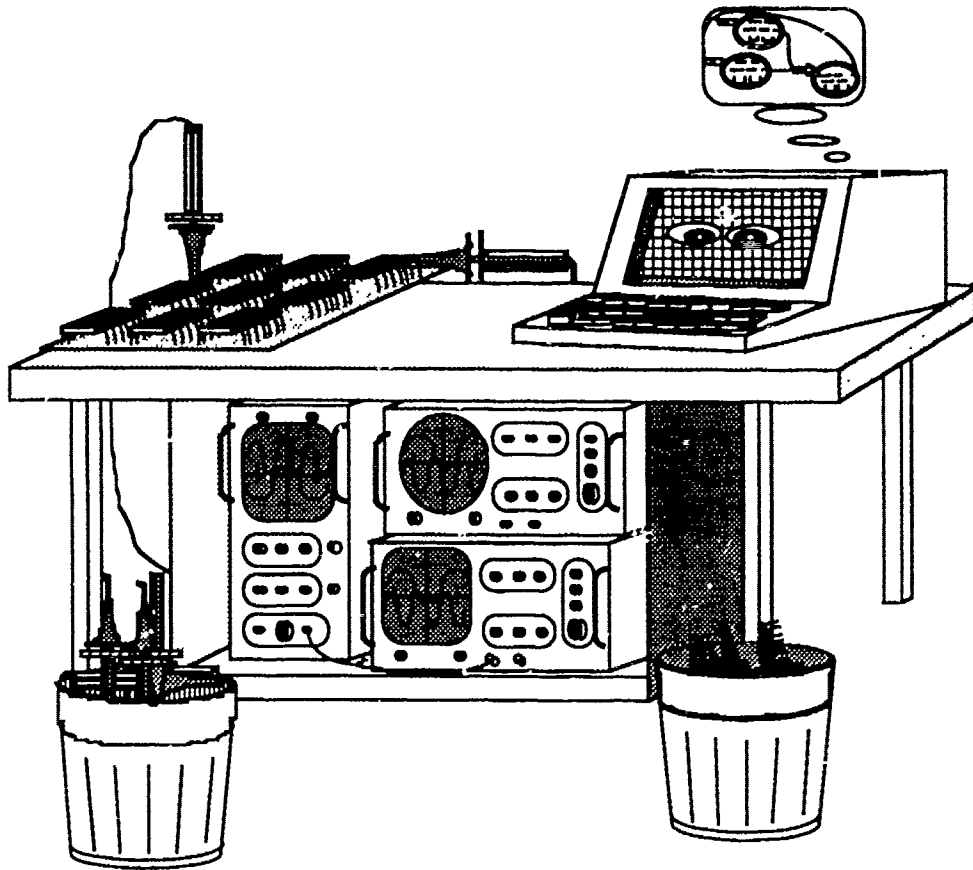


Figure 6.1. The CARE system's instrument toolkit. Collections of circuit components are wired to make multiprocessors. A variety of probes and instruments allows the flexible monitoring of both system and application.

One of the most important features of the CARE system is the instrumentation that it provides (see Figure 6.1). The user can plug simulated probes onto the simulated multiprocessor. These probes take various measurements and are connected to instruments that display the different system characteristics. The instrumentation toolkit allows the user to watch the behavior of the system both from the point of view of hardware performance and the application program. This, for instance, allows the identification of bottlenecks and hot-spots during system execution. An example of a CARE instrument is shown in Figure 6.2.

The CARE machine architecture will not be discussed in any detail in this paper, but some elucidation at this point should allow better understanding of the references made to the underlying hardware in forthcoming sections.

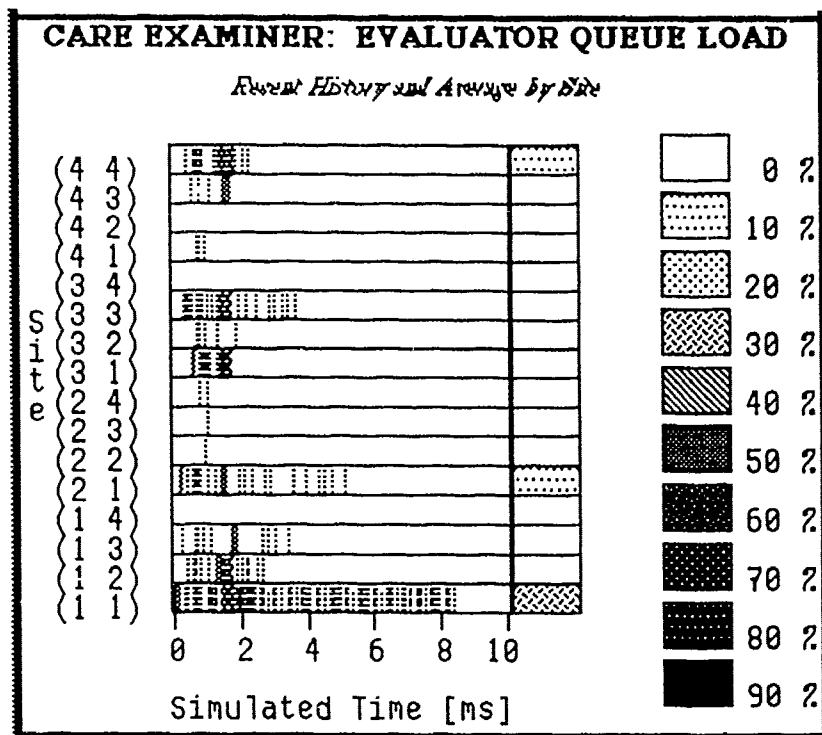


Figure 6.2. One of the instruments provided by the CARE system. This one shows the lengths of the Evaluator's process queue over time for each processing element (Site)

Each processing element in the CARE machine is made up of two processors; one the *Operator*, whose purpose is to execute operating system functions and to perform the task of inter-processor communication, and the *Evaluator*, whose task is the execution of user code (see Figure 6.3). This design allows the user application to continue with its work, while communication is taking place.

The communication behavior of the simulated hardware used by Poligon and that used by Cage are different. In the Poligon system the CARE simulator is used to simulate an array of the processing elements, connected in a toroidal manner, such that each processing element can talk to its eight neighbors (up/down, left/right and diagonal).

The Cage system uses an array wired in a similar manner, but in this case half of the processing elements in the array are specialized to act solely as memory controllers/servers, that is the Evaluators are not used. This scheme combined with the dynamic, cut-through routing communication protocol [Byrd 87] used by Cage ensures that each of the processors executing user code has equal access to the memory-only processing elements. In this way Cage uses the distributed CARE architecture to simulate a shared-memory machine.

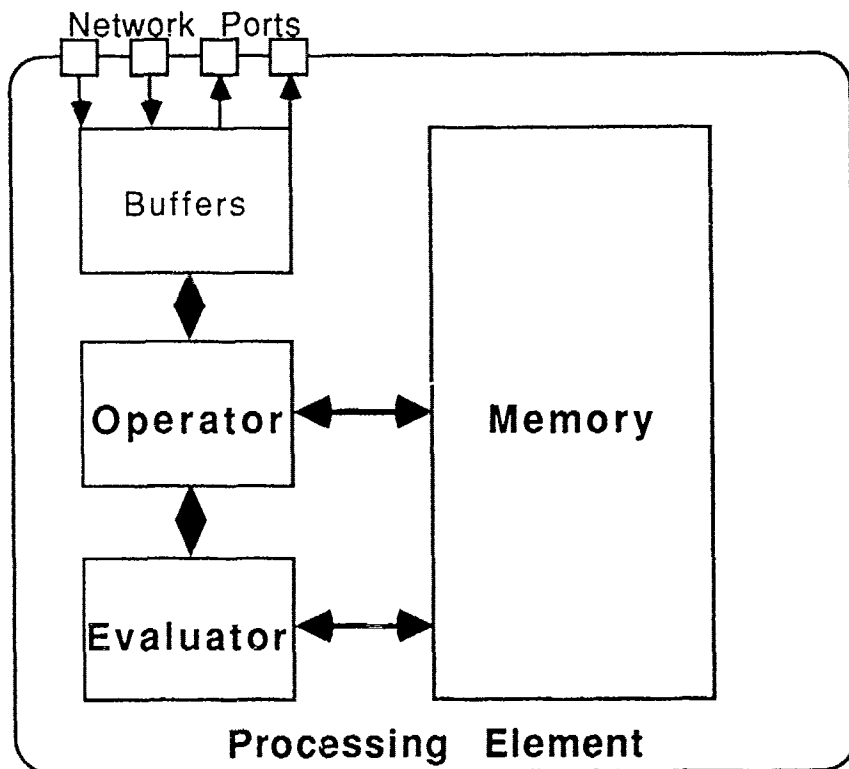


Figure 6.3. The CARE machine processing element

7. The Elint Application

The Elint Application is a situation understanding application used in our experiments. It is, in fact, part of a larger signal understanding system called Tricero, developed by ESL [Williams 84]. It was selected as an application, not only because it was in the problem domain in which we were interested, but also because it is a system of moderate complexity. It is complex enough to stress our software architectures and to give us a reasonable understanding of the problems of implementing real concurrent blackboard applications, yet simple enough that we could concentrate on the development of Cage and Poligon, rather than the application itself.

The Elint task is to integrate reports from multiple, geographically distributed, passive radar collection sites in order to develop an understanding of the position and intentions of aircraft travelling through the monitored airspace (see Figure 7.1). As the aircraft travel, they use a number of different radar systems for such tasks as ground tracking/altimetry and target acquisition and tracking. The passive radar receivers in the Elint application are able to detect the bearing of the radar emissions (the position of the emitter must be deduced from more than one radar system, since these are passive devices) and the type of radar system which is making the emissions.

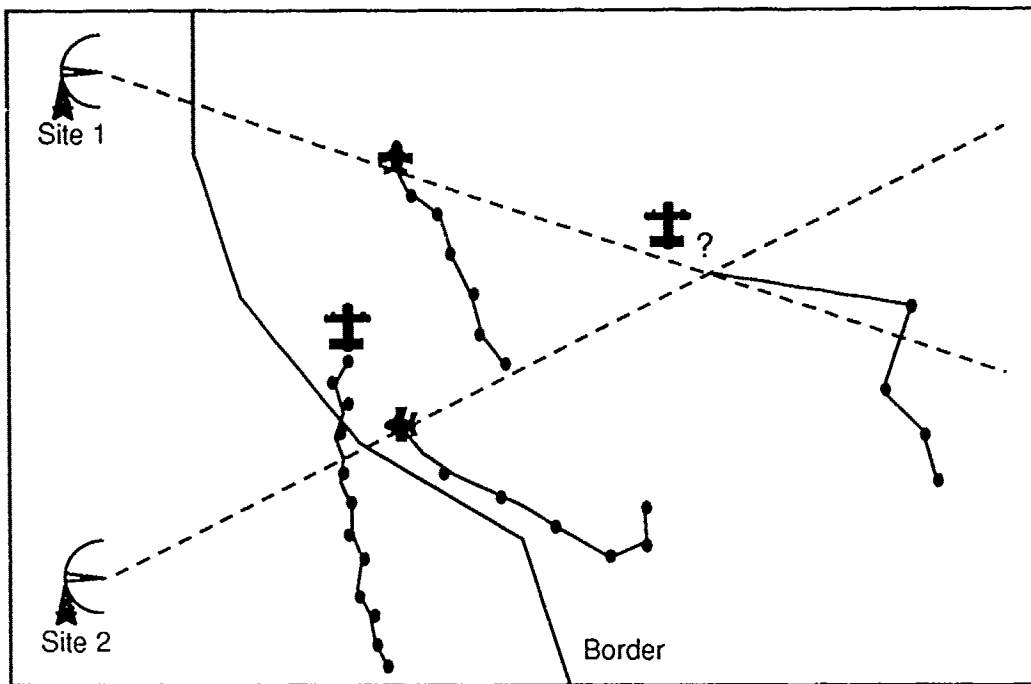


Figure 7.1. The Elint problem. The radar collection sites must track the aircraft using the bearings of the received emissions. In this case the system must distinguish between the real positions of the aircraft and positions which are impossible.

The application takes the multiple streams of reports from the collection sites, abstracts them into hypothetical radar emitters (perhaps aircraft), and tracks them as they travel through the monitored airspace. These emitters are themselves abstracted into clusters (perhaps formations of aircraft or single aircraft using multiple radar systems), which are themselves tracked (see Figure 7.2). Sometimes an aircraft in a cluster would split off, forcing the splitting of the representation of the cluster and rationalization of the supporting evidence. The nature of the radar emissions from the aircraft are also used to determine the intentions and degree of threat of each of the clusters.

The Elint application has a number of characteristics which are of significance.

- The system must be able to deal with a continuous input data stream, and there is a need for real-time processing. (The Elint application on both Cage and Poligon is a *soft* real-time application, processing continuous input data as fast as possible. It is not a *hard* real-time application, since it does not guarantee any specific response time.)
- The application domain is potentially very data parallel. The ability to reason about a large number of aircraft simultaneously is very important.

- The aircraft themselves, as objects in the solution space, are loosely coupled.

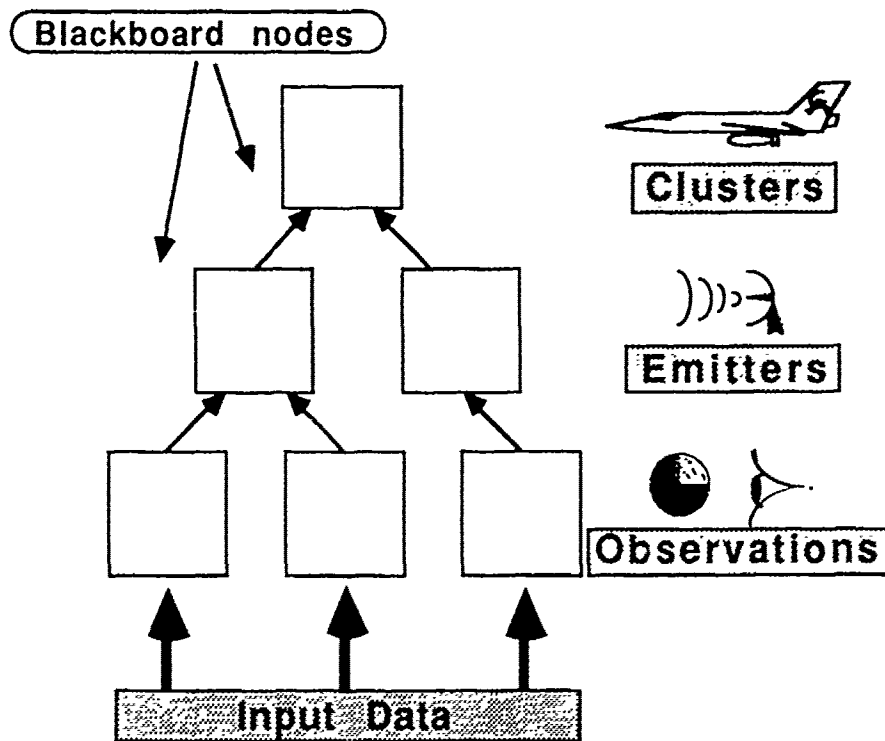


Figure 7.2. The Elint Application. Sensor data is abstracted into hypothetical radar emitters, which are tracked as clusters of emitters.

An artifact of the application, which should be well understood, is the idea of an input data sampling interval. Since the Elint application is, in some sense, a simulation of the real world, it has a clock of its own which ticks at a constant rate with respect to the time in the real world. The data that comes into the system is timestamped. When the application's clock has reached a time which is the same as the timestamp on the input data record, the data is introduced into the system. The simulated time between two of these ticks can in certain circumstances be used to provide a measure of the throughput of the system. Thus, the tick interval is a parameter that can be varied to measure the system's potential throughput. For any given experiment the input data that was being used defines the number of radar emissions detected in that timeslice. From here on in this paper, therefore, we will use the term *timeslice* to indicate a period whose length is equal to one domain clock tick and *input data sampling interval* or *sampling interval*, for short, as the length of one timeslice in simulated time. The sampling interval will typically be quoted in simulated milli or microseconds¹.

¹This is one aspect in which the Elint application is not realistic. In the real world, reports arrive at Elint data collection sites at a rate of one every few seconds. In order to stress our systems we had to turn up this rate until reports were arriving, in the case of Poligon, every 300 microseconds.

8. Experiments and Results

In this section we describe the experiments performed on the Poligon and Cage systems using the Elint application described in the previous section. We explain the reasons for performing the experiments, present the results of the experiments in detail, and draw conclusions from them.

It should be noted that the experiments mentioned here do not represent all of the experiments performed, but are simply those which we performed after learning from earlier ones. The earlier experiments taught us how better to perform the experiments and helped us to find numerous infelicities in both the Poligon and Cage systems and their respective Elint implementations.

8.1. Understanding the Graphs

In the following sections a large number of graphs will be shown, most of which will have the same format. The graphs plot either speed-up or both speed-up and input data sampling interval against the number of processors on which the experiment was performed. When both sampling interval and speed-up are plotted on the same graph, the sampling interval will always be labelled on the left Y-axis and the speed-up on the right Y-axis. A typical speed-up graph is shown in Figure 8.1.

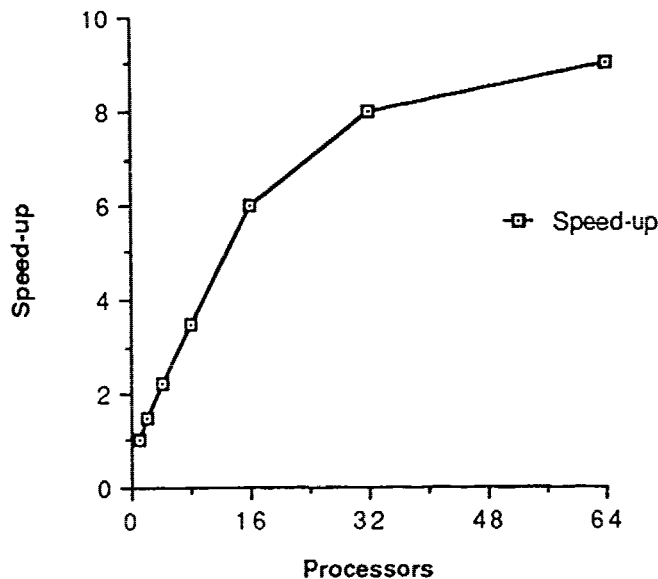


Figure 8.1. An example showing speed-up plotted against the number of processors used.

In the best of all possible worlds, linear speed-up would result; that is, for each new processor that was added, the speed-up would increase linearly. The plot would be a straight line. In practice, however, the amount of realizable speed-up often tails off as the number of processors increases, giving the characteristic shape for the curve in Figure 8.1.

For the sake of completeness, the sampling interval will often be shown along with the speed-up. The speed-up is, in fact, simply calculated by dividing the sampling interval for the uniprocessor case by that for the N processor case. The display of the sampling interval shows how the system's throughput is affected by the number of processors. This is typically a decreasing curve as is shown in Figure 8.2 — the system speeding up as the sampling interval is going down.

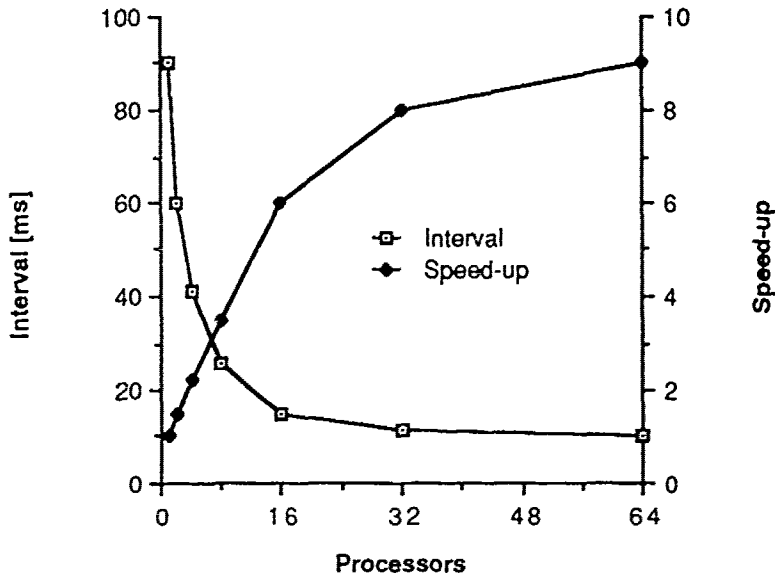


Figure 8.2. An example showing sampling interval and speed-up plotted against the number of processors used.

8.2. Experimental Method

An important part of these experiments is the method used to derive the measurements. Extensive experimentation was conducted before a method evolved, which could both define and measure the speed-up of these systems.

A simplistic method for measuring the speed-up of a parallel system would be to take the run-time for the application on a uniprocessor and then divide it by the run-time measured for different numbers of processors. This approach works well for non-real-time systems in which the behavior of the system is not affected by the speed of the computation. In a real-time system with continuous streams of input data, however, the behavior of the system changes according to the degree to which the system is loaded. For example if more processors are added to a system it can become data starved, failing to deliver the speed-up of which it is capable.

To counter this phenomenon a different methodology was devised. A series of experiments is performed, during which the input data sampling interval is established such that on the largest processor network size the system

is never data starved. The speed-up is measured using this sampling interval for other processor configurations, knowing that the delivered speed-up for the large multiprocessor configuration would not be data starved. We found, however, that with all system parameters held constant (except for the number of processors) the application program was still behaving differently for the different experiments. This was because for small numbers of processors the system was getting backed up, and it was spending a significant amount of time queue-thrashing. That is, it was trying to keep data in order which, if the system had not been so overloaded, would not have got out of order in the first place. This had the effect of making the application seem to run slower on smaller numbers of processors, thus giving an artificially high apparent speed-up.

What was needed, therefore, was a method for measuring the system's speed-up, while making sure that the system was always operating under the same load conditions. To accomplish this, the speed of the application on any particular processor configuration is defined as the lowest sampling interval (i.e. highest throughput) that still gives non-increasing latencies in the results. The latency measure is defined to be the time between the data coming into the system and the system emitting any reports concluded from the data.

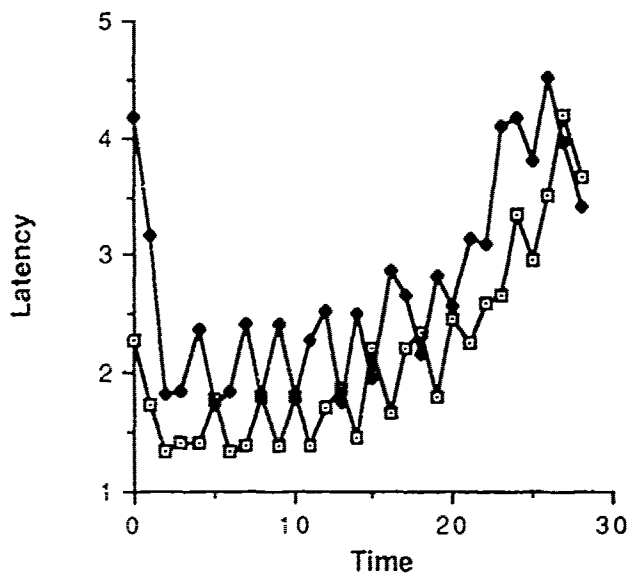


Figure 8.3. Showing increasing measured latency over domain time.

Examples of increasing and constant latencies are shown in Figures 8.3 and 8.4. If the system can keep up with the sampling interval specified, the latency value should be largely constant, otherwise latencies increase over time as the system backs up.

In summary, the following method is used to measure the system's speed: for any given number of processors, the application is run with different sampling intervals until one is found that produces non-increasing latencies. This sampling interval defines the processing speed for a given processor configuration. For a speed-up experiment, the above process is repeated for different processor configurations until the speed-up curve levels off.

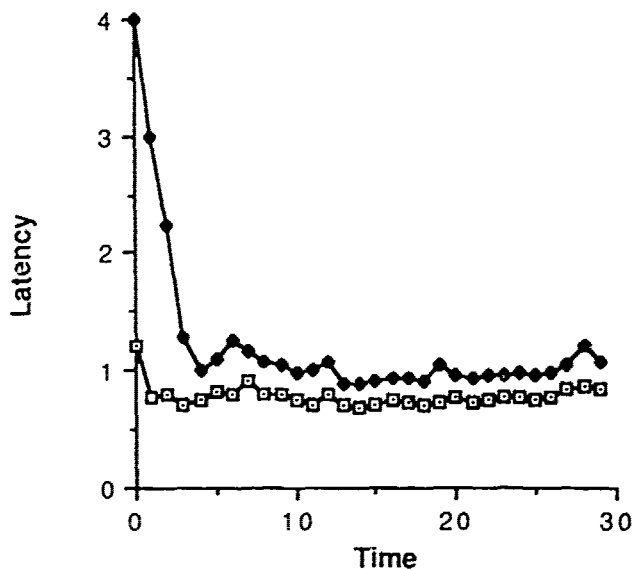


Figure 8.4. Showing constant measured latency over domain time.

8.3. Data Sets

An important aspect of the experiments on the Elint application is the scenario used to drive the experiment. A scenario represents the simulated radar information that a "real" system would have received. In a real system, one would expect that the number of received radar emissions would vary over time. Although realistic, this sort of scenario is very hard to perform experiments on, since there are bound to be times when the system is either data starved or overloaded. Because of this, two of the data sets used for the experiments have the particular property that they have a constant density of input data over time.

The important characteristics of these data sets, therefore, are the number of radar emissions detected in each time unit, the number of radar emitters, and the number of clusters (see Section 7).

It should be noted that these data sets are used to measure the overall peak system performance for a given data set having the characteristics mentioned below. The system's response to transients in the amount of input data in a timeslice was not measured, nor was its performance for input

data with less 'typical' characteristics; for instance, a small number of aircraft, each using a large number of radar systems, or a large number of aircraft, each using very few radar systems.

The characteristics of the three data sets used are enumerated below.

Fat 240 Observations, 4 Emitters, 2 Clusters, 8 Observations per time-slice, 30 time-slices, 2 Observations per Emitter per time slice.

Thin 60 Observations, 1 Emitters, 1 Clusters, 2 Observations per time-slice, 30 time-slices, 2 Observations per Emitter per time slice.

Lumpy 186 Observations, 12 Emitters, 1 Cluster (which splits), 2 inconsistent observations, 1 ID error, variable number of Observations per time-slice, 30 time-slices.

In the description of the experiments to follow, these data sets will be referred to by the names "*Thin*", "*Fat*" or "*Lumpy*" to save time, space and confusion.

8.4. Experiments with the Cage System

Seven separate sets of experiments, labeled C-1 to C-7, were run using the Elint application on Cage. The objectives of the first two sets of experiments were to compare Cage with Poligon, and to evaluate the efficiency and efficacy of the blackboard model for parallel execution. The third experiment compared different process granularity within Cage. The last four experiments were attempts to improve the performance of C-3 with different resource allocation schemes, more available processors, and a more efficient and accurate underlying simulator.

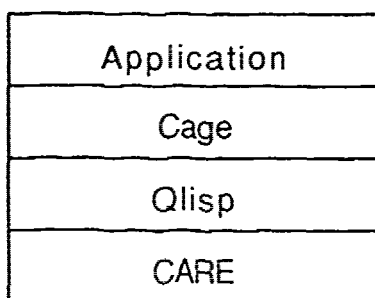


Figure 8.5. System Structure.

Cage is simulated on CARE as described in Section 6. In addition, Cage uses Qlisp [Gabriel 84], a queue-based multi-processing Lisp, which provides parallel evaluation of Let expressions and Lambda closures (see Figure 8.5). Each processor in the simulation is a multi-process machine.

Processes are assigned to available processors by a simple, non-preemptive round-robin scheme.

8.4.1. Experiment C-1: Basic Speed-up

8.4.1.1. Description

Experiment C-1 simply measures the speed-up attainable for a varying numbers of processors. For this experiment the scheduler started many knowledge source executions in parallel, waiting until they were done before selecting another set to run in parallel. Using a mixed data set with clusters, splits, inconsistencies, and id errors (the "Lumpy" data set) this experiment exercised all the problem solving capabilities in the Elint application. Experiment C-1 was run serially on one processor and on multiprocessors ranging from 2 to 16 processors. By comparing the time required to run the data set on one processor with the time required to run with the multiprocessors, a measure of speed-up was obtained. This is the simplistic speed-up measurement described in Section 8.2.

8.4.1.2. Purpose

The main purpose of this first experiment is to get a base-line speed-up measurement for a simple concurrency configuration. The concurrency options used were concurrent knowledge source execution with synchronization control. This measurement can be used as a basis for comparing the performance of the system using more complex concurrency configurations.

8.4.1.3. Results

The results of this first experiment are shown in Table 8-1 and Figure 8.6.

Processors	Speed-up at 20 ms Sampling Interval	Speed-up at 80 ms Sampling Interval
1	1	1
2	0.9	1.5
4	1.7	1.8
8	2.4	1.96
16	—	2.03

Table 8-1. The results of Experiment C-1.

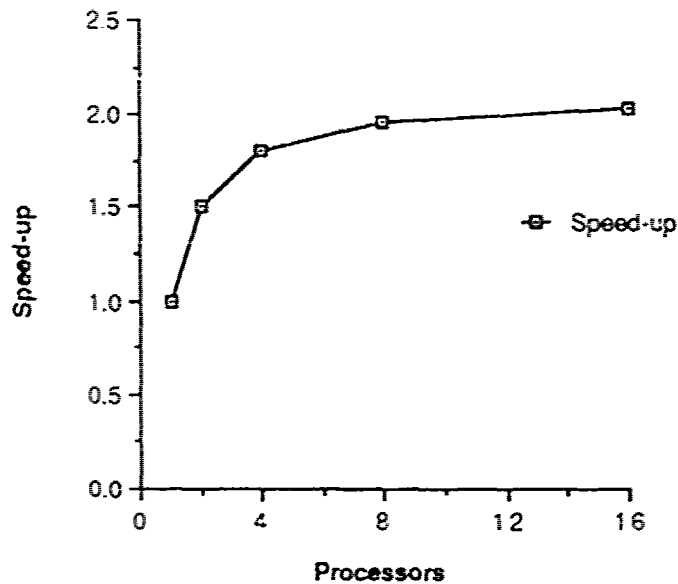


Figure 8.6. Speed-up derived in Experiment C-1.

8.4.1.4. Interpretation

The basic speed-up began to level off with 4 processors and reached a factor of 2 with 8 processors. To explain why only a factor of two speed-up was achieved, we need to look at the control cycle of a serial case. In the serial case (see Figure 8.7) the scheduler selects one knowledge source to execute from among all the knowledge sources applicable at that time.



Figure 8.7. Basic Control Cycle for Serial Execution.

In Experiment C-1 all the pending knowledge sources are executed in parallel, as seen in Figure 8.8.

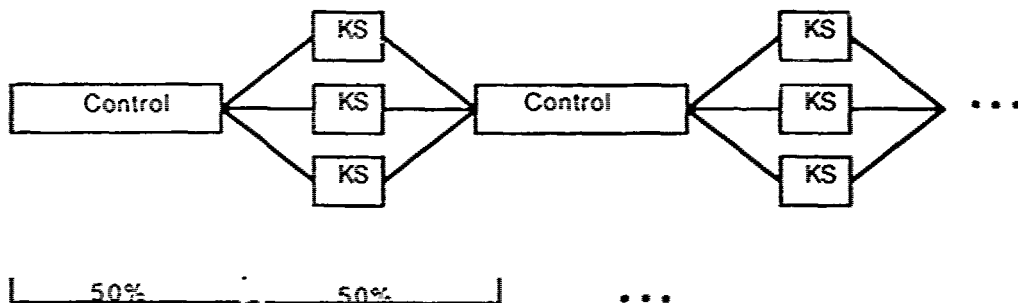


Figure 8.8. Control Cycle with Serial Control and Parallel Knowledge Sources.

Although the knowledge sources were run in parallel "Amdahl's limit" limits the speed-up to the longest serial component, in this case the scheduler plus the longest knowledge source. When all component parts of the

execution were individually timed, it was found that slightly less than half of the total execution cycle time was being spent in the serial, synchronizing scheduler.

Experiment C-1 demonstrates that when knowledge source invocation is synchronized, speed-up gains are limited by the combined grain size of the scheduler and the largest knowledge source, no matter how many knowledge sources are run in parallel. It should be noted, however, that the grain size of the knowledge sources, as well as that of the scheduler, is very application dependent. In the following experiment knowledge sources were executed in parallel without synchronization, but knowledge sources were still invoked by a central scheduler.

8.4.2. Experiment C-2: Speed-up Measurement using a Smooth Data Set

8.4.2.1. Description

The second experiment also measured speed-up, but in a manner that was felt to be more *fair* than the basic speed-up experiment, as explained in Section 8.2. Experiment C-2 was run with 1, 4, and 8 processors. In Experiment C-2 the knowledge sources were executed without synchronization, reducing the time spent waiting within the scheduler. As each knowledge source completed, the scheduler immediately invoked successor knowledge sources without waiting for any other knowledge sources to finish.

8.4.2.2. Purpose

The purpose of this experiment was to see if eliminating synchronization results in improved speed-up. Experiment C-2 also provides standardized measurements of speed-up and throughput to compare with results from Polygon, as was mentioned in the Introduction. This and subsequent experiments used the "rat" data set.

8.4.2.3. Results

Processors	Sampling Interval [ms]	Speed-up
1	700	1
4	225	3.11
8	180	3.89

Table 8-2. The results of Experiment C-2.

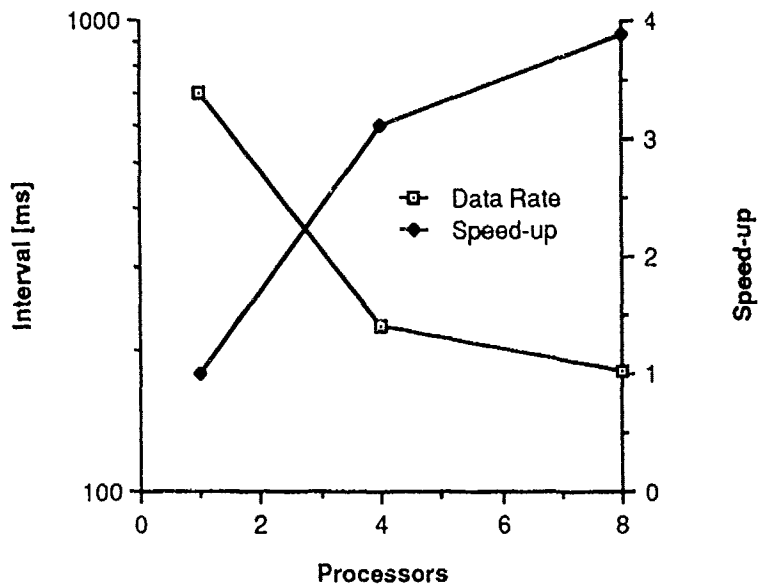


Figure 8.9. Results of Experiment C-2.

8.4.2.4. Interpretation

The *Ca_g* implementation of Elint has six different knowledge sources. During run-time many copies of the knowledge sources can be running concurrently. Theoretically, if the pipelines formed between the levels on the blackboard are well-balanced, no overhead for process creation or process switching is incurred, and the scheduler takes zero time (all of which are impossible), a speed-up of $1 + P(\#KSs)$ should be possible with knowledge source concurrency, where P is the number of pipes and $\#KSs$ is the number of knowledge sources in each pipeline. The *Fat* data set allows the creation of four pipes, so the maximum speed-up that is theoretically possible in this case is 25x.

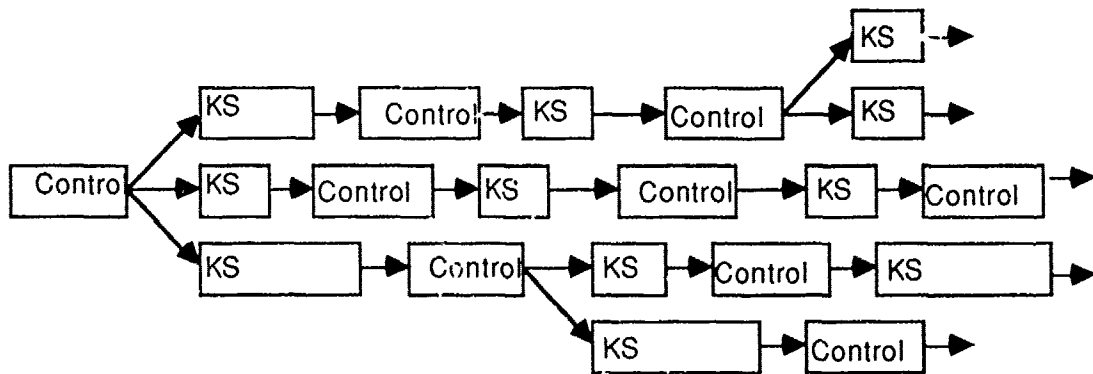


Figure 8.10. Logical View of Unsynchronized Knowledge Source Invocation.

The speed-up obtained by running knowledge sources concurrently without synchronizing was slightly less than 4. This is almost double the speed-up obtained with synchronization, though some of this difference will have been caused by the smoother data set used. The time spent in the scheduler

was reduced by almost half of that in Experiment C-1. However, it should be noted that the central scheduler is still a bottleneck. (See Figure 8.10) Given the architecture of blackboard systems, the time spent in the scheduler can be reduced, but not eliminated, without a major shift in the way we view blackboard systems. Polygon is one such shift.

One important result of Experiment C-2 was to confirm the ease with which different concurrency options (see Section 4.2) could be used in Cage. Once the Elint application was running with parallel knowledge sources and synchronization, only one minor change to one rule was required to make it execute parallel knowledge sources without synchronization correctly. No change to Cage itself is required when changing the concurrency specifications.

8.4.3. Experiment C-3: Asynchronous Rules

8.4.3.1. Description

In Experiment C-2 all possible concurrency at the knowledge source level was exploited. Experiment C-3 attempted to increase the speed-up by exploiting parallelism at a finer granularity. We hoped to gain an increase in the overall speed-up for each knowledge source by executing the rules of each knowledge source in parallel. There are several options in Cage for executing rules in parallel and we selected those that we expected to yield the most speed-up. The rules were executed with both condition and action parts running concurrently and without synchronizing between the condition and action parts. Otherwise the experimental variables of Experiment C-3 are identical to those of Experiment C-2 — the same data set, sampling intervals, and number of processors.

8.4.3.2. Purpose

The purpose of Experiment C-3 was to measure speed-up with process granularity at the level of rules.

8.4.3.3. Results

Processors	Speed-up over Experiment C-2	Total Speed-up
1	1	1
4	-6%	2.92
8	5.8%	4.12
16	n/a	5.6

Table 8-3. The results of Experiment C-3.

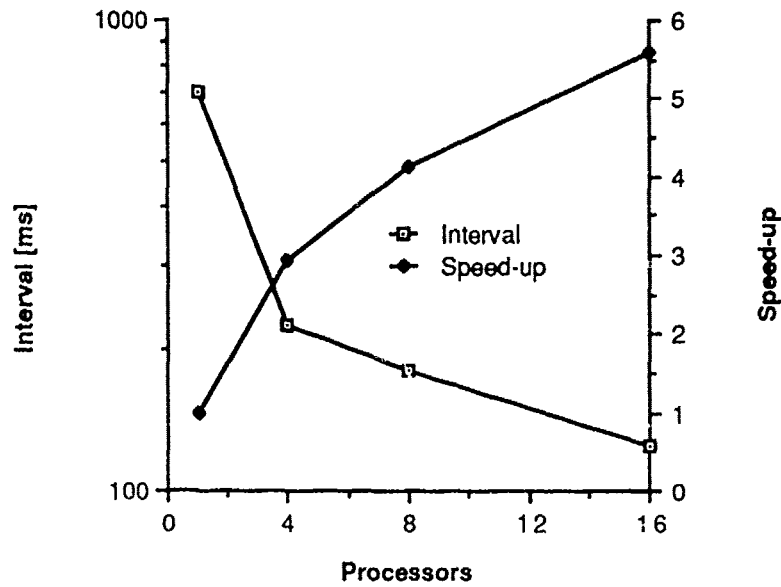


Figure 8.11. Experiment C-3.

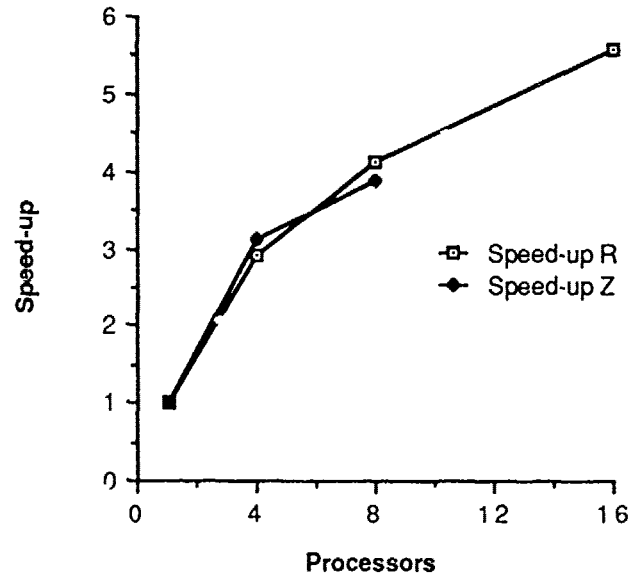


Figure 8.12. Experiments C-2 and C-3.

8.4.3.4. Interpretation

The results of Experiment C-3 were disappointing. For 8 processors only a 5.5% speed-up over Experiment C-2 was attained, for a total speed-up of 4.12. For 4 processors there was no speed-up at all over Experiment C-2. The overhead of spawning processes offset any gains from more parallelism.

There are several reasons for the small improvement in speed-up. In the Cage implementation of Elint, there are an average of 3.5 rules per knowledge source. Thus the maximum speed-up possible in Experiment C-3 is limited to 3.5 of the time spent executing knowledge sources in Experiment

C-2. A detailed time trace revealed that in Experiment C-2 only 26% of the execution time of a knowledge source was expended on actually running the rules. A factor of 3.5 speed-up of only 26% of the total execution time, can result in at most a total gain of 7.4% over the knowledge source execution time.

Number of processors	4	8
Change in speed-up from C-2 to C-3	-6%	5.5%

Table 8-4. The change in speed-up of the Experiment C-3 results over those of Experiment C-2.

These results are all for 8 processors. For 4 processors the gain was negated by the overhead cost of process spawning and resource allocation. The cost of spawning a rule was approximately the same as that for spawning a knowledge source.

As a result of these disappointing results, we ran Experiment C-3 on a 16 processor system in hopes of alleviating the congestion on the smaller grids. This resulted in slightly better results, a total speed-up of 5.6. This extra speed-up is due to the greater availability of free processors to handle the greater number of processes with rule level granularity.

8.4.4. Where Time is Being Spent in Cage

Throughout the Cage experiments with Elint, we had been troubled that the throughput Cage could achieve was low relative to Poligon. The best sampling intervals for Cage up to this point were around 120ms, while Poligon showed best sampling intervals in the order of a few milliseconds. In this section we explore the causes for the poor throughput.

8.4.4.1. Cage time Measurements

During the experiments all the component parts of Cage were timed. In addition, timings for various parts of Qlisp were also taken. Figure 8.13 shows the average times taken for the basic components of the Cage system to process one data point in one time interval.

As expected, most of the time was being spent setting-up and executing knowledge sources. Table 8-5 shows a breakdown of the time spent within a knowledge source. The times are averages for an entire simulation of the "Fat" data set, with 16 processors for Experiment C-3.

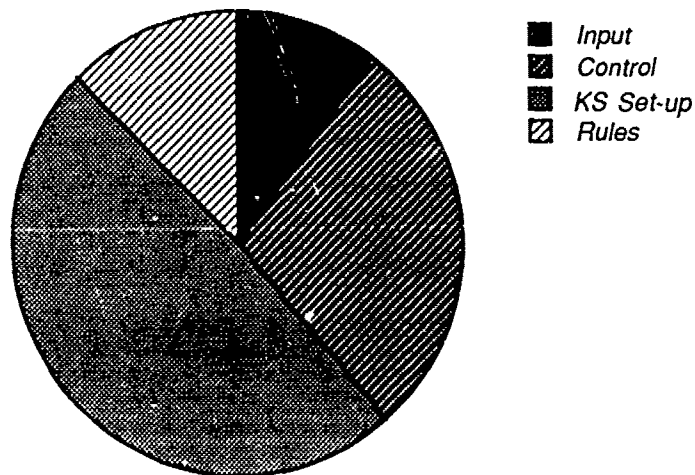


Figure 8.13. Average System Usage per Input Data Point.

Knowledge Source	Average [ms]	High [ms]	Low [ms]
Wait and start-up time	2.48	71.84	0.45
Instantiation	0.62	8.94	0.12
Definitions	27.52	184.92	0.90
Creation	17.56	123.90	0.95
Node Create	1.97	2.97	1.06
Match existing nodes	2.29	7.96	0.16
QLisp	15.23	120.67	0.95
Slot Reads	0.97	11.37	0.06
Rule wait and start-up	4.39	79.58	0.51
Rule execution	3.22	82.88	0.00 ¹
Total KS Execution time	28.45	186.09	1.19

Table 8-5. Time distribution for Typical Knowledge Source Execution¹.

8.4.4.2. Time Utilization

8.4.5. Experiment C-4: Process Allocation

While the averages in the table in Table 8-5 are interesting, pointing out obvious places that need to be remedied, they do not tell the entire story. The first trace files showed timings that were very spiky. For example, while the average time for a knowledge source instantiation is 0.62ms, there were times when instantiation took as long as 8.94ms and other times when it only took 0.12ms. An initial explanation was that the spikes were caused by

¹The lowest time for rule execution was too short to register accurately on Care's clock which has an error margin of 30 us.

blocked and descheduled processes.¹ This is an indication of problems in resource allocation.

If the spikiness is due to competition between processes for processors, then we should see an increase in speed-up and a decrease in the spikiness by increasing the number of processors available to run the processes. An earlier C-3 experiment did show a significant improvement in speed-up between 8 to 16 processors and a reduction in spikiness. To prevent processes from being descheduled when they block, a variation on Experiment C-3 was run. Certain processes were pre-allocated to specific processors which were made unavailable for other processes; that is, some processors became special purpose processors. The pre-allocated processes were the input handler, the node creation/match handler, and the scheduler — all of which are large and used often. This experiment was run only on a 16-processor system, the minimum amount of hardware deemed necessary for this configuration.

The results of this experiment were not conclusive.

Number of processors	16
Speed-up for C-3	5.6
Speed-up with allocation	5.7
Increase	3%

Table 8-6. Performance improvement over Experiment C-3 by specialized process allocation.

The increase in speed-up of 3% (see Table 8-6) falls within the margin of error for these experiments and is not significant. However, while speed-up was not significant there was a reduction in the spikiness observed in the traces. The highs in Table 8-5 were reduced in every case, with an average decrease of 5.4ms, or 8%. However, the queue lengths for knowledge sources and node creation/match increased, indicating that (1) insufficient numbers of processors were available for the knowledge sources because of the three pre-allocated processors and (2) the node creation/match handler probably needed two or more processors to handle its load.

8.4.6. Experiment C-5: Process Allocation

A second experiment involving specialized processor allocation was more successful. In this case only one processor, the input-handler, was used to execute the entire input procedure. Previously the creation of new input

¹Qlisp will deschedule a blocked process by placing the blocked process on the local processor queue and running the next process on that queue if there are other processes waiting on that queue. The blocked process must then wait for the new process to finish before the blocked process can resume.

nodes (objects on the observation level), one for each input data item, had been handled by a separate creation handler. By eliminating the cost of spawning the separate creation process, and with it the possibility of blocking the input process while waiting for the creation to complete, the input node creation time was decreased by 59%. Also, spikiness in the creation measurement almost disappeared.

Experiment	Average	High	Low
Input node creation on different processor [C-3]	15.39	91.54	5.05
Input node creation on input processor	4.87	6.78	4.23

Table 5-7. Results of Experiment C-5, variation on Experiment C-3.

Although tuning the application by pre-allocating some processors for special purposes does gain some speed-up, an easier solution may be to use more processors. The last two experiments test this hypothesis by using an additional 16 processors, 32 in all.

8.4.7. Experiment C-6: Multiple Node Creators

Experiment	ms	% over Exp C-5
Experiment 5 Single creation processor	40	n/a
Experiment 6 Multiple creation processors	31	22%
Experiment 7 Local creation	25	37%

Table 8-8. Throughput Results of Experiments C-5, C-6 and C-7.

In this and the final experiment the number of available processors was increased to determine if an insufficient number of processors was limiting the throughput. In this experiment the number of node creation process handlers was also increased from 1 to 4 in an attempt to break-up the node creation bottleneck. A major disadvantage of using more than one processor for creation is the possibility of two processors creating the same node at the same time. In order to bypass this problem, we dedicated a processor for each level of the blackboard to create its nodes. By tying the creation processes to individual blackboard levels, we avoided the problem of duplicate nodes. In preliminary runs of experiment C-6 we found that the addition of 16 processors, without specialized allocation of those processors, resulted in a negligible improvement in throughput. However, the allocation

of some of those new processors specifically for node creation handling resulted in a 22% improvement in throughput over the best results of Experiment C-5, as shown in Table 8-8.

8.4.8. Experiment C-7: Local Node Creation

In the final experiment we tried to eliminate the node creation bottleneck completely by doing all creation on the local processor, instead of on one or more specially allocated processors. To prevent the creation of duplicate nodes, the blackboard level node was locked by the knowledge source or the rule requesting the creation, until either a new node was created or an existing one was found. The use of local creation (on the same processor as the knowledge source or the rule requesting the node creation) improved throughput to 25ms, or a 37% improvement over Experiment C-5 (see Table 8-8).

8.4.9. Summary Discussion

There are two important measurements that can be considered the cumulative results of the Cage experiments. These are the maximum relative speed-up — comparing uniprocessor runs with multiprocessor runs, and the minimum sampling interval — measuring the total throughput.

8.4.9.1. Speed-up

Speed-up is a relative measure, comparing the uniprocessor speed with multiprocessor speed, using the methodology discussed in Section 8. The maximum speed-up achieved by Cage was 5.9 using a 32 processor grid with knowledge sources and rules running concurrently without synchronization. The factors limiting speed-up to 5.9 include:

- Existence of central scheduler.
- Serial definition section of knowledge sources.
- Inefficient allocation of processes to processors.
- The high overhead of closures within knowledge sources which caused the copying of large amounts of local data combined with slow communication between the processors and memories.

Serial Definitions: In Cage each knowledge source consists of a set of local bindings, which we call *definitions*, and a set of condition-action rules which can reference the local definitions. The definitions include references to blackboard nodes, calculations with values retrieved from those nodes, and the creation of new nodes. The definitions are the only part of the knowledge sources executed serially during the Cage experiments. By executing the definitions for each knowledge source in parallel we could theoretically expect as much as a 40% increase in speed-up because there

are an average of 11.5 definitions per knowledge source and definitions account for about 89% of the total knowledge source cost.

Executing the definitions in parallel is an option in Cage. The speed of the definitions would then be limited by the longest definition. A side-effect of a definition can be the creation of a new node. From Table 8-5 we can see that creation of a new node or matching for an existing node costs 63% of the total definition time. However, these definitions, as specified by the Elint application, are likely to have a number of implicit points of serialization because of data dependencies and do not all make equal computational demands. Thus the achievable improvement, in practice would be much less than the 63% quoted above.

Resource Allocation: The second way to gain speed-up is to improve the resource allocation. However, in most AI programs it is not possible to pre-specify optimal allocations because of the dynamic nature of the programs. Given an application in Cage, a good resource allocation scheme can be evolved through experimentation, as was seen in Experiments C-4 , C-5, and C-6.

For Experiments C-1 to C-3 the identical allocation scheme was used regardless of the number of processors used, statically assigning some processes (input handler, for example) to specific processors but allowing them to be used by other processes. For example, in Experiment C-5 data input time was reduced from 15ms to 6.5ms with hand crafted processor allocations. Likewise, in Experiment C-6, which assigned separate processors to each blackboard level for node creation, throughput improved by 22% over Experiment C-5, which used a single creation process. This general scheme could be used for specific applications and specific numbers of processors.

Qlisp: The final factor limiting speed-up for Cage is the high overhead costs of the use of the Qlisp implementation, particularly Qlisp process closures. A Qlisp process closure is expensive for Cage, because Cage requires the copying of the context (the local definitions of a knowledge source) from the spawning processor to the executing processor. This overhead accounts for approximately 2/3 of the total node creation time.

8.4.9.2. Throughput

Throughput is an absolute measure, measuring the rate at which input data can be processed, or the sampling interval, as discussed in Section 7. When the throughput that Cage can achieve is compared with that of Poligon, it is relatively low. The minimum sampling interval for Cage is about 10 times that of Poligon for the same number of processors. Cage was limited to a best sampling interval of about 25ms.

The general reasons that limit speed-up also apply to the relatively poor throughput. First, a more efficient use of Qlisp, eliminating one unnecessary call, led to a 22% reduction in the sampling interval. Second, the latest

test runs show a reduction in the sampling interval of 43% with the use of the latest CARE simulator. Third, as seen in Experiment C-6, the sampling interval was reduced to 31ms with 32 processors with some simple resource allocation optimization.

To summarize, Cage can execute multiple sets of rules, in the form of knowledge sources, concurrently. If the rule parallelism within each knowledge source could provide a certain speed-up, and if many knowledge sources could be run concurrently without getting in each other's way, it was hoped that we would get a multiplicative speed-up. The extra parallelism coming from working on many parts of the blackboard, in other words, by solving sub-problems in parallel. It was found, however, that the use of a central scheduler to determine which knowledge sources to run in parallel drastically limits speed-up, no matter how many knowledge sources are executed in parallel. This is primarily a function of the granularity of the serial components. Nonetheless, a trade-off must be made between the high cost of process creation and switching, and granularity. We were able to get a speed-up factor of 4 by running knowledge sources in parallel. However, we were not able to get any significant speed-up by running the rules within each knowledge source in parallel. due in part to: (1) the large chunk of serial definitions in each knowledge source; (2) the fact that there is only an average of 4.5 rules in each knowledge source, and (3) the high overhead cost of process creation and switching. With more efficient definitions, additional rules, and faster process switching we may be able to get better relative speed-up and a higher throughput.

8.5. Experiments with the Poligon system

The following sections detail a number of experiments performed on the Poligon system using the Elint application. The purpose of these experiments was as follows:

- To measure the benefits of pipeline and data parallelism in the application.
- To determine the ability of the system to exploit rule parallelism.
- To estimate the costs of running the system without system optimizations, which reduce the ability of the programmer to debug applications.
- To determine whether some changes to the timing of the system were valid.
- To measure the costs of many of the primitive operations in Poligon so as to be able to estimate the granularity of a Poligon program.

8.5.1. Experiment P-1: Thin Data Set

8.5.1.1. Description

This experiment simply used the experimental method elucidated in Section 8.2. Nothing special was done to the system in order to perform it. The data set was run on 1, 2, 4, 8, 16, 32, 64 and 128 processor systems. In order to determine each experimental data point, the input data sampling interval was adjusted until the latencies for the reports generated were non-increasing.

8.5.1.2. Purpose

Processors	Sampling Interval [ms]	Speed-up
1	9	1.0
2	5.5	1.6
4	4.5	2.0
8	4.0	2.3
16	3.1	2.9
32	2.6	3.5
64	2.5	3.6
128	2.5	3.6

Table 8-8. The results of Experiment P-1.

The purpose of this experiment was threefold. First, it was to measure the performance of the system by deriving both speed-up and minimum sampling interval measures. Second, the experiment was intended to provide a base-line for comparison with subsequent experiments. Third it was intended, to evaluate the speed-up provided by the Elint application as a result of pipeline parallelism. This latter can be done using this data set because the data set results in the creation of only one pipe in the solution, unlike subsequent experiments using the *Fut* data set.

8.5.1.3. Results

The results derived from this experiment are shown in Table 8-8 and are also shown in Figure 8.14.

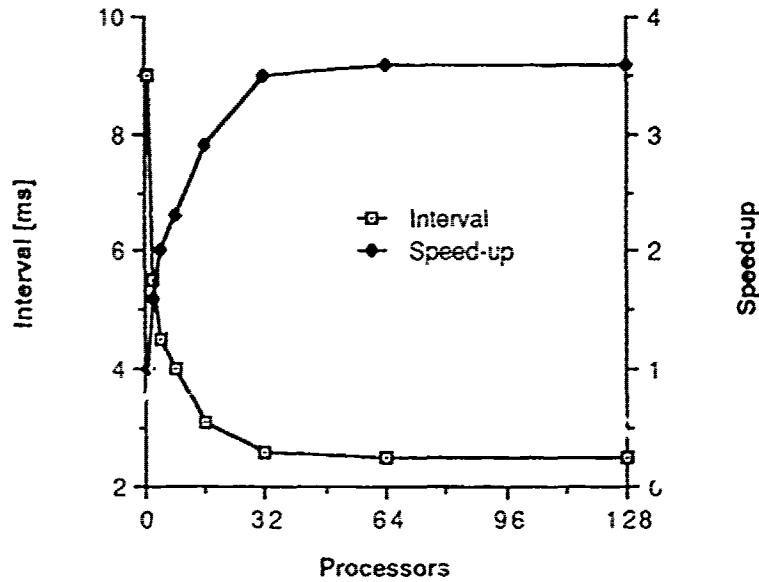


Figure 8.14. The results of Experiment P-1.

Minimum sampling interval: 2.5ms with 64 processors.

8.5.1.4. Interpretation

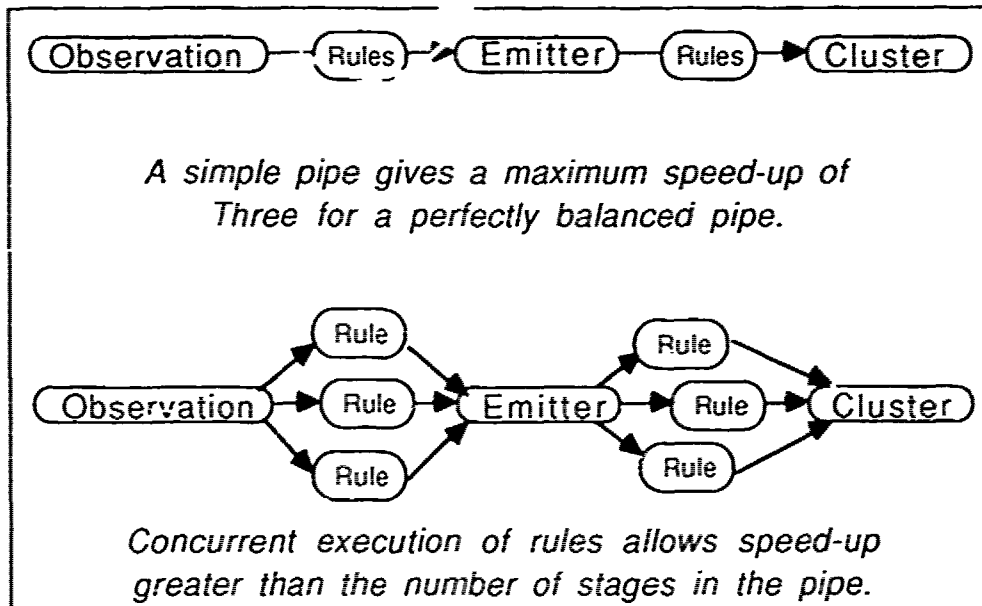


Figure 8.15. Rule Parallelism in Poligon.

From this experiment we can see that the Poligon system has produced a speed-up of 3.6 as a result of pipeline parallelism. This is a fairly encouraging result, since it shows that a certain amount of parallelism is being achieved due to parallel rule execution. We can conclude this because the pipes formed by the Elint application have only three stages resulting in a maximum speed-up of three for a simple pipe (see Figure 8.15).

8.5.2. Experiment P-2: Fat Data Set

8.5.2.1. Description

This experiment was exactly the same as Experiment P-1, except for the data set used, which was the "Fat" data set.

8.5.2.2. Purpose

The purpose of this experiment was twofold. First, it was to provide a baseline by which the Polygon system's performance could be compared with the implementations of the Elint application using the Lamina¹ and Cage systems. Second, it was intended to give a measure of the ability of the Polygon system to exploit parallelism in the data. This could not be determined from the previous experiment because the data set only allowed the creation of a single pipe during the execution of the program. In the *Fat* data set there were multiple emitters and clusters, which caused, quite intentionally, the creation of multiple pipes during the solution of the problem.

In the *Fat* data set one would expect four pipes to be created. However, this does not mean that one would necessarily expect the speed-up to be four times greater than that delivered by a data set only one quarter as "wide" (the *Thin* data set), although one might hope that it would be.

8.5.2.3. Results

Processors	Sampling Interval [ms]	Speed-up
1	31	1.0
2	18	1.7
4	15	2.1
8	16	1.9
16	10	3.1
32	4	7.8
64	2.9	10.7
128	2.7	11.5

Table 8-9. The results of Experiment P-2.

The results derived from this experiment are shown in Table 8-9 and in Figure 8.16.

¹For a more information on these experiments, please see [Delagi 85].

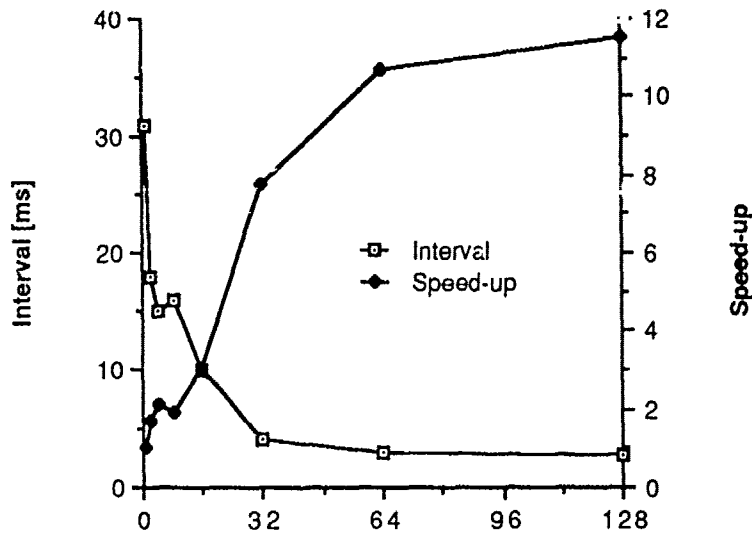


Figure 8.16. The results of Experiment P-2.

Minimum sampling interval: 2.7ms with 128 processors.

8.5.2.4. Interpretation

The minimum sampling interval shown above is a measure of the application's ability to process data. This figure is compared to the best sampling intervals of the implementations of Elint using the Cage and Lamina systems in Table 8-10.

There are a number of reasons why the Poligon implementation of the Elint system was not as fast as that using Lamina. These fall into two main groups, those due to the encoding of the application and those due to the framework itself.

System	Best Sampling Interval [ms]
Lamina	0.5
Poligon	2.7
Cage	25

Table 8-10. A comparison of the peak throughputs of the Lamina, Poligon and Cage implementations of the Elint application.

Application: The Elint application in Poligon was intentionally not tuned. That is to say, the application was an attempt to make an implementation of the original, serial implementation of Elint done using AGE. As a result of this, the application was coarser grained than the Lamina implementation (for example Lamina used seven-stage pipes). It was also not redesigned so as to improve its efficiency, whereas the Lamina implementation went through a number of different designs, so as to improve its efficiency and the balance of its pipes. Similarly, the Lamina implementation of Elint

used a carefully crafted resource allocation strategy, while Poligon used a simple, random allocation strategy. Although effectively the only tuning done to the Poligon implementation of Elint was the addition of type declarations, this should not be taken as an indication that this is the sort of performance that one might expect from an application written by a naïve user. The implementor of the application was, in fact, also the implementor of the Poligon system.

Framework: It is clear that there are significant costs associated with maintaining the abstraction model and mechanisms supported by Poligon. This has a substantial effect upon the minimum grain size that the system is able to achieve. The granularity of the Poligon system will be discussed below in greater detail. In short, the costs incurred by the Poligon framework over those incurred by the Lamina implementation of Elint are: (1) the cost of rule invocation, (2) the cost of reading slots (due to the complex behavior of slots) and the cost of writing slots (due to the *smart-slot* protocol), and (3) the costs of communication.

The second conclusion that can be drawn from this experiment concerns Poligon's ability to exploit data parallelism. The minimum sampling interval for this experiment was not statistically different from that in Experiment P-1 (see Table 8-11).

We can conclude, therefore, that almost linear speed-up results from increasing the width of the input data stream.

Experiment	Data Set	Best Sampling Interval [ms]
P-1	<i>Thin</i>	2.5
P-2	<i>Fat</i>	2.7

Table 8-11. The peak throughput of the Poligon Elint application for different data sets.

8.5.3. Experiment P-3: Multiple Rules

8.5.3.1. Description

This experiment was performed using the *Thin* data set. All data points were measured on a 128 processor network. The Poligon system was modified so that whenever a rule was triggered to fire it would actually fire N rules, where N was a user definable parameter. Of these rules all but one were specialized so that they performed all of their operations except for executing their action parts. This can be done because of the guaranteed side-effect free semantics of rule condition parts.

8.5.3.2. Purpose

The reason for performing this experiment was to try to find some measure of the ability of the Polygon system to exploit "Rule" parallelism, that is, achieve speed-up through the concurrent activation of rules. Unfortunately, the Elint application was very sparse in knowledge and it is very hard to determine whether a system with different amounts of knowledge is solving qualitatively the same problem, so adding more knowledge would not give a good measure. It was decided, therefore, to invoke dummy rules to simulate, as closely as possible, the costs of rule invocation without actually executing the action parts of the rules. This guarantees that the system still performs as it should. If the Polygon system is able to exploit rule parallelism, then one would expect that the minimum input data sampling interval would remain constant, irrespective of the number of dummy rules fired. The slow-down experienced by the application should, therefore, give a rough measure of the usefulness of Polygon's architecture to exploit multiple, simultaneous rule activations.

8.5.3.3. Results

The results of Experiment P-3 are shown in Table 8-12 and Figure 8.17.

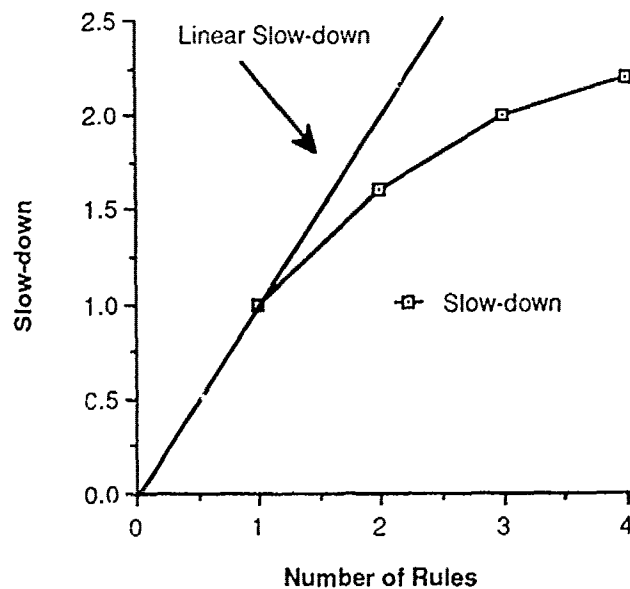


Figure 8.17. The slow-down due to invoking dummy rules in the Elint application. Values below the "Linear Slow-down" line indicate the exploitation of useful parallelism.

Number of rules fired (including non-dummy rule)	Sampling Interval [ms]	Slow-down
1	2.5	1
2	4.0	1.6
3	5.0	2.0
4	5.5	2.2

Table 8-12. The results of Experiment P-3.

8.5.3.4. Interpretation

First, it should be noted that this experiment was performed on a fixed processor network size to eliminate one more variable factor. However, as the number of dummy rules activated increases, one would expect that the resources would become more scarce, reducing the performance of the system. This should be kept in mind when considering these results.

If one assumes that the amount of work performed by a dummy rule activation is approximately equivalent to the amount of work done by a non-dummy rule, then we can conclude that the system slowed down by only a factor of 2.2, while doing four times as much work. This is by no means a perfect result, but it shows that the Polygon system can scale to cope with a knowledge base at least four times as large as that used by the Elint system and not clog up completely. In this case "Large" is taken to mean the average number of applicable rules for any given set of slot updates. Untriggered rules cost nothing in terms of rule invocation overhead.

The slowing down of the system was due to the following:

- The serial execution of the code which invokes rules.
- Resource contention.
- Communication overhead.

8.5.4. Experiment P-4: Make-Instance

8.5.4.1. Description

In this experiment the 64 processor data point of Experiment P-2 was rerun with the Polygon system modified so as to charge for the time taken during the creation of Polygon nodes.

8.5.4.2. Purpose

During all of the other experiments reported here the creation of the actual Lisp machine *Flavors* instance, which represents a Polygon node, is taken

to take zero time. The reason for this is that Polygon nodes are implemented as *Flavors* instances and each of their slots are also *Flavors* instances. In a "real" system one could, with suitable compilation, implement these nodes simply as arrays. Even when the system does not charge for this instantiation the simulation still charges for the initialization of the node, which does not happen at *Flavors* instance creation time, since this occurs after the system's call to *Make-Instance*. In a real system, therefore, the creation of a node could be accomplished simply by allocating the memory and *BLTing*¹ a template into it. This could be done in a time which would be very small relative to the other times for operations in the system. Thus, it was decided not to charge for this instantiation in the simulation, because *Flavors* instance creation is very expensive due to its initialization protocol.

This experiment, therefore, was a reality check to determine how much the experimental results were being affected by not charging for the creation of instances.

8.5.4.3. Results

Minimum sampling interval achieved: 3.4ms.

8.5.4.4. Interpretation

When the Elint system charged for the instantiation of *Flavors* instances the fastest sampling interval that the system could handle slowed down by some seventeen percent. From this one can conclude both that the experiments were not vastly affected by not charging for instance creation and that in a "real" system one would want to design the system so as to avoid this extra cost which, although not vast, is still significant.

8.5.5. Experiment P-5: Optimization

8.5.5.1 Description

This experiment was performed by rerunning the 64 processor data point taken for the experiment on the *Fat* data set. But, before this was done the Polygon system and the application were recompiled so as to be running in Polygon's "development" mode, with all of the Polygon system's optimizations turned off and source code debugging switched on.

8.5.5.2. Purpose

This experiment was designed to show the relative performances of applications running in Polygon's development mode and in Polygon's "production" mode. A data point of this nature would allow one to make an estimate of the best case performance of one's system while still in the development phase.

¹BLT instructions are fast Block Transfer instructions.

8.5.5.3. Results

Minimum sampling interval achieved: 9.0ms.

8.5.5.4. Interpretation

This experiment showed that the benefit in application performance of the optimizations provided by Poligon over the development, unoptimized case is about a factor of three.

8.5.6. Experiment P-6: Granularity

8.5.6.1. Description

This series of experiments was performed on the Poligon system without it running under the CARE simulator. Experiments were performed which timed a large number of the following:

- Slot reads.
- Slot writes.
- Slot writes, including the execution of rule invocation code up to but excluding the actual triggering of the rule.
- Slot writes, including the execution of rule invocation code including the creation of the rule invocation context, but excluding the execution of the *When*¹ part of the rules
- Slot writes, which caused the triggering and evaluation of the *When* parts of rules.

Each component was run by finding a useful set of arguments for the relevant calls. It should be noted that this experiment ignored the cost of communication. This was taken as a fixed characteristic of the system.

8.5.6.2. Purpose

These experiments were designed to measure the cost of the fundamental operations in the Poligon system. This should allow the development of empirically derived formulae, which would allow the estimation of the computational grain size of a Poligon program.

Rule activation in Poligon goes through a number of stages, each of which will have associated costs. First there is the slot update, which causes the rule activation. The cost of slot updates will vary according to whether there were any attached rules or not. Second there is the context creation. This is the point at which the system creates the environment in which the

¹The *When* part of a rule is a sort of locally evaluated pre-condition.

rule will execute. It includes the copying of a template for the state information used during rule activation. Finally there is the evaluation of the *When* part of the rule. This is a piece of user code, which can clearly take an arbitrary amount of time. For this experiment, therefore, a "representative" *When* part was taken from the Elint application.

The minutiae of these experiments will not be given here. Instead, the results will be shown in a manner which allows the reader to see the granularity of the different measured components of the Poligon framework, without having to wade through a full explanation of the experiments themselves. As a result of these experiments, empirically derived formulae will be shown, which denote either the real performance or a "normal case" measure taken from the Elint system.

8.5.6.3. Results and Interpretation

The time taken to read a slot was found to be independent of the length of the slot's value list and linearly related to the number of slots read. Correcting for the processor speed for a CARE machine we have the formula for the calculation of the cost of slot reads in Poligon.

$1.36 + 0.94n$ microseconds, where n = the number of slots being read.

The cost shown for slot reads is very low. Of course, this only gives a measure of the cost of local reads in Poligon and the measurement does not account for the cost of communication at all. However, because of the nature of blackboard systems, where rules tend to deal with data locally available and then pass their conclusions on to other nodes, most read operations that happen are, in fact, local. Making this operation fast is one of the main causes for the difference in performance between Poligon and Cage. In Cage all slot accesses have equal cost, but that cost is much higher than Poligon's local slot accesses because of having to make a read to shared memory.

Like slot read operations, the cost of writing slots is linearly related to the number of slots being written. The formula for the cost of non-rule invoking slot updates is as follows, correcting for the speed of the processor as above.

$18 + 53.7n$ microseconds, where n is the number of slot updates.

Because the user can supply arbitrarily complex code, which is executed at slot update time, it should be noted that this figure only reveals the lower bound for the cost of slot updates.

An expression was derived, by experiment, for the cost of rule invocation in Poligon. Some of the values in the expression are bound to be case depen-

dent, but this expression should, nevertheless, be representative of the normal behavior of the system.

The cost of rule invocation is, therefore, given by the sum of the costs of the following operations:

- Slot writes for a slot with an attached rule.
- Creation of the rule activation context object.
- Copying of the definition template for the context object.
- Execution of a typical *When* part.

All but the first of these are application dependent characteristics. The results actually measured, therefore, are taken from a "representative" part of the Elint application. Substituting in measured values for the categories above, we have the cost of rule invocation being:

$$128 + 160 + 370 + 400 = 1058 \text{ microseconds.}$$

This expression should not be taken as a final figure describing the potential performance of the Poligon architecture, but rather a measure of the performance that could be achieved without a major rewrite of the system and without spending a great deal of effort on the optimization of the system. It should be easy to eliminate most of the time due to instance creation and due to definition template copying in a production quality system. With sundry other optimizations a figure better than half of the one millisecond quoted above should be readily deliverable.

9. Discussion

In this paper we discussed the relationship between the blackboard model, its existing serial implementations, and the degree to which the parallelism intuitively thought to be inherent in the blackboard problem solving model is really present.

Cage and Poligon, two implementations of the blackboard model designed to operate on two different parallel hardware architectures, were described, both in terms of their structure and the motivation behind their design.

Our framework (or shell) development, application implementations on these frameworks, and initial performance experiments to date has taught us that: (1) it is difficult to write "real-time," data interpretation programs in a multiprocessor environment, and (2) performance gains are sensitive to the ways in which applications are formulated and programmed. In this class of application, performance is also sensitive to data characteristics.

The "obvious" sources of parallelism in the blackboard model, such as the concurrent processing of knowledge sources, do not provide much gain in

speed-up if control remains centralized. On the other hand, decentralizing the control, or removing the control entirely, creates a computational environment in which it is very difficult to control the problem-solving behavior and to obtain a reasonable solution to a problem. As granularity is decreased, to obtain more potential parallel components, the interdependence among the computational units tends to increase, making it more difficult to obtain a coherent solution *and* to achieve a performance gain at the same time. We described some of the methods employed to overcome these difficulties.

In the application class under investigation, much of the parallelism came from pipe-lining the blackboard hierarchy and from data parallelism; both from the temporal data sequence and from multiple objects (aircraft, for example). The Elint application was unfortunately knowledge poor, so that we were unable to explore knowledge parallelism extensively, except as a by-product of data and pipeline parallelism and in the somewhat artificial form described in Section 8.5.3. Elint has been implemented in both Cage and Poligon, and a number of experiments have been performed. The experiments were designed to measure and to compare performance by varying different parameters: process granularity, number of processors, sampling interval, data arrival characteristics, and so on.

Cage can execute multiple sets of rules, in the form of knowledge sources, concurrently. If the rule parallelism within each knowledge source can provide a speed-up in the neighborhood cited by Gupta, and if many knowledge sources can run concurrently without getting in each other's way, we can hope to get a speed up in the tens. Extra parallelism comes from working on many parts of the blackboard, in other words, by solving many sub-problems in parallel. Unfortunately, experiments to date have not yet shown this (see Section 8.4).

It was found that the use of a central controller to determine which knowledge sources to run in parallel drastically limits speed-up, no matter how many knowledge sources are executed in parallel. Amdahl's limit and synchronization come strongly into play. The implication for Cage is that knowledge-source invocation should be distributed, without synchronization. This will eliminate two major bottlenecks; a data-hot spot at the event list, and waiting for the slowest process to finish during synchronization. One solution to this is to distribute the blackboard, which is one of the main characteristics of Poligon.

The performance of the Poligon system is limited by a different set of constraints. Although a Poligon programmer can, in principle, pick any desired size for the data grain size of the application's blackboard nodes, there will be some optimal grain size for a given application. If the blackboard nodes are small, then there will be more of them and the rules in the system will be more distributed. This should result in more potential parallelism and more communication.

Poligon tries to shield the user from the system cost associated with larger data grain sizes by allowing the concurrent execution of all the rules that may be interested in a given blackboard change. However, because of the non-trivial cost of rule invocation, and because resources are always scarce in the real world, it may be better to commit to a larger grain size and avoid the extra cost of communication and process management. Finding the optimum grain size for a program is still an unsolved problem. The development of a system, which can take a specification of the user's program requirements and compile it into the best grain size, is an important topic for research as more multiprocessor systems become available and programmers strive for higher productivity.

It is clear that much more research is needed in this area before a combination of a computational and problem-solving model can be developed that is easy to use, that produces valid solutions reliably, and that can consistently increase speed-up by a significant amount without undue programmer effort.

10. Conclusions

We have described the purpose of the Advanced Architectures project at Stanford University. In particular we have discussed the Cage and Poligon sub-projects. Cage and Poligon are two different types of concurrent blackboard system.

The same application, called Elint, has been mounted on each of these frameworks and experiments have been performed. The experiments, the experimental method and the results have been enumerated and discussed. From these experiments it has been shown that:

Cage: A peak speed-up of $5.7\times$ was achieved for 16 processors, and improved to $5.9\times$ for 32 processors. The throughput of the system was limited by four main factors: an inherent serialization due to the centralized scheduler, the unoptimized Cage system, the overheads due to the Qlisp concurrent Lisp language, and the overheads due to using a simulator for a distributed-memory multiprocessor as a simulator for a shared-memory machine. The fastest input sampling interval achieved by Cage with minimal optimization and 32 processors was 25ms.

Poligon: A peak speed-up of $11.5\times$ was achieved with a best input data sampling interval of 2.7ms. Pipeline parallelism contributed about $3\times$ of this. The remaining speed-up was due to parallelism extracted from the data being processed. At least within the bounds of the experiments described, near linear speed-up has been shown for increasing complexity of the input data. It seems likely, therefore, that given more data the system would be able to achieve better results. It has been shown that as the knowledge base size increases, the Poligon system should deliver significantly better than linear slow-down, given sufficient resources.

Comparing the two systems, Poligon out performs Cage by approximately a factor of 10. (Note that the speed-up of 5.9x for Cage and 11.5x for Poligon are not comparable, since the measurements are relative measures within the same framework between uniprocessor and multiprocessors. They indicate different abilities to exploit parallelism.) This is no great surprise, since the Poligon system takes a significantly more aggressive stance with respect to performance.

It is not clear whether the speed-up factors we obtained would apply to other problems. As mentioned, throughout, the possible opportunities for concurrency and granularity are very application dependent, and thus it is very difficult to generalize from the results of one application. Nonetheless, in both Cage and Poligon the speed-up for our application came from: (1) data parallelism present in Elint, (2) pipelining of reasoning steps, (3) parallel matching for relevant rules, and (4) knowledge parallelism where more than one piece of knowledge was applicable for a given state. These sources of parallelism are fairly general and can be exploited by most applications.

Writing a "real-time" applications for Cage and Poligon was by no means simple. Many problems arose regarding timing measures, data consistency and coherence, and test scenarios that would not have arisen in other types of problem. Nonetheless, by attempting to solve a difficult problem, we were able to develop techniques and methodologies that will be useful for other applications in this class as well as in broader classes of problems. Much of what we learned has become a part of the frameworks; others were described in Section 4.4 for Cage and Section 5.2 for Poligon. It is our belief that both architectures represent viable ways of constructing concurrent blackboard systems.

11. References

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism: The Cage System*. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.
- [Amdahl 67] Gene M. Amdahl. *Validity of a Single Processor Approach to Achieving Large Scale Computing Capabilities*. Proceedings of AFIPS Computing Conference 30, 1967.
- [Brown 82] Harold Brown, Jack Buckman, et al. *Final Report on Hannibal*. Technical Report, ESL, Inc., 1982. Internal Document.
- [Brown 86] Harold D. Brown, Eric Schoen and Bruce Delagi. *An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures*. Technical Report KSL-86-69,

Knowledge Systems Laboratory, Computer Science Department, Stanford University, October 1986.

- [Byrd 87] Gregory T. Byrd, Fussel Nakano and Bruce A. Delagi. *A Dynamic, Cut-Through Communications Protocol with Multicast*. Technical Report STAN-CS-87-1178, Computer Science Department, Stanford University, 1987.
- [Corkill 83] Daniel D. Corkill and Victor R. Lesser. *The Use of Meta Level Control for Coordination in Distributed Problem Solving*. Proceedings of the 7th International Conference on Artificial Intelligence: 748-755, 1983.
- [Delagi 86a] Bruce Delagi. *CARE Users Manual*. Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 86b] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-70, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 88] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Durfee 85] Edmund Durfee, Victor Lesser and Daniel Corkill. *Coherent Cooperation Among Communicating Problem Solvers*. Technical Report, Department of Computer and Information Sciences, September, 1985.
- [Ensor 85] J. Robert Ensor and John D. Gabbe. *Transactional Blackboards*. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.
- [Fennell 77] Richard D. Fennell and Victor R. Lesser. *Parallelism in AI Problem Solving: A case Study of Hearsay-II*. IEEE Transactions on Computers: 98-111, February, 1977.
- [Forgy 77] Charles Forgy and John McDermott. *OPS, A Domain-Independent Production System Language*. Proceedings of the 5th International Joint Conference on Artificial Intelligence 1:933-939, 1977.
- [Gabriel 84] Gabriel, Richard P. and McCarthy, John. *Queue-based Multi-processing Lisp*. Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984.

- [Gupta 86] Anoop Gupta. *Parallelism in Productions Systems*. Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1986. Ph. D. dissertation.
- [Hewitt 73] Hewitt, C., P. Bishop and R. Steiger. *A Universal, Modular Actor Formalism for Artificial Intelligence*. Proceedings of the 3rd International Joint Conference on Artificial Intelligence: 235-245, 1973.
- [Lesser 83] Victor R. Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks*. The AI Magazine, Fall:15-33, 1983.
- [McCune 83] Brian P. McCune and Robert J. Drazovich. *Radar with Sight and Knowledge*. Defense Electronics, August 1983.
- [McDermott 83] John McDermott and Allen Newell. *Estimating the Computational Requirements for Future Expert Systems*. Technical Report, Internal Memo, Computer Science Department, Carnegie-Mellon University, 1983.
- [Newell 62] Allen Newell. *Some Problems of Basic Organization in Problem-Solving Programs*. In M. C. Yovits, G. T. Jacobi and G. D. Goldstein (editors), Conference on Self-Organizing Systems, pp 393-423. Spartan Books, Washington, D.C., 1962.
- [Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 82] H. Penny Nii, Edward A. Feigenbaum, John J. Anton and A. Joseph Rockmore. *Signal-to-Symbol Transformation: HASP/SIAP Case Study*. AI Magazine vol 3-2, 23-35, 1982.
- [Nii 86] H. Penny Nii. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, vol. 7-2 and vol. 7-3, 1986.
- [Rice 86] James Rice. *Poligon: A System for Parallel Problem Solving*. Technical Report KSL-86-19, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
- [Rice 88] James Rice. *The Advanced Architectures Project*. Technical Report KSL-88-71, Knowledge Systems Laboratory,

Computer Science Department, Stanford University, January 1989.

- [Schoen 86] Eric Schoen. *The CAOS System*. Technical Report KSL-86-22, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
- [Selfridge 59] Oliver G. Selfridge. *Pandemonium: A Paradigm for Learning*. Proceedings of the Symposium on the Mechanization of Thought Processes: 511-529, 1959.
- [Shafer 86] Steven A. Shafer, Anthony Stentz and Charles Thorpe. *An Architecture for Sensor Fusion in a Mobile Robot*. Proceedings of the 1986 IEEE International Conference on Robotics and Automation, 1986.
- [Smith 81] Smith, B.J. *Architecture and Applications of the HEP Multiprocessor Computer System*. Proceedings of the International Society for Optical Engineering. San Diego, California, August 25-28, 1981.
- [Spain 83] David S. Spain. *Application of Artificial Intelligence to Tactical Situation Assessment*. Proceedings of the 16th EASCON83: 457-464, September, 1983.
- [Williams 84] Mark Williams, Harold Brown and Terry Barnes. *TRICERO Design Description*. Technical Report ESL-NS539, ESL, Inc., May, 1984.

Signal Understanding and Problem Solving: A Concurrent Approach to Soft Real-Time Systems

by

H. Penny Nii and James Rice

(Nii@Sumex-Aim.Stanford.Edu & Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract¹

In this paper we present the results of building and running a knowledge-based signal interpretation program on multi-processors. A serial, knowledge-based program to interpret passive radar signals from multiple sites was re-designed and re-implemented on a framework for concurrent problem solving. The primary objective for the use of multi-processors was to gain performance improvement. A number of experiments were performed to evaluate the performance gains under varying circumstances. Factors that motivate and constrain problem solving in parallel in general, and real-time signal interpretation in particular, are discussed.

1. Introduction

A few of the results from an experiment in speeding up a signal understanding program using a concurrent approach to problem solving are described in this paper. This experiment is one of several experiments conducted within the Expert Systems on Multi-processor Architectures Project of the Knowledge Systems Laboratory at Stanford University [6].

The Architectures Project is conducting several experiments to understand the computational characteristics of complex knowledge-based, or expert, systems in a parallel computational environment. The primary goal of this experiment is to exploit what appears, at first glance, to be features inherently parallel in blackboard systems [3]. A simple blackboard architecture AGE [5] was modified into a system called Poligon [7], a concurrent problem solving and programming environment. A signal understanding and information fusion systems was built using Poligon.

ELINT (Electronic INTelligence) is an application for interpreting pre-processed, passively acquired radar emissions from aircraft was chosen. The ELINT application was a part of a larger blackboard system called TRICERO, which interpreted and fused radar emissions and voice data (COMINT) for the purpose of situation assessment [8].

The ELINT application was rewritten for Poligon. Poligon runs on a hardware system simulation environment called CARE [2]. CARE is written in Common Lisp and runs on Texas Instrument ExplorerTM machines² and the Common Lisp platforms. The CARE simulated machine uses dynamic cut-through routing through the communication grid for inter-processor communication. Message transit time is not predictable. As a consequence, without the imposition of expensive network protocols (with the corresponding serialization of execution), communication and the processing that is triggered by it is intrinsically non-deterministic in the sense that two executions of the same program on the same input data can result in different problem solutions depending on different message arrival orders [1]. The challenge for ELINT on Poligon was to produce consistent solutions with minimum control (serialization), which reduces parallelism and performance. One of the objectives of the experiment was to determine if there is a trade-off between knowledge and control in blackboard systems. The second objective was to exploit different types of parallelism in expert systems, namely data parallelism, knowledge parallelism, and inference parallelism. The third objective was to determine whether linear speedup, with respect to number of processors, could be achieved.

¹ This paper was presented as an invited talk at the Twenty Third Conference on Signals, Systems and Computers, Asilomar CA October 1989.

² Explorer is a trade mark of Texas Instruments Corporation.

In the following sections we describe the serial ELINT system, followed by the concurrent ELINT system written in Poligon. We then describe our speed-up measurement method and some experimental results.

2. The Serial ELINT Application

ELINT was originally implemented in AGE [5], a software tool for developing blackboard expert systems. Blackboard systems consist of three major components: (1) A global data store, called the blackboard, which holds input data and intermediate results. The data on the blackboard are hierarchically organized. (2) A collection of rules and procedures, called knowledge sources, which derive the intermediate results. A knowledge source operates on a particular portion of the blackboard, usually using data on one level of the hierarchy as input, and producing results on another level. (3) The knowledge sources are activated by a control module that dynamically determines the situation under which the knowledge sources are to be applied. That is, the controller determines the line of reasoning, and the basic reasoning is opportunistic, that is, the most productive solution path is chosen depending on the solution state at any given point in time. (See Figure 1 for an example system.)

ELINT is a relatively simple, but non-trivial, application. The inputs to the ELINT system are multiple, time-ordered streams of processed observations from multiple collection sites, some of which may be mobile. ELINT's objective is to correlate, on the blackboard (which emulates the situation board), a large number of input data into individual radar emitters producing those emissions (Figure 1). The emitters are then aggregated into a smaller number of clusters. A cluster is defined as a collection of emitters which are co-located over time. That is, if emitters have the same location fixes, with some resolution, for some period of time, then they are considered to form a cluster. Conceptually, a cluster is a single platform, or two or more platforms that are co-located over time (for example, an aircraft and its wingman). The system must split a cluster when the fixes of co-located emitters diverge.

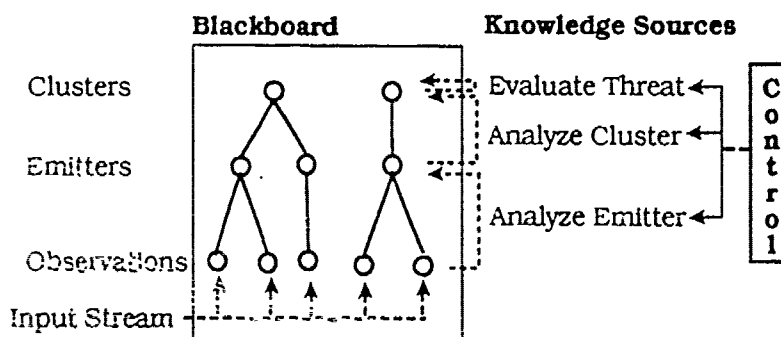


Figure 1. Serial ELINT in AGE

The determination of whether two observed emissions are from the same emitter, both for intra- and inter-sites, is based on the electronic characteristics of the emissions and on signature analysis. This determination may be in error, and the ELINT system must cope with such an error.

The primary output of the ELINT system is periodic status reports about the tracks and activities of the clusters of emitters.

The different analyses, inference, and reporting activities are performed by the knowledge sources. The basic reasoning strategy in ELINT is a data-driven accumulation of evidence in the input data stream to support the existence of emitters and their tracks. The existence of a cluster is inferred from the behavior of the emitters over time. Since data for an emitter

can be collected at different sites, the system must also determine whether input data from multiple sites belong to the same emitter, that is, data fusions must be performed.

3. The Concurrent ELINT Application

As in the serial implementation, a concurrent version of ELINT must be able to deal with continuous input data streams, and there is a need for real-time processing. However, it is a *soft* real-time application, processing continuous input data as fast as possible. It is not a *hard* real-time application and does not guarantee any specific response time.

Some basic differences between the concurrent ELINT system mounted on Poligon and the serial system described above is summarized below. The concurrent formulation of ELINT is shown in Figure 2.

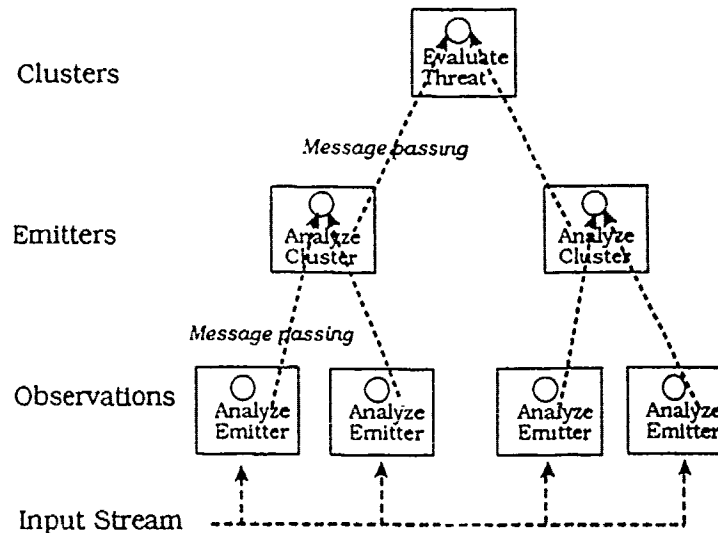


Figure 2. Concurrent ELINT in Poligon

- There is no centralized control. This was motivated by a desire to remove any obvious serial processing. The elimination of a central control requires a new mechanism for activating the knowledge source modules. A knowledge source is activated as a daemon when data associated with the knowledge source is changed on the blackboard. The association between the knowledge sources (actually, rules in the knowledge source) and the data (actually, properties of the objects on the blackboard) that trigger their invocation is made at compile-time.
- When centralized control is eliminated, it also eliminates all global synchronization and any mechanism for the focus of attention. This means that different parts of the program will run at different speeds, and each part will have a different idea of how the solution is progressing. That is, no assumption about the global coherence of the situation board can be made.
- Having eliminated the centralized controller, there is no need for the separation of the knowledge sources from the blackboard. Thus, the knowledge sources are associated directly with the objects on the blackboard that might be the source of activation of the knowledge sources at compile-time. In this way, when a blackboard object is changed, a relevant knowledge source can be activated without the intervention of a controller.
- In conventional, serial blackboard systems, knowledge sources are units of scheduling. If only the knowledge sources are executed in parallel, a great deal of potential parallelism will be lost by the failure to exploit parallelism at a finer grain. Therefore, rules in

the knowledge sources are executed in parallel, and knowledge sources take no part in the sequencing of rule invocations.

- A knowledge source attached to an object communicates changes to be made on other objects by sending messages.
- Poligon runs on a distributed-memory multi-processor. Shared-memory concepts such as global variables are not supported. Because no global variables are allowed, a way is needed to define sharable, mutable data, while still trying to reduce the bottlenecks that can be caused by shared data structures. Poligon uses a generalized class hierarchy to structure the blackboard. The objects belonging to a hierarchy level are instances of a class. The classes are represented on the blackboard and are active, serving as managers that create instance objects. Level managers also store data shared between all of their instances to support operations which apply to all members of a class. Shared data can therefore be implemented in a distributed manner.
- Most blackboard systems represent the properties of an object simply as lists of values associated with the property name. Because knowledge source executions are atomic in serial systems, programs can assume that no external modification will have happened to a value between the time it is read and written by a knowledge source. In asynchronous, parallel systems, because a large number of rules can be attempting to perform operations on the same property simultaneously, a mechanism is needed to assure data consistency without slowing down the access to object properties (a large critical section would reduce parallelism). As an aid in maintaining consistency, Poligon provides *smart properties*. They are smart in the sense that they can have associated with them user defined behavior which can make sure that operations performed on the data leave that data consistent.
- The problem of data consistency within any given object property is reduced by the property being able to determine cheaply and locally whether a modification is reasonable. Global solution coherency can be enhanced by the same process — objects can evaluate whether a modification will lead to a more precise solution. That is, knowledge can replace serial control used to maintain consistency. This causes a sort of distributed hill-climbing which helps the system evolve towards a coherent solution.

3.1. CARE Simulation Machine

The multi-processor ELINT application written in Poligon runs on the CARE simulator. CARE [2] is the name given both to the simulator used on the Advanced Architectures Project and to the hardware designs being developed on that simulator. CARE consists of a kit of components with which to construct simulated multi-processor configurations.

The instrumentation toolkit in CARE allows the user to watch the behavior of the system both from the point of view of hardware performance and the application program. This allows the identification of bottlenecks and hot-spots during system execution.

Each processing element in the CARE machine is made up of two processors, the *Operator*, whose purpose is to execute operating system functions and to perform the task of inter-processor communication; and the *Evaluator*, whose task is the execution of user code (Figure 3). This design allows the application work and communication to go on simultaneously.

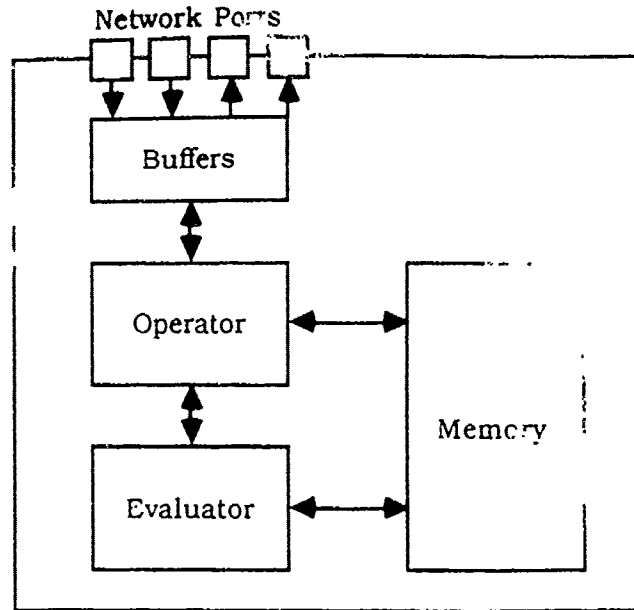


Figure 3. The CARE machine processing element

In the Polygon system the CARE simulator is used to simulate a network of processing elements, connected in a toroidal manner, such that each processing element can talk to its eight neighbors (up/down, left/right and diagonal). The basic idea is to assign blackboard objects to the processors, create processor closures for activated rules within the knowledge sources, and distribute these closures to other processors for execution.

4. Measuring Real-Time Systems

Since the ELINT application is, in some sense, a simulation of the real world, it has a clock of its own which ticks at a constant rate with respect to the time in the real world. The data that comes into the system is time-stamped. When the application's clock has reached a time which is the same as the time stamp on the input data record, the data is introduced into the system. The simulated time between two of these ticks can, in certain circumstances, be used to provide a measure of the throughput of the system. Thus, the tick interval is a parameter that can be varied to measure the system's potential throughput.

A simplistic method for measuring the speed-up of a parallel system would be to take the run-time for the application on a uniprocessor and then divide it by the run-time measured for different numbers of processors. This approach works well for non-real-time systems in which the behavior of the system is not affected by the speed of the computation. In a real-time system with a continuous stream of input data, however, the behavior of the system changes according to the degree to which the system is loaded. For example if more processors are added to a system it can become data starved, failing to deliver the speed-up of which it is capable.

To counter this phenomenon a different methodology was devised. A series of experiments is performed, during which the input data sampling interval is established such that on the largest processor network size the system is never data starved. The speed-up is measured using this sampling interval for other processor configurations, knowing that the delivered speed-up for the large multi-processor configuration would not be data starved. It was found, however, that with all system parameters held constant (except for the number of processors) the application program was still behaving differently for the different experiments. This was because for small numbers of processors the system was getting

backed up, and it was spending a significant amount of time queue-thrashing. That is, it was trying to keep data in order which, if the system had not been so overloaded, would not have got out of order in the first place. This had the effect of making the application seem to run slower on smaller numbers of processors, thus giving an artificially high apparent speed-up.

What was needed, therefore, was a method for measuring the system's speed-up, while making sure that the system was always operating under the same load conditions. To accomplish this, the speed of the application on any particular processor configuration was defined as the lowest sampling interval (that is, highest throughput) that still gives non-increasing latencies in the results. The latency measure is defined to be the time between the data coming into the system and the system emitting any reports concluded from that data.

If the system can keep up with the sampling interval specified, the latency value should be largely constant, otherwise latencies increase over time as the system backs up.

In summary, the following method is used to measure the system's speed: For any given number of processors, the application is run with different sampling intervals until one is found that produces non-increasing latencies. This sampling interval defines the processing speed for a given processor configuration. For a speed-up experiment, the above process is repeated for different processor configurations until the speed-up curve levels off.

4.1. Data Sets

An important aspect of the experiments on the ELINT application is the scenario used to drive the experiment. A scenario represents the simulated radar information that a "real" system would have received. In a real system, one would expect that the number of received radar emissions would vary over time. Although realistic, this sort of scenario is very hard to perform experiments on, since there are bound to be times when the system is either data starved or overloaded. Because of this, two of the data sets used for the experiments have the particular property that they have a constant density of input data over time.

The important characteristics of these data sets, therefore, are the number of radar emissions detected in each time unit, the number of radar emitters, and the number of clusters.

It should be noted that these data sets are used to measure the overall peak system performance for a given data set having the characteristics mentioned below. The system's response to transients in the amount of input data in a timeslice was not measured, nor was its performance for input data with less "typical" characteristics; for instance, a small number of aircraft, each using a large number of radar systems, or a large number of aircraft, each using very few radar systems.

The characteristics of the data sets used for experiments reported below are described below.

Fat 240 Observations, 4 Emitters, 2 Clusters, 8 Observations per time-slice, 30 time-slices, 2 Observations per Emitter per time slice.

Thin 60 Observations, 1 Emitters, 1 Clusters, 2 Observations per time-slice, 30 time-slices, 2 Observations per Emitter per time slice.

5. Experimental Results

A few of the experiments relevant to real-time signal interpretation and data fusion are described below. Wherever reference is made to absolute time, the measurement is in

terms of CARE simulated hardware. Each processing element of this machine has about the performance of a TI Explorer II+ Lisp Machine.

5.1. Speed-Up and Throughput

The Thin data set creates one pipe on the blackboard, and the Fat set creates four pipes; that is, its data is four times as dense. The small data set can be thought of as representing one platform; the second, four platforms. The results from the two data sets allow us to: (1) measure the peak throughput for the larger data set; (2) determine the contribution to speed-up due simply to pipe-line parallelism; and (3) measure the system's ability to exploit data parallelism.

The results of the two data sets are shown in Figure 4.

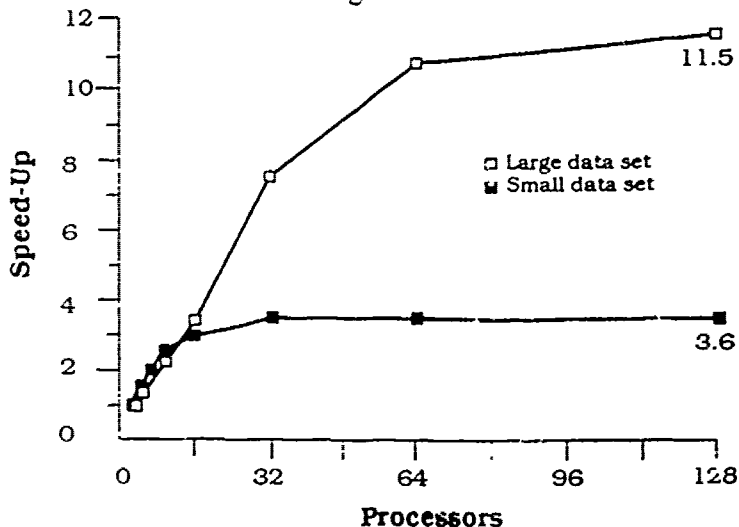


Figure 4. Speed up for Fat and Thin data sets for different number of processors.

In this experiment, following was learned:

1. The peak speed-up shown in this application due to pipe-line parallelism was 3.6. This showed that although the length of the pipe was three, speed-up was greater than three due to the concurrent execution of rules by the different stages (blackboard objects) in the pipe.
2. Almost linear speed-up was achieved with respect to the size of data set.
3. The peak throughput for the system measured in the Fat data set was about 340 μ s per signal data record. Because of the linear increase in performance with respect to data density, it is believed that higher performance can be achieved with more data. By comparison, the serial ELINT on AGE took about 3.7 second to process each data record.

5.2. Exploiting Large Knowledge Bases

In this experiment, only the Thin data set was used. The system was modified so that, whenever an ELINT rule was invoked, N rules would be invoked, rather just one. N-1 of these rules had the special characteristic that they performed almost all of the processing required except for any blackboard modifications - that is, the side-effects in the action parts of the rules were not executed. This gave a measure of the system's performance if the knowledge base was N times larger, while still giving the correct problem-solving behavior.

The results from this experiment are shown in Figure 5.

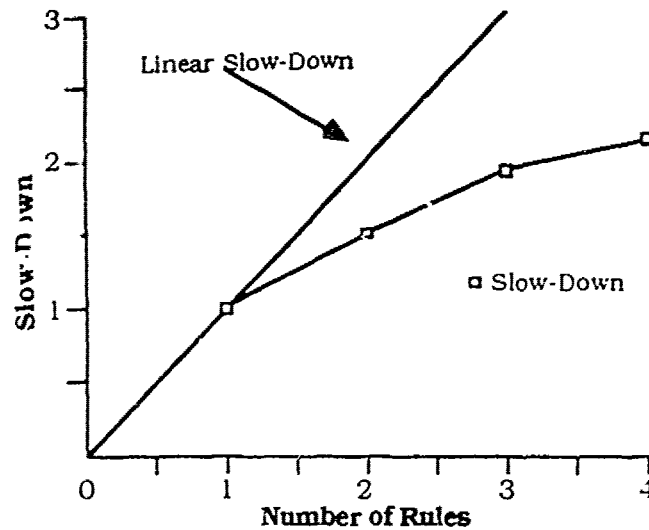


Figure 5. Slow-down plotted against the number of rules being fired for each rule invocation

If the system is able to exploit parallelism in the knowledge base to the full, one would expect that the system would not slow down at all as new rules were added, that is, the line shown in Figure 5 would be horizontal. If, on the other hand, the system bogged down completely as more rules were added, one would expect that the result would be worse than linear slow-down, that is, the plot would appear above the "linear slow-down" line. As can be seen, the performance was better than linear. In order to perform four times as much work, it took only 2.2 times as long. The implication is that, as long as there are sufficient computation resources, the system would deliver good performance for a knowledge base whose size is at least up to four times that of the current application.

5.3. Granularity of Rules

In this experiment some of the internal mechanisms in Poligon were timed to get some empirical measure of the granularity of the system.

A number of mechanisms are of crucial importance to the performance of the system. Among them are reading and writing of property values, and invoking rules. In order to determine the costs of these operations, they were performed repeatedly in a manner which allowed the individual costs to be measured with some precision.

The results are described below. It should be noted that the results neglect any communication overhead, so they are only representative of local operations.

1. Property value reads take $1.36 + .94n$ μ s, where n is the number of properties being read at once.
2. Property updates take $18 + 53.7n$ μ s, where n is the number of properties being written. Since arbitrary user code can be executed during the update operation, this is a representative figure from the ELNET application.
3. The overhead cost of starting up a rule's execution is about 1 ms per invocation.

A substantial part of the time taken performing these operations could be optimized considerably in a production quality system. This experiment shows, however, that there is a lower bound to the granularity that the user can expect to achieve. For computations taking less than a few milliseconds it may not be worth starting up a rule to perform the computation, the cost of parallel execution would exceed the serial execution time.

6. Conclusions

The serial and concurrent implementations of a real-time signal understanding programs were described. The results of experiments performed on the concurrent ELINT application were discussed. The following improvement in performance was observed: A peak speed-up of 11.5 \times was achieved with a best input data sampling interval of 2.7ms. Pipeline parallelism contributed about 3 \times of this. The remaining speed-up was due to parallelism extracted from the data being processed. At least within the bounds of the experiments described, near linear speed-up is possible for increasing complexity of the input data.

The following general observations can be made about concurrent problem solving, at least, on the Poligon concurrent blackboard framework:

- Knowledge, or reasoning, concurrency can be achieved through pipe-lining. However, reasoning, which consists of chain of inference, is inherently a sequential process. Thus, reasoning speedup is limited to the length of the inference process; that is, the number of inference steps determines the length of the available pipe.
- Data concurrency is application and data dependent, but, it is easiest to exploit. Data parallelism manifests itself as multiple pipes on the blackboard.
- To exploit pipeline parallelism, the pipes must be balanced; that is, the knowledge sources must be of uniform granularity and have the same data density. Additionally, when data flows up a hierarchical pipe, the communication up the hierarchy must decrease in proportion to the amount of branchiness.
- Problems can be solved without global control. This, however, depends on the problems being decomposable into loosely-coupled or independent subproblems. Furthermore, we found it necessary for each subproblem to have its own local goals and evaluation function in order to do local hill-climbing to maintain local data consistency.
- Rules can run in parallel, and this parallelism contributes to speedup. In order to run rules in parallel, data needed by the rules needs to be copied and encapsulated to prevent contention on the blackboard objects.
- Writing a real-time application was by no means simple. Many problems arose regarding timing measures, data consistency and coherence, and test scenarios that would not have arisen in other types of problem. Nonetheless, by attempting to solve a difficult problem, we were able to develop techniques and methodologies that will be useful for similar applications. A more detailed description of the concurrent problem-solving architecture and experiments can be found in [4].

7. References

- [1] Harold D. Brown, Eric Schoen, and Bruce Delagi, "An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures," Tech. Report KSL-86-69, Knowledge Systems Laboratory, Computer Science Department, Stanford University, October, 1986.
- [2] Bruce Delagi, "CARE Users Manual," Tech. Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

- [3] Englemore and Tony Morgan (eds.), "Blackboard Systems," Menlo Park, CA: Addison-Wesley, 1988.
- [4] H. Penny Nii, Nelleke Aiello, and James Rice, "Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems," in Les Gasser and Michael N. Huhns (eds.), *Distributed Artificial Intelligence, Volume II*, Pitman Publishing Limited and Morgan Kaufmann, 1989.
- [5] H. Penny Nii and Nelleke Aiello, "AGE: A Knowledge-based Program for Building Knowledge-based Programs." *Proc. of International Joint Conference on Artificial Intelligence 6*, pp. 645 - 655, 1979.
- [6] James Rice. "The Advanced Architectures Project," Technical Report KSL-88-71, Heuristic Programming Project, Computer Science Department, Stanford University, 1988, also in *AI Magazine*, Winter 1989.
- [7] James Rice. The Design and Implementation of a High-Performance, Concurrent Blackboard System Shell - How We Did It With Poligon. Technical Report KSL-89-37, Heuristic Programming Project, Computer Science Department, Stanford University, 1989.
- [8] Mark Williams, Harold Brown, and Terry Barnes, "TRICERO Design Description," Technical Report ESL-NS539, May 1984, ESL, Inc., Sunnyvale, CA.

**AIRTRAC Path Association:
Development of a Knowledge-Based System
for a Multiprocessor**

by

Alan C. Noble and Everett C. Rogers

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, California 94305

This Working Paper is intended for use only within the KSL, and has not been approved for general distribution. It contains information that may be incomplete, preliminary, or not reviewed according to standard procedures. External distribution can only be arranged by contacting the author. This research was supported by DARPA Contract F30602-85-C-0012 and NASA Ames Contract NCC 2-220-S1.

Abstract

Very little is known about programming knowledge-based systems on multiprocessors. To understand the effectiveness of parallel implementations of such systems, programming problems and performance issues need to be studied at all levels of the computational hierarchy, from hardware to application.

AIRTRAC is a simulated radar signal interpretation system for tracking and classifying aircraft, and runs on CARE, a simulated distributed-memory multiprocessor architecture. CARE consists of 1 to 1000 processor-memory pairs communicating via a network that provides reliable message delivery but without message ordering. AIRTRAC is implemented in LAMINA, an object-oriented applications programming interface to CARE, and ELMA, a high-level interface built on top of LAMINA. These are Zetalisp extensions which provide mechanisms and language interface syntax for expressing and managing concurrency.

This report documents the development of the Path Association module of AIRTRAC, from design through implementation and testing, and describes the features of the supporting ELMA interface, which was developed in parallel with the application. We define the criterion of *sustainable data rate* for quantitative performance evaluation and describe experiments to determine performance under different conditions and their results. We discuss the techniques and constructs we used and the lessons we learned in the course of developing Path Association. We believe these lessons and results will be useful to others working in the field of parallel symbolic computation.

1. Introduction

This paper describes the development and testing of a multiprocessor-based continuous signal data interpretation system for tracking and classifying¹ aircraft, called AIRTRAC. This work is motivated by the two following issues:

- Can we achieve significant speedup of large symbolic (knowledge-based) applications through concurrency?
- What techniques and constructs are useful for application software development for multiprocessors?

1.1. High-level Project Goals

AIRTRAC is one project within the Advanced Architectures Project (AAP) of Stanford University's Knowledge Systems Laboratory. The high-level goals of the AAP are to realize:

- software architectures for symbolic applications using parallelism to achieve high-speed computation
- hardware architectures to support those parallel computations

1.2. Research Methodology

Very little is known about programming knowledge-based systems on multiprocessors. To best understand the effectiveness of parallel implementations of such systems, programming problems and performance issues need to be studied at all levels of the computational hierarchy, from hardware to application. The approach taken by the Advanced Architectures Project has been to take a vertical slice through the space of design alternatives and perform experiments with the resulting systems.

¹ The portion of AIRTRAC implemented to date does not classify observed aircraft.

Table 1.1 Our Vertical Design Slice

The following are the design choices we are exploring in the AIRTRAC project, from application down through hardware:

AIRTRAC	continuous signal interpretation application
ELMA	applications programming interface
LAMINA	object-oriented parallel programming language
CARE	distributed-memory, message-passing multiprocessor hardware architecture

Table 1.1 outlines the slice through the space of design choices that we have made. Each of the following four sections describes one level in this vertical slice, with the application level being represented by AIRTRAC.

1.3. CARE--Hardware level

CARE (Concurrent ARchitecture Emulator) is a distributed-memory, asynchronous message-passing architecture, simulated by a highly-instrumented system called SIMPLE. CARE models 1 to 1000 processor-memory pairs, or *sites*¹, communicating via a packet-switched network. Message delivery between sites is reliable, but messages are not guaranteed to arrive in the order of origination.

CARE and SIMPLE have been described at length elsewhere [Delagi87a, Delagi87b].

1.4. LAMINA--Language level

LAMINA is the basic language interface to CARE and consists of Zetalisp with extensions. The extensions provide primitive mechanisms and language syntax for expressing and managing concurrency and locality. Three styles of programming are supported: functional, shared-variable, and object-oriented. All three are based on the notion of a stream, a data type which represents the promise of a potentially infinite sequence of values.²

¹ Throughout this paper the term *site* is used to describe a CARE processing element.

² It is relevant to note that LAMINA's predecessor, CAOS, [Brown86] was based on the notion of a future, or the promise of a single value (resulting from a computation). It was observed, however, that communication between objects was fairly regular; a given object, having communicated with another, invariably communicated with that same object again. The stream notion captures this behavior much more naturally, and was thus chosen as the basic datatype for LAMINA. In LAMINA, a future is the special case of a stream with only one value.

As in other object systems, objects in object-oriented LAMINA (hereafter referred to simply as LAMINA) encapsulate state (instance variables) and behavior (methods). Methods are invoked by message sending but unlike the case of sequential systems this involves sending a packet containing the message from one LAMINA object to another, typically on different sites. Message sending is non-blocking and the time required for communication is thus visible to the LAMINA programmer. Methods run atomically within processes which are restartable but not resumable.¹ An object and its methods can be considered a non-nested monitor; exclusion is guaranteed by the fact that only one method is ever scheduled to run at a time, and then runs to completion. The time required to create a LAMINA object is also visible to the programmer.

The reader is referred to [Delagi 87b] for more details about LAMINA.

1.5. ELMA--High-level language level

ELMA (Extended LaminA for Memory-management Applications) is a high-level parallel programming interface to CARE based on object-oriented LAMINA. ELMA is a specialized interface for applications which involve extensive dynamic object creation and deallocation and require some form of memory management. This interface was developed in parallel with the application.

1.6. AIRTRAC--Application level

AIRTRAC, our application, is typical of continuous signal data interpretation problems, for which projected performance limits of uniprocessors fall short of the speed required by orders of magnitude. Multiprocessor parallel computing must be used to attain the necessary levels of performance. This motivates us to explore mounting applications such as AIRTRAC on multiprocessors.

1.7. Overview of This Paper

Chapter 2 provides an overview of the AIRTRAC application as a whole. Chapter 3 describes design techniques for concurrent programs. Chapter 4 describes the design, implementation, and features of the AIRTRAC Path Association module, and discusses the underlying approach and issues motivating this effort. Chapter 5 describes the ELMA programming interface which embodies the techniques and constructs we found useful in the course of developing Path Association. Chapter 6 describes performance issues for concurrent programs and measured performance results for Path Association. In Chapter 7 we discuss the lessons we learned, and in Chapter 8, future work we propose. Finally, Chapter 9 summarizes this work.

¹ There is also a more expensive resumable cousin.

2. AIRTRAC Application

In this section we describe in more detail the application under development—the AIRTRAC system. We discuss the purpose of the system, its inputs and outputs, and the component levels that make up AIRTRAC.

2.1 Purpose of AIRTRAC

The high-level goal of AIRTRAC is to monitor the flight of aircraft in a particular region of airspace and to interpret and predict their behavior, given tracker data from one or more radar sites within the region.

2.2 AIRTRAC Input

The inputs to the AIRTRAC system are simulated output data from one or more active radar and signal processing systems tracking aircraft in a given region of airspace.¹ Each piece of input data, called a *Radar Track Report (RTR)*, represents the observation of an aircraft from one radar site during a periodic time interval (a *scantime*). Each radar observation is assumed to have been processed by the radar tracking system so that an RTR provides the information listed in Table 2.1.

¹In our experiments we assume that the region is viewed from a small but reasonable number of radars, on the order of two to four.

Table 2.1 AIRTRAC Input

Each Radar Track Report contains the following:

<i>observation scantime</i>	the time of the observation
<i>radar ID</i>	the identifier of the radar site observing the track
<i>track ID</i>	an integer (unique to the radar site) assigned by the radar to the track to which the observation belongs
<i>aircraft type</i>	the type of the aircraft under observation, indicated by signal characteristics
<i>position</i>	the location (x,y) of the aircraft at the time of the observation
<i>position covariance</i>	the (Ex Ey) estimate of the error associated with the reported position
<i>velocity</i>	the velocity (Vx Vy) of the aircraft calculated at the time of the observation
<i>velocity covariance</i>	the (Ex Ey) estimate of the error associated with the reported velocity

Several important characteristics of the data that a radar tracker produces are reflected in the list in Table 2.1. First, a tracker initially assigns a unique track identifier to an aircraft track when the radar system observes that track for the first time. It continues to assign the same track ID to any subsequent observations if it determines, usually by a simple track extension algorithm, that those observations correspond to the previously-detected track. As soon as no such observation succeeds by this algorithm for a track during even a single scantime, that track is considered "lost" and its track ID dropped.¹

Second, a radar tracker is assumed to be capable of determining the type of each aircraft under observation from the particular characteristics of the signal it receives. Finally, the algorithm employed by the tracker calculates covariance figures for the position coordinates and velocity vectors that it reports, providing a measure of the probability of error associated with each of these values. This error information is based on factors such as the strength of the signal and the distance from the aircraft to the radar site. All of this information is passed along as input to AIRTRAC in the form of an RTR.

¹ Basically, this algorithm predicts an area where the next observation for a track should appear, based on a simple linear projection of points already received for that track. If no such observation is forthcoming in the predicted area, the track is lost.

2.3. AIRTRAC Output

The ultimate goal of the AIRTRAC system is to provide continuous information about all aircraft in a monitored region--a kind of "situation assessment." Say, for example, authorities wished to keep a watch on potential drug smuggling activity near a border region. They might use a system like AIRTRAC to manage the acquisition and interpretation of radar data about aircraft traveling across the border.

The types of output that we might expect from such a system could include, among other things:

- *track histories*--a complete history of all aircraft platforms in the region based on fused track data from different radar tracking sources,¹
- *path prediction*--a determination of the future flight paths of observed platforms
- *event prediction*--estimated times and/or locations of "notable" events, such as airport landings or border crossings
- *platform classification*--a categorization of all hypothesized aircraft, based on their histories and predicted future flight paths (e.g., "smugglers" and "not smugglers"),
- *strategy assessment*--an interpretation of the motives of platforms to support their classifications and predicted flight paths,²
- *collision avoidance*--warnings of potential danger for one or more platforms in the system.

The work of specifying the exact content of the output of the entire AIRTRAC system remains to be completed in future stages of the project, but these are the current goals.

2.4. AIRTRAC Modules

The AIRTRAC system is decomposed into three major modules. The modules are Data Association (already completed), Path Association (the main focus of our research and the motivation for this paper), and Path Interpretation (yet to be completed). As shown in Figure 2.1, each module takes as input the output from the previous module. AIRTRAC, then, can be viewed as an

¹ The word "platform" is the term used to denote a hypothesized aircraft.

² For instance, AIRTRAC might determine that an aircraft is attempting to avoid radar surveillance because it has an erratic flight path, thereby supporting a classification of "smuggler."

application that employs several distinct levels of abstraction and reasoning leading to its final output.

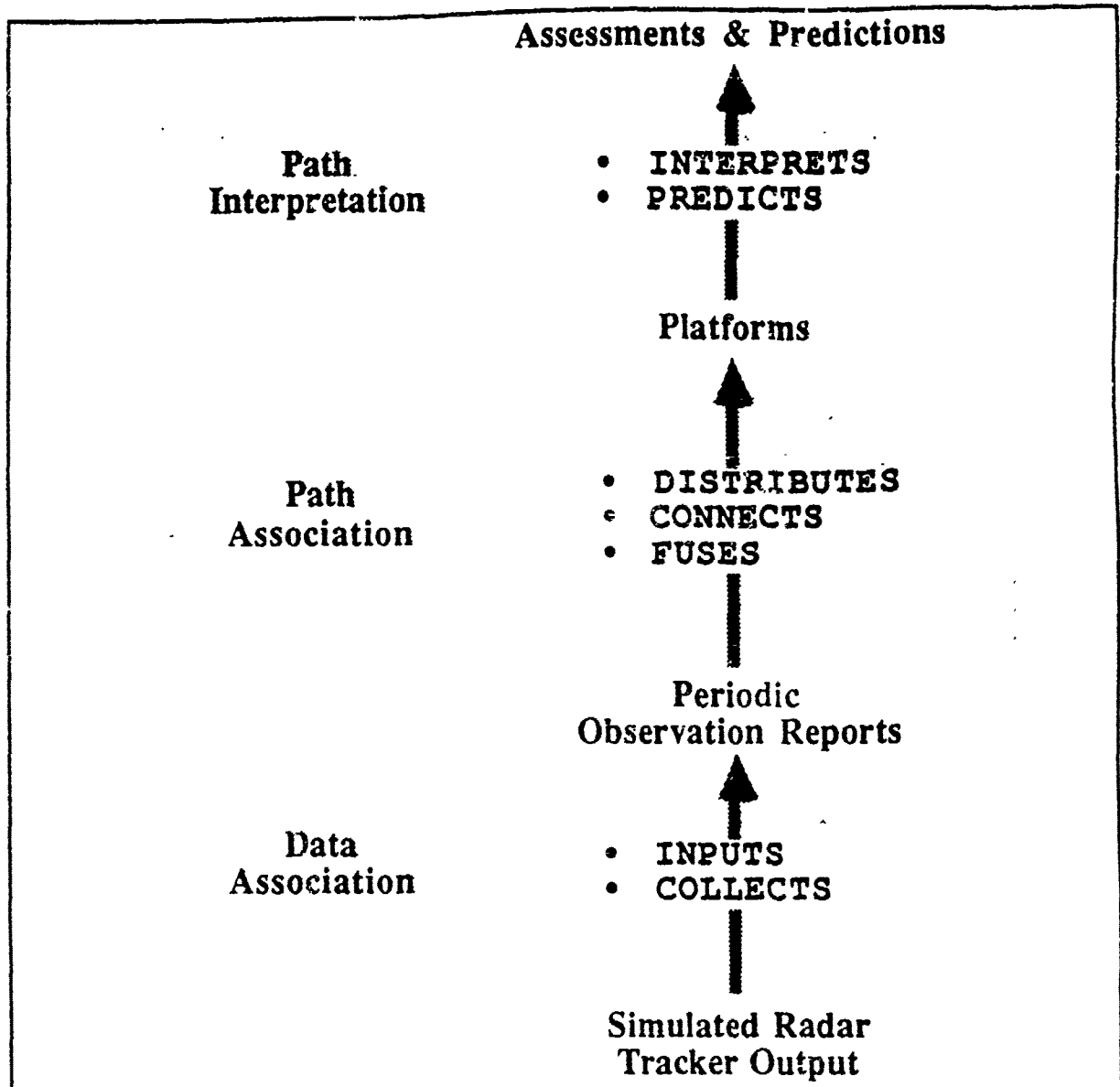


Figure 2.1 AIRTRAC Modules--Functions and Output

Let's now examine the function of each module individually.

2.4.1. Data Association

The Data Association module was completed in June of 1987 by Russell Nakano and Masafumi Minami [Nakano87].

2.4.1.1. Function

The primary function of Data Association is to accept, at regular time intervals, output information from all radar trackers that report on aircraft visible within the region at a particular scantime. It then identifies and collects together, in time-ordered sequence, all RTRs which belong to the same aircraft track (have the same track identifier as given by the trackers). Periodically, Data Association abstracts the individual RTRs it has gathered for a particular track into a *Periodic Observation Record* for that track.

2.4.1.2. Output: Periodic Observation Records

The output of Data Association are Periodic Observation Records (PORs). A POR is an abstraction of a sequence of RTRs from one radar for an individual aircraft track. It represents a regular portion of an aircraft's flight path as seen from a single radar. The *POR period* is the "length" (in scantime units) of every POR produced by Data Association; that is, the interval of time in which RTRs for a track are processed and abstracted into a POR.

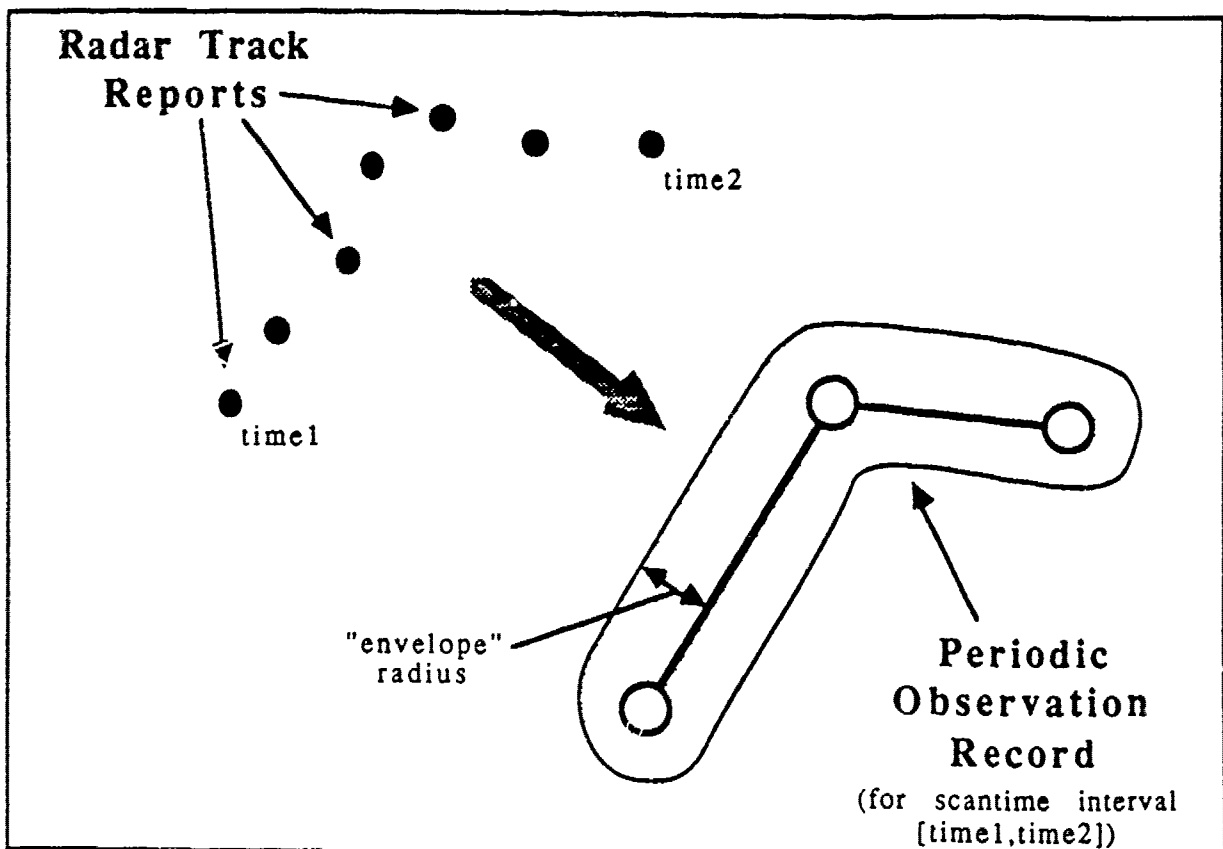


Fig. 2.2 How Data Association creates a Periodic Observation Record

Figure 2.2 presents a graphical representation of the abstraction process that takes place in the creation of a POR. Stated simply, Data Association creates a POR by fitting one or more line segments through the points given by RTR

coordinates. These "segments," or *line estimates*, are actually a sequence of <x-position y-position time> points, where the terminal point of one segment is also the beginning point of the next; i.e., they have the form

$$((x_1 \ y_1 \ t_1) (x_2 \ y_2 \ t_2) \dots (x_n \ y_n \ t_n)),$$

where n is the number of line estimates.¹ The line estimate radius, or error "envelope" of a POR, is conservatively calculated so as to completely contain the position covariances of each of the RTRs represented by the line estimates. The POR period is uniform across all PORs in the system; every POR represents a sampling of the same number of RTRs. Furthermore, the beginning and ending times of PORs are synchronized.

Part of the functionality of Data Association is detecting when an RTR with a new track ID has first been received and when an RTR of a known track ID fails to arrive during a scantime (when a track has been lost by the tracker). One of the notable attributes of a POR is its *status*, a keyword that Data Association assigns to a POR indicating its position (beginning, middle, or end) within an aircraft's track.² A status of *:create* means that the POR is the first (in data time) of a track, representing the beginning portion of that track. Similarly, an *:inactivate* status signals the final POR of a track. A POR with a status of *:update* is part of a continuing track, neither first nor last. A status of *:create-and-inactivate* means that both the first and the last RTRs of a track were received by Data Association during a single POR period.³

The information contained in a POR is listed in Table 2.2.

¹ In most cases the number of line segments fitted is exactly one. However, if there are sufficiently large changes in course executed by the aircraft within a POR period, these are reflected by additional line segments in the POR for that period.

² A *keyword*, in the usual sense of the term, is simply a constant symbol, commonly written with a leading colon.

³ From here on, a "*:create* POR" is understood to mean a POR whose status is either *:create* or *:create-and-inactivate*. Likewise, an "*:inactivate* POR" means one with status *:inactivate* or *:create-and-inactivate*.

Table 2.2 POR Information

A Periodic Observation Record produced by Data Association contains the following:

<i>track ID</i>	the identifier of the aircraft track assigned by the tracker to the RTRs in this POR
<i>radar ID</i>	the identifier of the radar site
<i>status</i>	the observation status of this POR; a keyword, one of (:create :update :inactivate :create-and-inactivate)
<i>aircraft type</i>	the type of the aircraft
<i>period begin time</i>	the beginning scantime of the POR period covered by this POR
<i>period end time</i>	the end scantime of the POR period covered by this POR
<i>actual begin time</i>	scantime of the earliest RTR of the track within this POR ¹
<i>actual end time</i>	scantime of the latest RTR of the track within this POR
<i>line estimates</i>	the sequence of line segments fitted through the RTRs for this POR
<i>line estimate radius</i>	the estimate (Ex Ey) of error associated with points in the line estimates
<i>velocity vectors</i>	velocities (Vx Vy) for the aircraft at the beginning and end of the POR
<i>velocity covariances</i>	estimates (Ex Ey) of the error associated with the given velocities

2.4.1.3. Data Association: The Real Story

An important admission concerning Data Association is necessary at this point. As it happens, we are not using the full-blown implementation of Data Association together with our Path Association module, for two very important reasons. First, the pragmatics: Due to CARE system changes made since the time Data Association was completed, the code for that module was rendered unreliable without a non-trivial amount of translation. Also, simulation time

¹ If the earliest RTR of a track is received during a POR period (the POR's status is :create), then the *actual begin time* of the POR may be different than the *period begin time*. The same relationship applies to the *actual end time* and the *period end time* of an :inactivate POR. For :update PORs the actual and period begin and end times are the same.

constraints made it impractical for us to run Path Association in conjunction with Data Association.

Second, the actual output of Data Association did not meet the demands of Path Association. Whereas our module required the periodic updating of track data, mostly for fusion purposes, Data Association produced only notification of the creation or inactivation of entire radar tracks. We could have chosen to make modifications to Data Association to make it produce the desired output, but given the other problems we faced in trying to make continued use of the module, we decided not to spend the time necessary for such an effort.

Instead, we simulate the output of PORs from what we believe to be a more appropriate Data Association. This is done by a manager object in the Path Association system called, not surprisingly, the *Data Association Simulator* (DAS).

2.4.1.4. Summary of Data Association Results

The results of the Data Association experiments demonstrated that performance of a concurrent program improves with additional processors, achieving a significant level of speedup in execution time. A complete report on Data Association and its experimental results can be found in [Nakano87].

2.4.2. Path Association

The Path Association module of AIRTRAC has been the main focus of our work this past year. This module completes the abstraction of radar output and fuses together data acquired from different radar sources to produce information that can be reasoned about by the next module, Path Interpretation.

2.4.2.1. Function

There are three main functions of Path Association:

- Distribution--accepting PORs from Data Association and distributing them to objects called *Flight Path Segments*, which collect all PORs with the same radar ID and track ID.
- Connection--"connecting" all Flight Path Segments with the same radar ID that seem to belong to the same aircraft flight path by associating them together in objects called *Observed Flight Paths*.
- Fusion--"fusing" Observed Flight Paths with different radar IDs that appear to be equivalent representations of a single aircraft's flight path by grouping them together in objects called *Platforms*.

2.4.2.2. Output: Platforms

The output of Path Association is a collection of Platforms, representing hypothesized aircraft in the real world as seen from one or more radar sources. A Platform incorporates all information about a single aircraft available through its individual Observed Flight Paths. Platforms are dynamic entities that are continuously being created, updated, terminated, and removed from the system to reflect the rapidly changing state of the region under observation, in which aircraft are constantly appearing and disappearing.

A detailed description of the Path Association module will be presented in Chapter 4.

2.4.3. Path Interpretation

The final module of AIRTRAC, not yet implemented, is Path Interpretation. We envision Path Interpretation as being the portion of AIRTRAC that performs higher-level reasoning functions.

2.4.3.1. Function

Path Interpretation responds to significant events that occur at the Platform level. It analyzes and interprets data contained in Platforms and makes assessments and predictions about the hypothesized aircraft represented by these Platforms.

2.4.3.2. Output: Histories, predictions, classifications, assessments

As mentioned earlier in Section 2.3, the output of Path Interpretation comes in the form of continuous information about aircraft within the region of interest. As events occur at the Platform level, Path Interpretation interprets these events, maintaining track histories and providing predictions, classifications, and other assessments of platform activity.

3. Concurrent Programming Techniques

This chapter describes design techniques for concurrent programs. We first briefly discuss the general approaches to concurrency: pipelining and replication. We then discuss a style of object-oriented programming well-suited to CARE-like parallel architectures.

3.2. Parallelism

Two techniques are known for exploiting parallelism: pipelining and replication.¹ Both of these techniques are used extensively in AIRTRAC.

3.2.1. Pipelining

A pipeline is a computational structure which consists of a sequence of stages through which a computation flows. It has the property that new operations can be initiated at the start of the pipeline while other operations are in progress through the pipeline. Thus any process which can be decomposed into sequential steps can be *pipelined*; an n -stage pipeline for an n -step process. Pipelining is therefore a very appropriate way of handling the flow of information between the levels of abstraction in an interpretation system such as Data Association or Path Association. On a multiprocessor, each stage is assigned to a separate processing unit which inputs from the previous stage and outputs to the next stage. An optimal n -stage pipeline has n times the throughput of one of its constituent stages. Unfortunately, a pipeline will be less efficient, however, if one or more stages in the pipeline

- 1) requires more time than other stages, or
- 2) depends on input from more than just the stage immediately before,

since either of these conditions will make some stages busier than others. The next section describes a remedy to the first condition.

3.2.2. Replication

Any computational structure, from a single process such as a stage in a pipeline to an entire pipeline, can be copied, or *replicated*, for increased performance. For example, search is well-suited to replication if the search space can be cleanly divided. A problem in which each of n replicated search mechanisms optimally handles $1/n$ th of the search space can theoretically be

¹ See [Brown 86] for a more detailed explanation.

solved n times faster than a single search mechanism working through the entire search space. Ideally, concurrency gained through replication is orthogonal to concurrency gained through pipelining. Nevertheless, just as performance gains in pipelines are limited by inter-stage dependencies, performance gains in replicated structures are limited by inter-structure dependencies. In the case of parallel search, for example, the need for synchronization in order to avoid fruitless search is a limiting factor.

3.1. Programming Style

In this section we discuss a specialization of the object-oriented LAMINA programming style which has evolved out of earlier work programming the CARE family of architectures (some predating LAMINA) and used in AIRTRAC. Applications are structured in terms of two types of objects--managers and subordinates--which communicate via messages. Computation is accomplished through the execution of explicit *trigger* methods executing on instances of these object types, and also through the execution of implicit *continuations* of these methods.

A general description of object-oriented LAMINA can be found in [Delagi 87b].

3.1.1. Manager objects

Managers, as the name implies, are objects which are responsible for tasks involving many other objects such as distribution of data, coordination of problem solving and dynamic object creation. Managers are often allocated statically, i.e., at initialization time, in which case the number required is determined a priori and depends on the particular application and its input data. Certain applications can benefit from the use of ephemeral or dynamically created managers. Dynamic managers, however, require additional control since the sphere of influence of each manager must be determined at run time and cannot be hard-wired in advance.

3.1.2. Subordinate objects

In keeping with the corporate analogy, subordinates are objects subject to the control of managers. Subordinates are created by managers; collectively, they typically contain most of the state of the system. Their tasks are dictated by one or more controlling managers. In general, subordinates can be *allocated* (created), *deallocated*, and *reallocated* many times over in the course of program execution, in response to prevailing needs of the application.

3.1.3 Continuations

Most computation takes the form of explicit (named) method execution on explicit manager and subordinate objects. Computation can also occur as an implicit *continuation* of a method. Such a continuation occurs in the context of the object executing the method (as defined by the values of instance variables and bindings in the environment). The method executing which spawned the continuation finishes normally and executes its next task. The continuation executes each time values are received on specified input streams. See page 12 of [Delagi 87b] for more details of the continuation mechanism.

4. Path Association

As described earlier, AIRTRAC is an attempt to develop a knowledge-based system in a multiprocessor hardware environment. This effort decomposes into two distinct, yet quite related, tasks: coming up with a solution to the aircraft tracking problem (the *knowledge-based* portion), and determining the appropriate software architecture to realize a correct implementation of our solution in a parallel, distributed environment (the *multiprocessor* portion). It must be stressed that we did not go about solving these tasks separately; rather, design decisions in each dimension were necessarily made with consideration for features and constraints of the other.

In this chapter we examine the Path Association module of AIRTRAC. We first present a complete functional description of Path Association in the context of a simple, yet non-trivial, example. This description involves the definition of the various LAMINA objects and their associated tasks, as well as a discussion of some of the important issues that arise in the problem domain. We then analyze the overall system software architecture and its implementation in a distributed-memory, message-passing multiprocessor environment. Next we describe the underlying design philosophy motivating our programming approach. Finally, we highlight the significant characteristics of Path Association that set it apart from previous research in the AAP, including Data Association.

4.1. Functional Description

4.1.1. The Example Scenario

The clearest way to explain the functionality of Path Association is to provide an example of a typical domain scenario and walk through it step by step. This will serve as the backdrop for the descriptions of the various objects in the system and their responsibilities. The scenario we will employ throughout this section is a relatively simple one: there are two radar trackers in our region (hereafter radar1 and radar2) observing aircraft of only one type (call it type-A). Though we would expect greater numbers of both these quantities in a more realistic situation, this example is quite appropriate for our expository purposes.

On the following page, Figure 4.1 shows a graphical representation of the PORs provided by Data Association in our chosen scenario--the input to Path Association.¹ The figure presents a "snapshot" of the current situation in the region of airspace under observation as seen from a God's-eye view at some particular time. There are two views of the region, one for each radar. The short

¹ This and future figures of the example scenario are actual screen images taken during an AIRTRAC simulation.

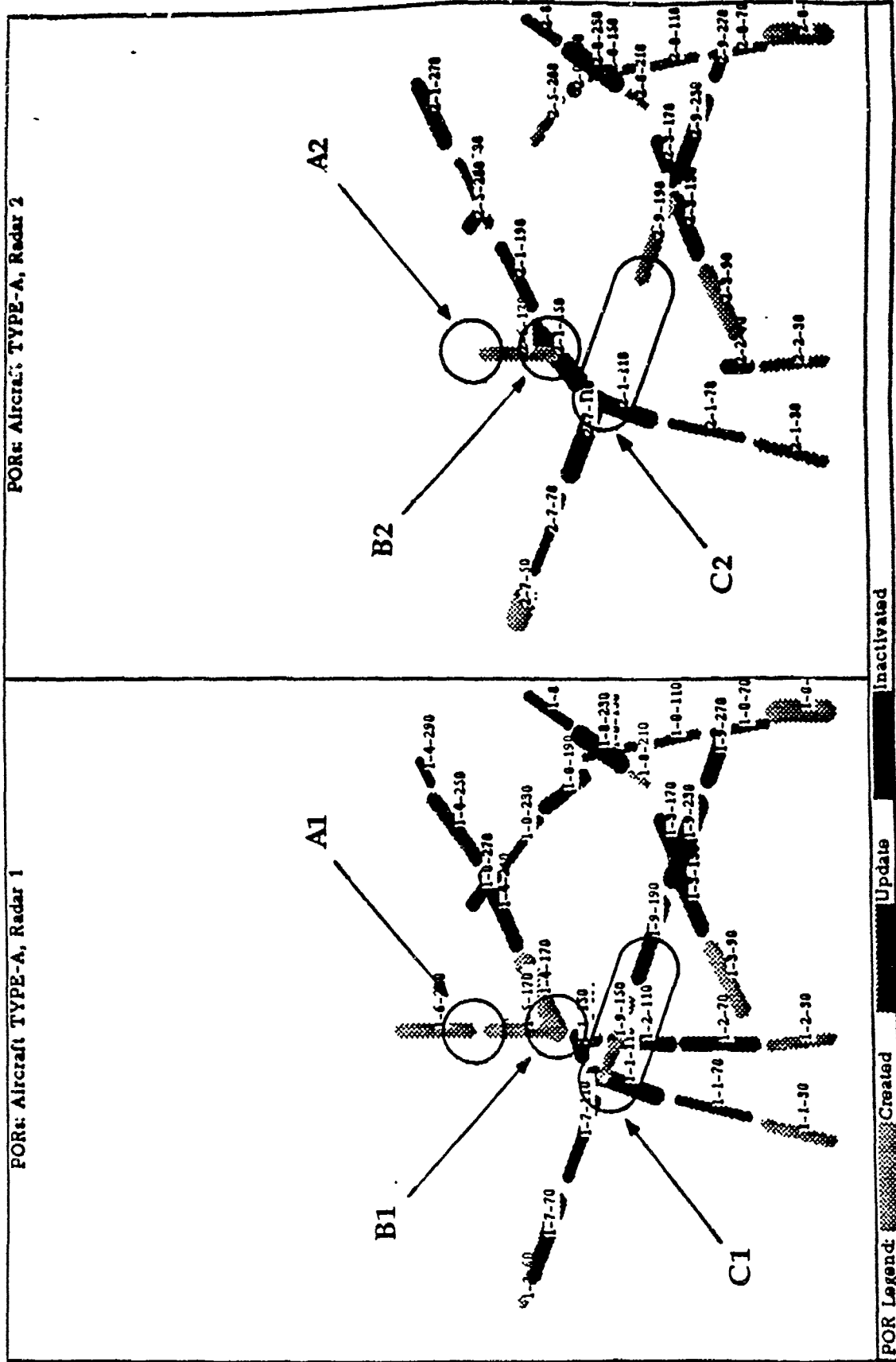


Figure 4.1 The Example Scenario

segments in the display represent PORs--light-shaded ones are :create PORs, dark-shaded ones are :update PORs, and black ones are :inactivate PORs. Beside each POR is a string of three numbers. Though these strings tend to clutter the display somewhat, they are nonetheless meaningful: the first number in a string gives the radar ID, the second the POR's track ID, and the third a scantime within the POR. Together these keys offer clues as to when and where tracks begin and end. For example, there are three tracks seen by both radar1 and radar2 that begin at the very bottom of the region at roughly the same time (around scantime 30). In addition, the width of each POR gives some indication of its line estimates radius.

Examining the display in Figure 4.1, one may note that, at the time the snapshot was taken (sometime after scantime 290), both radar1 and radar2 had been following a number of different tracks within the region. In fact, careful evaluation of the two radar views allows one to deduce that the example scenario involves the passage of 5 aircraft across the region. Of course, it is relatively simple for us as humans to perform this sort of visual comprehension. We can quickly and easily connect broken track segments and fill in missing pieces of tracks in one radar view with information from the other radar in order to come up with the "big picture" of the scenario--complete flight paths of all aircraft in the region. This is precisely the sort of real-time information fusion and interpretation that is so difficult to achieve in knowledge-based systems; it is the primary goal of Path Association.

4.1.1.1. What is interesting about this scenario?

There are many aspects of the example scenario that lend themselves to interesting observation. We shall choose a few of these to examine closely as we make our way through the different stages of Path Association. In Figure 4.1 there are 6 highlighted areas, labeled A1, A2, B1, B2, C1, and C2. A1 and A2 are actually the same physical area as seen from the two different radar views; the same applies for B1-B2 and C1-C2. Let's see what's happening in each of these areas at the POR level:

- A1) Track 1-5 has been lost and track 1-6 has begun.
- A2) Track 2-6 has been lost but no other track has been picked up.
- B1) A busy area: tracks 1-1 and 1-2 both end and tracks 1-4 and 1-5 both begin.
- B2) Track 2-1 shows no break while track 2-6 has begun.
- C1) Track 1-7 has ended and track 1-9 has been picked up.
- C2) Track 2-7 has been lost and track 2-9 has begun, with a large gap in between (both spatially and temporally).

4.1.1.2. What does this example tell us about radar capabilities?

Right away we can see that a radar tracker is not tremendously sophisticated. It often produces information that is unclear or even incomplete. One reason for this lies in the simple track extension algorithm the tracker uses to process observations. Any change, however slight, in the regular observation of an aircraft's track (a change in direction, a sudden shift in velocity—even a tiny glitch in the system) may result in a loss of the old track ID and the assignment of a new one. The tracker simply cannot distinguish between a true break in a flight path and a temporary loss of, or adjustment in, the observation of an aircraft; this sort of reasoning is part of AIRTRAC's job. As evidence, tracks have been lost and new ones started at several points during this scenario, even though the gap between them appears very small (A1, B1, C1).

A second source of incompleteness in radar data is due to the possible inability of a tracker to detect objects in a particular area--so-called radar "shadows." This might happen, for instance, if there is a large mountain within a radar's region of coverage. If the mountain lies between the radar site and a certain area, a plane flying into that area seems to disappear. Perhaps its track is picked up again with a new ID when the plane leaves the shadow, but the result is usually a large gap in its visible flight path. Such a situation seems to exist in our scenario in the output of radar2 near C2, where two crossing paths are lacking large sections of track data not missing in the observations by radar1 in area C1.

It is clear, then, that the information given to Path Association reflects the limited characteristics of the individual radar tracking systems, each of which sees the world in a different way. The purpose of the Path Association module is to combine the information it gains from all radar sources to overcome local radar shortcomings to produce a complete and accurate high-level description of the aircraft in the region.

4.1.2. Distribution

The first phase of Path Association processing involves the *distribution* of Periodic Observation Records from Data Association into dynamic objects called *Flight Path Segments*. This task is performed by manager objects called *Flight Path Managers*.

4.1.2.1. Flight Path Managers (FPMs)

A Flight Path Manager (FPM) handles the input PORs of a particular aircraft type and radar ID. Data Association knows which FPM to send a POR to on the basis of these two invariants. The number of FPMs in the system is thus equal to the number of distinguishable aircraft types times the number of different radar sources. (In a typical scenario, for example, in which we observe 3 different types of aircraft from 3 radar trackers, there would be 9 FPMs.) The function of an FPM

is to accept PORs from Data Association and to process them according to their track ID and status.

Each FPM maintains a list of the track IDs for PORs it has received, along with the names of the Flight Path Segments it has allocated for tracks. A *new* track ID is one for which no previous POR has been received by the FPM.

4.1.2.2 Flight Path Segments (FPSs)

Flight Path Segments (FPSs) are dynamic objects subordinate to an FPM. They represent a collection of PORs with the same track ID (and therefore same radar ID and aircraft type). Among an FPS object's slots are those listed in Table 4.1. Included in this list is the FPS's *status*, a keyword which is either *:active* or *:inactive*. The status *:active* reflects the fact that the FPS is continuing to receive *:update* PORs by way of its FPM. An FPS sets its status to *:inactive* when it receives an *:inactivate* POR.

Table 4.1 Flight Path Segment

The information in a Flight Path Segment includes the following:

<i>track ID</i>	the identifier assigned to this track by the radar tracker
<i>radar ID</i>	the identifier of the radar site from which the track is being observed
<i>status</i>	track status; a keyword, one of (<i>:active</i> <i>:inactive</i>)
<i>aircraft type</i>	the type of the aircraft
<i>line estimates</i>	the sequence of POR line estimates
<i>initial velocity</i>	velocity (Vx Vy) of track at time of creation
<i>final velocity</i>	velocity (Vx Vy) of track at time of termination
<i>OFP parent</i>	name of Observed Flight Path parent

Why are FPSs not enough to represent an entire flight path of an aircraft? The answer, of course, lies in the inability of a radar tracker to maintain a consistent lock on the aircraft's position as it makes its way through the region. As mentioned before, it is simply too common for a radar tracker system to lose a track due to a number of reasons (sharp turn, radar shadow, etc.). There must be some object superior to an FPS that represents a complete flight path, overcoming the natural incompleteness of radar tracking systems; in fact, we will describe this object shortly.

4.1.2.3. The Distribution Process

Figure 4.2 shows the process of distribution in terms of the LAMINA objects involved and the communication among them.

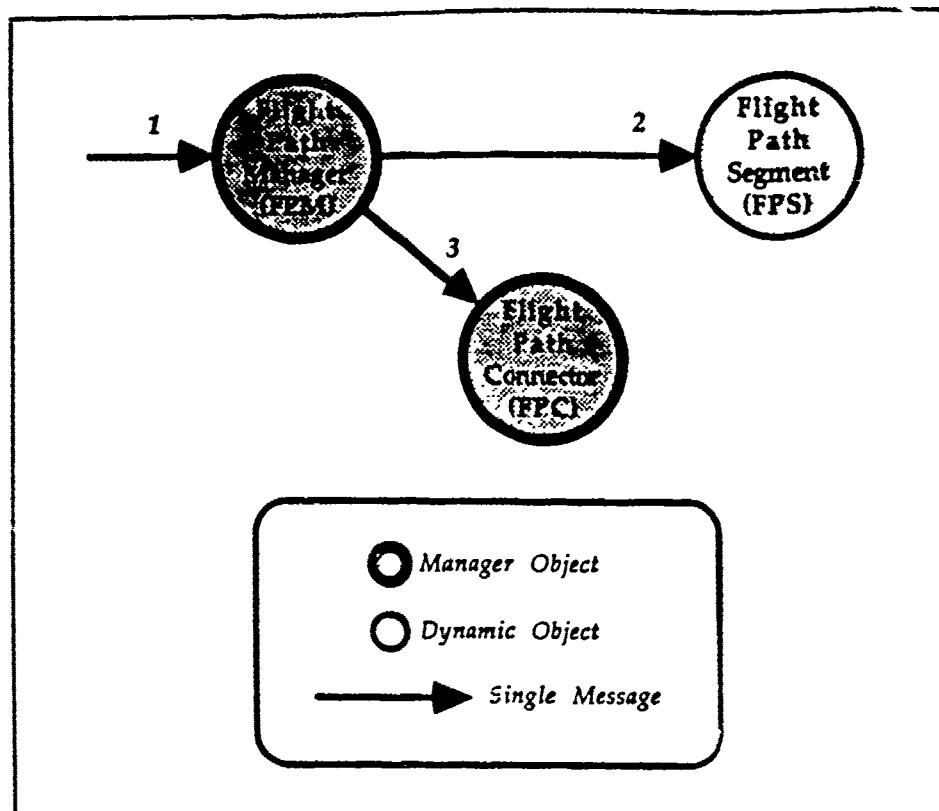


Figure 4.2 The Distribution Process

When a POR arrives from Data Association at the FPM in a *:distribute-POR* message [1]¹, the following takes place:

- a) If the track ID of the POR is a new one, then a new FPS is created and sent an initialization message which contains the new POR [2].
- b) Otherwise, if the POR's track ID has previously been registered with the FPM, then the POR is sent directly to the track's FPS in an *:add-new-POR* message [2].

¹ In the description of the distribution process, and that of connection and fusion to follow, a bold number in brackets refers to the corresponding LAMINA message(s) in the figure for that process. In this case, [1] denotes the message received by the FPM in Figure 4.2 (shown as an arrow labeled "1").

- c) If the status of the POR is either *:create* or *:create-and-inactivate*, the FPM knows that it has received the first POR of the FPS.¹ In this case a *:connect-at-creation* message is sent to the Flight Path Connector associated with this FPM (the one that handles FPSs of the same aircraft type and radar ID) [3]. This message contains all information pertaining to the beginning of the FPS (including time, position, and velocity, as well as its name and remote address), and initiates a connection procedure as discussed in Section 4.1.3.
- d) If the status of the POR is either *:inactivate* or *:create-and-inactivate*, the FPM knows that it has received the last POR of the FPS. In this case a *:connect-at-termination* message is sent to the associated Flight Path Connector with all POR information about the end of the FPS in addition to its name and remote address [3].

4.1.2.4. Distribution in the Example Scenario

How does our example scenario appear at the distribution level? Figure 4.3, on the next page, shows the graphical display of FPSs that exist in the system.² Each FPS is seen as an unbroken line segment with an identifying string printed near its origin. [Note: The boxes drawn in each region are graphical aids in the connection process; ignore them for now.]

The display shows how the PORs given to Path Association have been collected by the FPSs. As expected, each FPS line is drawn from the beginning of its *:create* POR to the end of its *:inactivate* POR (as they appear in Figure 4.1). The areas of interest we are following exhibit no further development at this point; whenever a track is picked up an FPS begins, and whenever a track is lost an FPS ends. The really exciting stuff is yet to come.

4.1.3. Connection

Looking at Figure 4.3 we can see obvious locations where tracks were broken for some reason and new ones picked up soon afterwards. To our eyes it is clear that many of these individual tracks belong together in the same entire flight path. It is the task of the next stage of Path Association, connection, to determine which FPSs should be associated together to form complete observed flight paths.

¹ Note that the first POR of an FPS is not necessarily the one received earliest in time at the FPM. It is possible, due to message disorder, for an *:update* or *:inactivated* POR of an FPS to arrive before the *:create* POR. "First," in this context, always denotes a POR with status *:create* or *:create-and-inactivate*. Likewise, the "last" POR of an FPS always has an *:inactivate* or *:create-and-inactivate* status.

² This and the other displays of Path Association graphics show the different levels of computation at the same instant in time -- a system-wide snapshot.

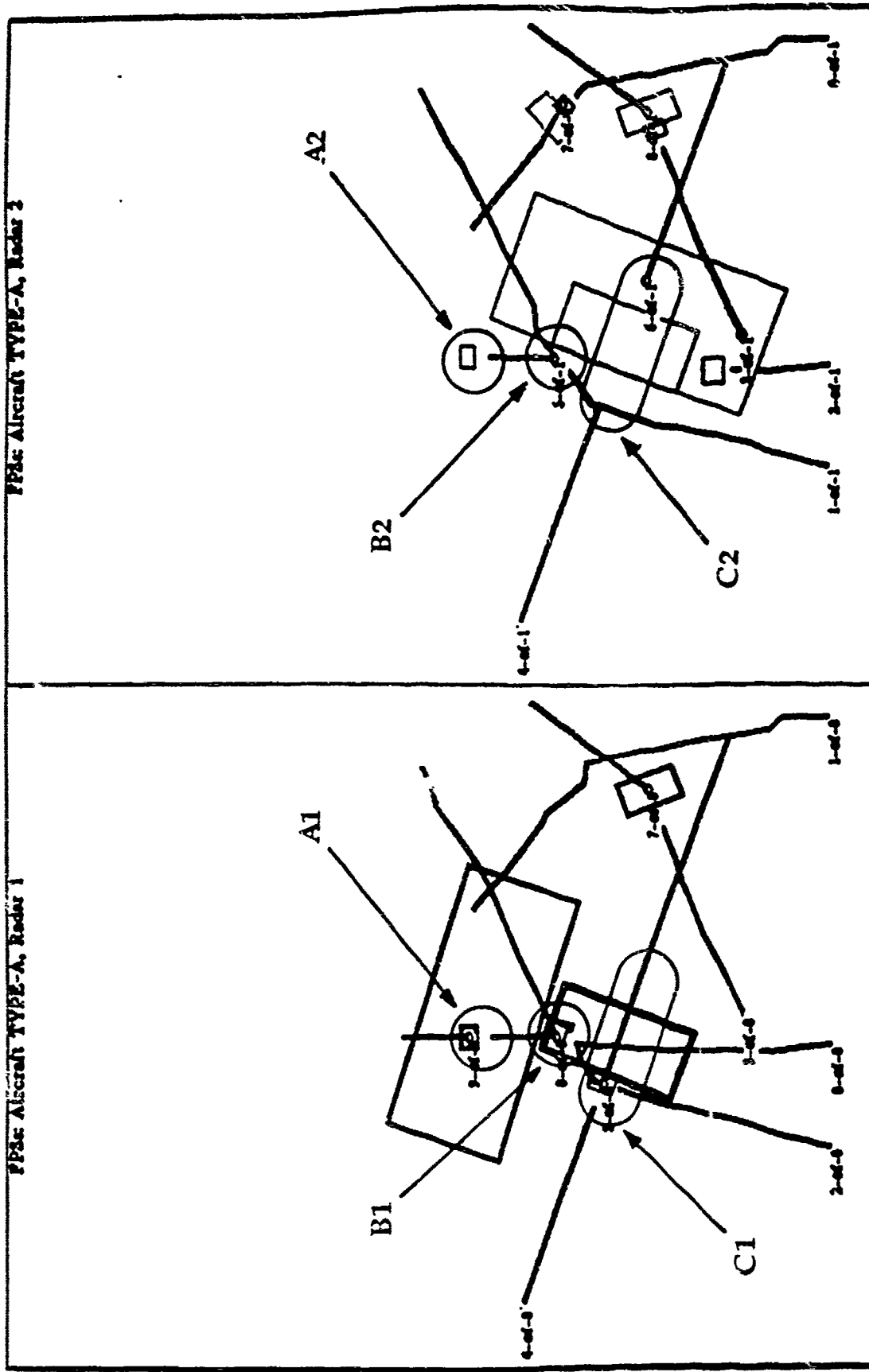


Figure 4.3 Distribution in the Example Scenario

The coordination of this search is performed by manager objects called *Flight Path Connectors*.

4.1.3.1. Flight Path Connectors (FPCs)

A Flight Path Connector (FPC) receives information about the creation and termination of all FPS tracks of the same aircraft type and radar ID from its associated FPM. (There is a one-to-one correspondence between FPMs and FPCs.) It attempts to "connect," or match, pairs of FPSs that logically could be parts of the same flight path by determining whether one FPS is potentially a continuation of another's track, using criteria detailed below. All connected FPSs are grouped under common objects called *Observed Flight Paths*.

An FPC handles *only* FPSs that share the same aircraft type and are reported by the same radar tracker. It would not make sense to try to connect segments from different aircraft types or across different radars (multi-sensor fusion of corresponding path segments is performed in the final stage of Path Association). Each FPC maintains a list of the FPSs it has been notified of, including all data relevant to the beginning and/or end of each FPS (time, position, velocity). It also keeps a list of the Observed Flight Path objects it has created, together with the names of FPSs linked together under them.

4.1.3.2. Observed Flight Paths (OFPs)

An Observed Flight Path (OFP) is a collection of one or more FPSs, connected by an FPC, that constitutes a complete flight path as seen from a single radar source. (Actually, the collection is not of the FPS objects themselves but of names and handles to these FPSs.) Every FPS is associated with exactly one OFP, even if does not connect with any other FPSs.

The information contained in an OFP object is listed in Table 4.2. Included in this list are slots for "connections" to other OFPs, a concept to be explained a bit later.

Table 4.2 Observed Flight Path

The information in an Observed Flight Path includes the following:

<i>radar ID</i>	the identifier of the radar site from which the track is being observed
<i>status</i>	flight path status; a keyword, one of (:active :inactive)
<i>aircraft type</i>	the type of the aircraft
<i>FPS children</i>	list of names of FPSs that make up this complete OFP
<i>creation connections</i>	list of names of earlier OFPs connected to this one
<i>termination connections</i>	list of names of later OFPs connected to this one
<i>UFP parent</i>	name of associated Unfused Flight Path (if one exists)
<i>P parents</i>	list of names of Platforms of which this OFP is a part
<i>PM</i>	Platform Manager that handles fusion searches for aircraft of this type

4.1.3.3. The Connection Criteria

What are the criteria an FPC uses to determine if two FPSs should be connected and made part of the same OFP? Essentially, FPC combines data from FPS endpoints with knowledge it has stored in a model of the aircraft to project regions where it expects possible continuation for a track, which it then uses to search for other FPSs whose beginnings or endings fall into that area.

4.1.3.3.1. Aircraft model

The *aircraft model* used in the connection process by an FPC is a frame-like data structure that incorporates explicit knowledge about cruising characteristics and maneuvering capabilities of the type of aircraft handled by the FPC. The items found in the model of an aircraft are listed in Table 4.3

Table 4.3 Aircraft Model

The model of each aircraft type which is used by the Flight Path Connectors during connection contains the following information:

<i>name</i>	the name of the aircraft type
<i>min cruising speed</i>	the minimum straight-line velocity needed to remain airborne
<i>max cruising speed</i>	the maximum straight-line velocity capability
<i>avg cruising speed</i>	the average (or expected) cruising speed for an aircraft of this type
<i>max landing speed</i>	maximum speed at which the aircraft is capable of landing
<i>max acceleration</i>	the maximum possible straight-line acceleration
<i>max deceleration</i>	the maximum possible straight-line deceleration
<i>min turning radius</i>	the minimum possible turning radius (at both minimum and maximum speeds)
<i>turning radius function</i>	a function relating turning radius to speed

4.1.3.3.2. Continuation region

Given two FPSs, an FPC checks for possible connection between them in the following manner. A pair of FPSs connect, of course, between the end of one segment and the beginning of another. Therefore, the FPC needs to know the latest reported position and velocity of the earlier FPS and the scantime at which it terminated. Likewise, the FPC needs to know the original position and velocity and the time of creation for the later FPS.

The first check for possible connection is to see if the difference in scantime units between the termination time of the earlier FPS and the creation time of the later FPS (call it *delta-T*) is within 1 and some maximum *connection search interval*, a preloaded system parameter that limits the time gap between connected FPSs to a reasonable length. If this test is passed the FPC then uses

- a) the termination position and velocity data of the earlier aircraft,
- b) *delta-T*, the time gap between the segments, and
- c) the performance figures from the aircraft model,

to compute a *continuation region* in which the creation position of the later FPS must fall if it is to successfully connect, as shown in Figure 4.4.¹ If the later FPS does indeed begin within this continuation region, then the two FPSs are considered to have met the connection criteria.²

Returning to Figure 4.3, we now understand what the boxes drawn at different points in the region represent—they are continuation regions computed during the connection process. By noticing the small circles (denoting FPS creation positions) that lie within several of these continuation regions we can predict FPSs that are likely candidates for the FPCs to connect. We shall see a little later on how things actually turn out.

¹ We omit the actual equations used for the computation as they are not directly relevant to the discussion at hand.

² There is actually one final check that takes place, in which the angle between the termination velocity vector and the creation velocity vector must not exceed some *maximum angular deviation* computed from the termination data and the aircraft model figures.

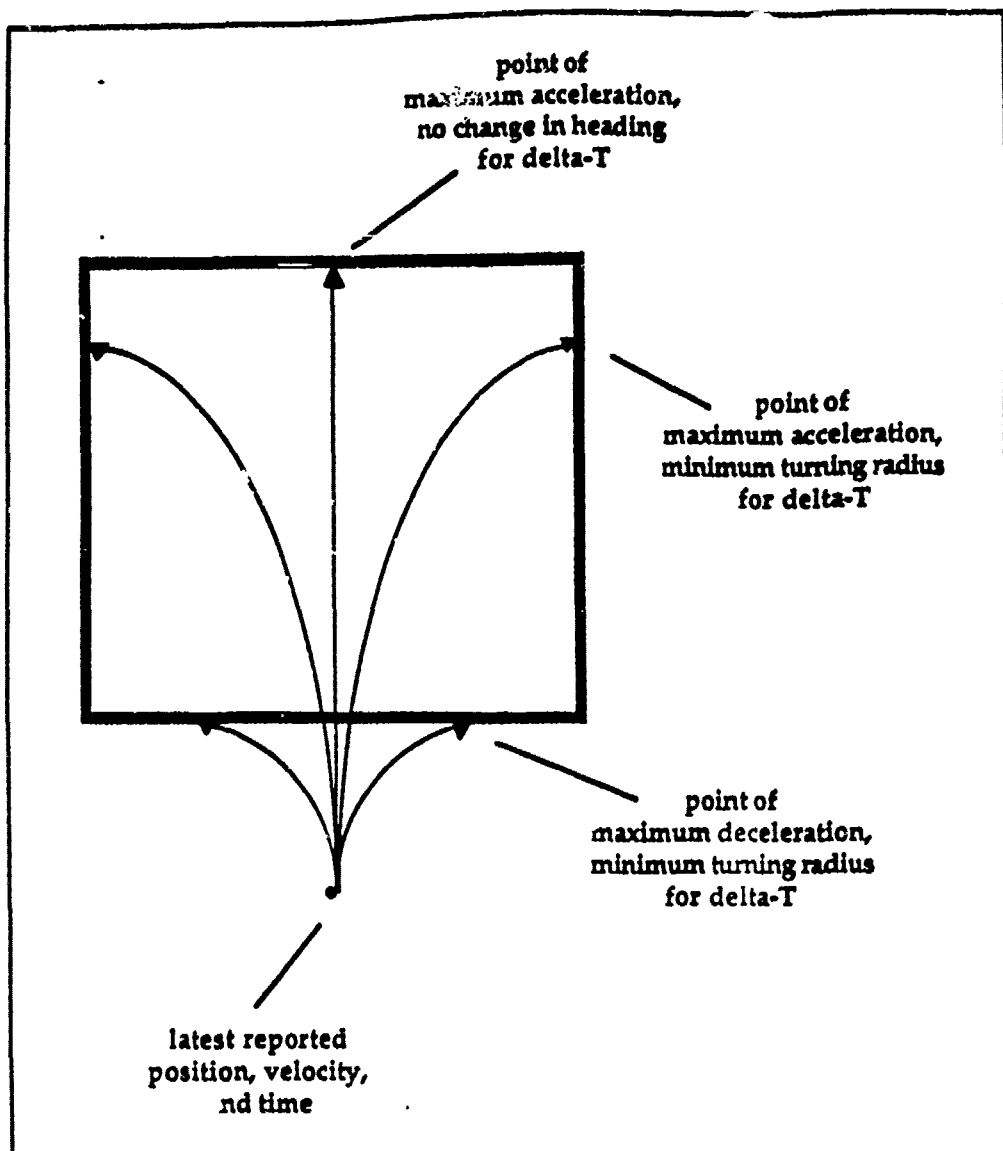


Figure 4.4 Calculating a Continuation Region

4.1.3.4. The Connection Process

Figure 4.5 shows the process of connection in terms of the LAMINA objects involved and the communication among them.

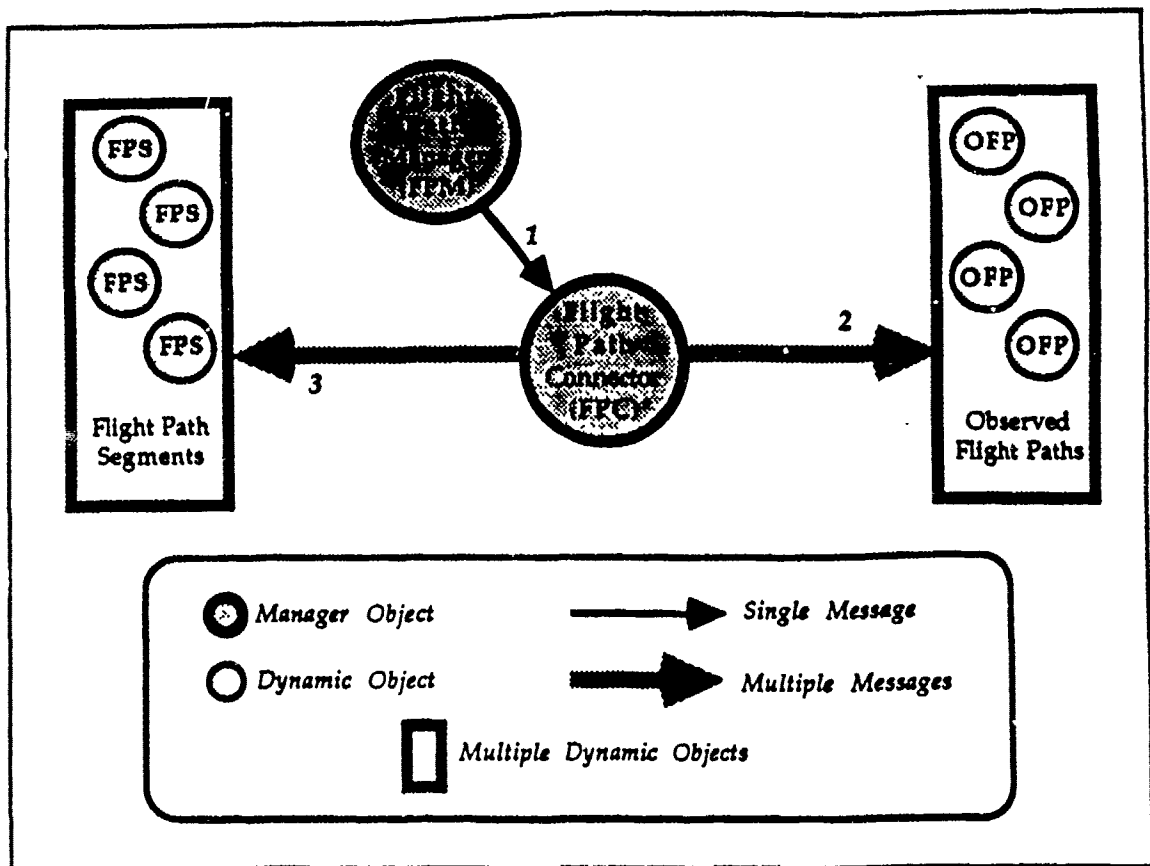


Figure 4.5 The Connection Process

Connection takes place when an FPM receives either a *:create* POR or an *:inactivate* POR for an FPS and notifies the FPC responsible for FPSs of the corresponding aircraft type and radar ID in a *:connect-at-creation* message or a *:connect-at-termination* message, respectively. In the following two sections we outline the steps taken in both of these cases.

For convenience, let us define a *created* FPS as an FPS for which a *:connect-at-creation* message has been received and processed by the responsible FPC. Similarly, a *terminated* FPS is an FPS for which a *:connect-at-termination* message has been received by the FPC.

4.1.3.4.1. Connect-at-Creation

When an FPC receives a *:connect-at-creation* message with information about the beginning of a track from an FPS's *:create* POR [1], the following takes place:

- a) The FPC searches its list of FPSs for those that have been terminated but not yet connected with any later FPSs, collecting those that connect with the newly created FPS.¹
- b) If there are no connecting terminated FPSs from a), then the FPC:
 - creates a new OFP and sends to it an initialization message that establishes the created FPS as its only child [2]
 - sends a *new-OFP-parent* message to the created FPS with the name and handle of the new OFP [3]
- c) If there is exactly one such terminated FPS from a), then that FPS and the newly created one are connected by the FPC:
 - an *add-new-FPS* message is sent to the OFP parent of the terminated FPS with the name and handle of the created FPS [2]
 - a *new-OFP-parent* message is sent to the created FPS with the name and handle of the terminated FPS's OFP parent [3]
- d) If there is more than one connecting terminated FPS from a), then this is an ambiguous connection case [see Section 4.1.3.5.]
- e) The FPC next searches its list of FPSs for those that have been terminated and previously connected to a later FPS (not including those from c) and d)), again collecting those that connect with the newly created FPS.
- f) If there are any such terminated FPSs from e), then this is an ambiguous connection case [see Section 4.1.3.5.]

4.1.3.4.2. Connect-at-Termination

Usually two FPSs that belong together in the same OFP arrive in order at the FPC. More precisely, if two FPSs should connect with each other, it is usually the case that the FPC knows about the termination of the earlier FPS before it learns of the creation of the later FPS. However, because messages sometimes arrive out of order, it is entirely possible that the FPC will be notified of these events in reverse order. If we were just to rely on a method of establishing connections between FPSs only when one was created, we would never be able to resolve such an out-of-order situation; an FPC simply could not connect a created FPS with a terminated FPS that it hasn't yet been told of.

Our solution to this difficulty is to perform connection between FPSs in both directions in time. In the expected case, the FPC searches backward in time for

¹ In this discussion two FPSs "connect" if they meet the connection criteria set forth in Section 4.1.3.3.

terminated FPSs that connect with an FPS given by a *:connect-at-creation* message, as described in the previous section. The FPC also searches forward in time for possible connection to created FPSs when a terminated FPS arrives in a *:connect-at-termination* message. The connect-at-termination procedure is similar to that of connection-at-creation, and it ensures that slightly out-of-order messages never results in a missed connection.

When an FPC receives a *:connect-at-termination* message with information about the ending of a track from an FPS's *:inactivate* POR [1], the following takes place:

- a) The FPC searches its list of FPSs for those that have been created but not connected with any earlier FPSs, collecting those that connect with the recently terminated FPS.
- b) If there are no connecting created FPSs from a), then the FPC does nothing else, except in one case: if the terminated FPS is not yet associated with an OFP (which can only occur, from the perspective of the FPC, if the FPS was terminated before it was created, another possible result of disordering). If this is the case, then the FPC:
 - creates a new OFP and sends to it an initialization message that establishes the terminated FPS as its only child [2]
 - sends a *.new-OFP-parent* message to the terminated FPS with the name and handle of the new OFP [3]
- c) If there is exactly one such created FPS from a), then that FPS and the newly created one are connected by the FPC. This is more difficult than in the connect-at-creation case c) because both FPSs have a parent OFP (though it is possible, as in b), that the terminated FPS has not yet been created and does not have an OFP parent.). To execute the connection the two parent OFP's must be *merged*, as follows:
 - an *:add-new-FPS* message is sent to the OFP parent of the created FPS with the names and handles of the terminated FPS and its sibling FPSs of the current OFP parent [2]
 - a *.new-OFP-parent* message is sent to the terminated FPS and its transferred FPS siblings with the name and handle of the created FPS's OFP parent [3]
 - the old OFP parent of the terminated FPS is deallocated, leaving only the terminated FPS's new OFP parent
- d) If there is more than one connecting created FPS from a), then this is an ambiguous connection case [see Section 4.1.3.5.]

4.1.3.5. Ambiguous cases

Occasionally an FPC finds that there is more than one connection possible for a given FPS, especially given the conservative nature of the continuation region, calculated so as to eliminate only those connection pairs which are clearly unreasonable. A typical situation in which this might occur is when two aircraft pass by a particular point in the region at roughly the same time and their flight paths cross. From a radar tracker's perspective, a crossing often appears as a break in both flight paths, resulting in the termination of one FPS and the creation of a new one for each path. An example of this can be seen in area B1 of Figure 4.3. The difficulty is that, by the connection criteria, the first FPS of one or both tracks may connect with the second FPS of *both* tracks. There is no notion of a "best" connection; any path extension passing the connection criteria is considered as good as any other, so there is no way to determine which is the true connection.

What can the FPC do in such a situation? In our object definitions we require that an OFP be composed of only a single path of connected FPSs—it cannot contain multiple connections between constituent FPSs.¹ We deal with the problem of ambiguity by introducing the concept of *connected OFPs*. In a case in which there are multiple possible connections between an FPS and other FPSs, each FPS is assigned to its own OFP and the different OFPs are then "connected" on their own level by establishing a connection relationship between them. That is, the earlier FPS's OFP is said to have the later FPS's OFP as a *termination connection*, and the OFP of the later FPS has the OFP of the earlier FPS as a *creation connection*. Each OFP maintains lists of its creation and termination connections to other OFPs.

By letting any ambiguity present following connection remain unresolved in the form of connected OFPs, we are counting on fusion to detect incorrect associations and establish the true complete flight path using information from other radar sources tracking the same aircraft. We shall see if fusion can, in fact, resolve these ambiguous cases.

4.1.3.6. OFP Updates

After an FPS is notified that it has been associated with an OFP, it begins to forward *update* messages to its parent OFP. These are messages that report when new track information has been obtained by an FPS so that an OFP is kept up-to-date of events at the FPS level. An *:FPS-update* message is sent to an OFP every time the FPS itself receives an update message from its FPM in the form of an *:update* POR. Included in each *:FPS-update* message from the FPS are the POR lines estimates it is provided in the *:update* POR. The OFP, in turn, forwards all

¹ The reason for this strict definition is, to a great degree, dictated by a cleaner representation of platforms at the fusion level, but this is mostly an implementation issue beyond the scope of this discussion.

update information onto its associated higher-level object, a Platform or Unfused Flight Path, as described in Section 4.1.3.6. An OFP does not save any FPS line estimate information within its own state.¹

4.1.3.7. Connection in the Example Scenario

Let's now take a look at the results of the connection process in our example scenario. On the following page, Figure 4.6 shows the graphical display of OFPs in the system at the same snapshot in time. Each OFP is drawn as a continuous line segment, with a string identifier near its origin. Of special note are the lighter-shaded lines drawn between several of the OFPs near areas B1 and C1 of radar1. They signify that the OFPs they join are connected OFPs, meaning there are a few points of ambiguity present in the scenario at the connection level.

Looking at our areas of interest we notice the following:

- A1) The two FPSs in Figure 4.3 have been together formed an OFP, a straightforward case of connection.
- A2) With the later FPS missing, the earlier FPS stands alone in its own OFP.
- B1) As expected, a bit of confusion. With several FPS connections possible, both sides of the cross have their own OFPs, with OFP connections established between them. Interestingly, there also seem to be OFP connections made between one or both of the higher OFPs in B1 and the OFP coming from the side of the region in C1. This is a case in which the support for a connected path seems weak but the connection criteria were met anyway.
- B2) Nothing unexpected. One OFP passes over the point at which another OFP is created, just as their corresponding FPSs did.
- C1) What seems likely to us to be a strong connection of FPSs in an OFP that travels straight across the region is dealt with at the connection level as two connected OFPs. This is due to the cross-over coincidence in B1.
- C2) A perfect example of the power of connection. Despite the big gap between the two FPSs they were able to be connected into a single OFP crossing the region.

¹ The reason for not storing line estimates is an issue of *cache consistency*. The OFP can never be certain that the line estimate information it has been presented is complete and up-to-date. Therefore, in reporting this information to initiate the fusion process (described in Section 4.1.4), the OFP is forced to request current track data from its FPS, eliminating the need to maintain this information itself.

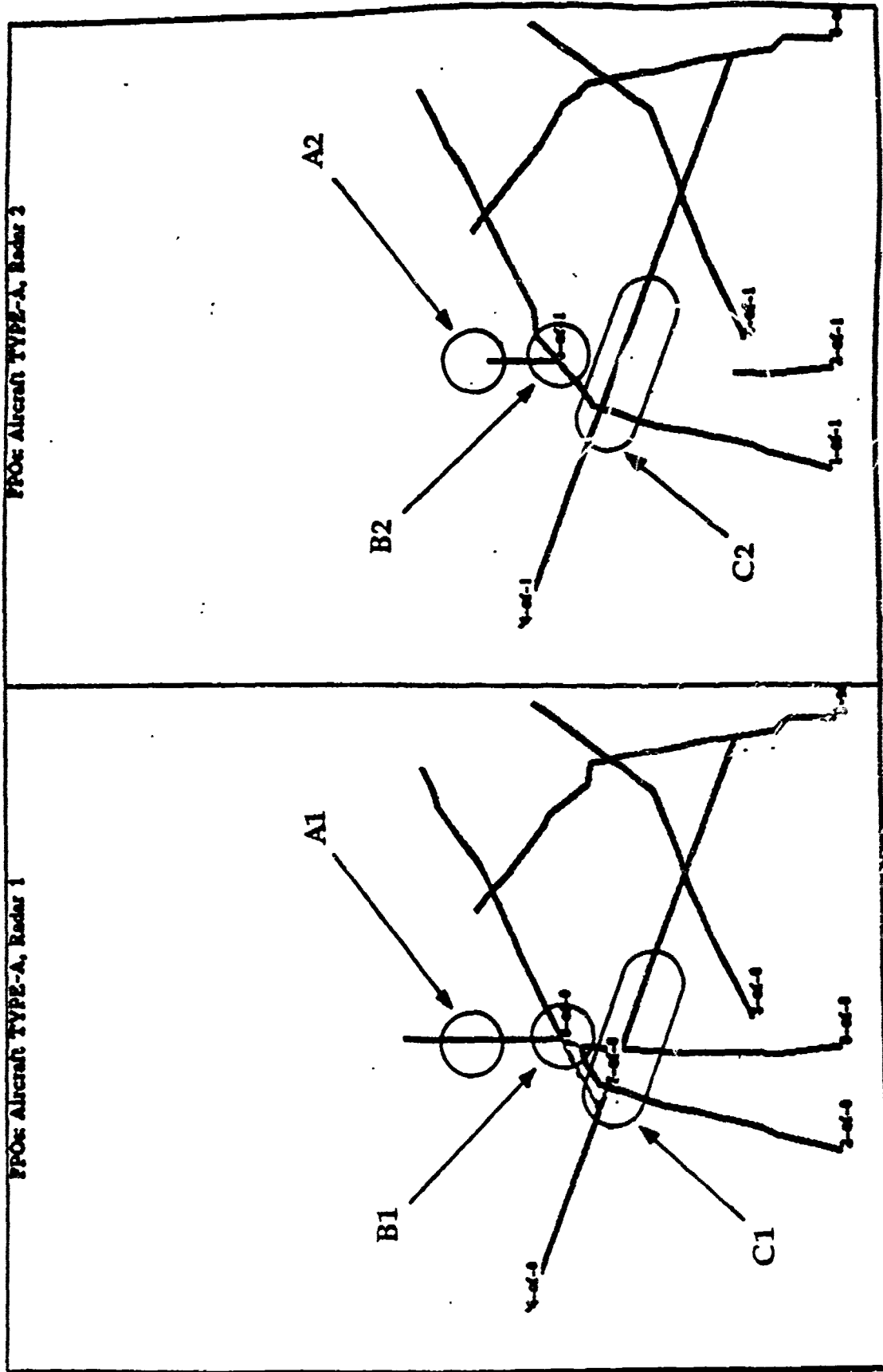


Figure 4.6 Connection in the Example Scenario

4.1.4. Fusion

In the distribution and connection phases all reasoning about flight path data is performed on PORs and FPSs observed from the same radar source. Up to now there has been no use of information from other radars to help resolve missing data conflicts present in radar tracks from individual radars. In the fusion stage of Path Association, however, information from all radar tracking sources is brought together to form the best available estimate of the flight of the real-world aircraft.

The dynamic objects produced by the fusion process are abstractions called Platforms. One of the main goals of fusion is to associate each and every OFP with at least one Platform so that its flight path data can be used in later AIRTRAC functions in Path Interpretation. Fusion takes place in a distributed search method coordinated by manager objects called Platform Managers.

4.1.4.1. Platform Managers (PMs)

Platform Managers (PMs) are, in the truest sense of the word, managers. They only oversee the real work that they delegate to their subordinates, unlike FPMs and FPCs which handle the entire computational chore by themselves at their level of processing.¹ PMs control the creation of Platforms and coordinate the fusion search among the many distributed, dynamic Platform objects; the actual fusion of information from different radar sources is performed by the Platforms themselves.

Each PM manages the fusion of OFP information for a particular aircraft type. It maintains a list of existing Platforms representing aircraft of that type in the system. (In addition, a PM keeps a list of a second collection of subordinate objects called Unfused Flight Paths, the nature and function of which will be presented later.)

4.1.4.2. Platforms (Ps)

A *Platform* (P) represents the hypothesis of a real-world aircraft passing through the region of airspace. It is a collection of supporting OFPs that each contribute their own radar's view of the aircraft's flight path to form a *composite Flight Path* (or simply *composite-FP*). The composite-FP is calculated from the track data from the FPSs of all individual OFPs using a best-fit approximation to

¹ The fact that the PMs are the only managers objects in the system that distribute a significant amount of their workload to subordinate objects reflects a design decision that we made to reduce the possibility of bottlenecks (performance hotspots) at the PM during the computationally more complex process of fusion. More is said about the consequences of this decision in Chapter 6.

arrive at the most informed estimate of the aircraft's complete and true flight path.

Platforms are the final output of Path Association. They contain the information listed in Table 4.3.

Table 4.3 Platform

The information in a Platform includes the following:

<i>status</i>	status of the hypothesized aircraft represented by the P; keyword, one of (:active :inactive)
<i>aircraft type</i>	the type of the aircraft
<i>OFP children</i>	list of names (and attributes) of all Observed Flight Paths that make up the P
<i>composite-FP</i>	the composite flight path computed from a best-fit of all line estimates of individual OFPs of the P
<i>PM</i>	the name of the Platform Manager in charge of Ps of this aircraft type

4.1.4.3. The Fusion Criteria

In this section we examine the criteria used to determine if tracks from different radars represent equivalent views of the same aircraft's flight path and should therefore be collected together in a single Platform. In the description of the fusion processes to follow, mention is made several times of an OFP "fusing" with a P. What exactly does this mean?

When a new Platform is created, its initial composite FP consists of the line estimate information from all of the FPSs in the OFP that caused its creation during the fusion process. Thus, the composite is a sequence of line segments (pairs of endpoints), each with its own estimate of error given by the POR line estimate radius. There may be gaps in this sequence for which no track data is available corresponding to scantimes between connected FPSs. In addition, included with each segment in the composite is a list of radar IDs represented by that segment.

During the fusion process, Platforms are requested to try to fuse OFPs with their current composite FP, and a fusion result signalling the success, failure, or indeterminacy of the fusion is returned. A result of :MATCH is returned if all three of the following conditions are satisfied:

1) The OFP and the composite overlap in time for a minimum of one scantime. That is, there exists at least one scantime which is represented in line segments in both the OFP and the composite.

2) Every point in the OFP "matches" the corresponding (same scantime) point in the composite; i.e., their error radii intersect at each and every point. (If any missing point is between the endpoints of a segment in either flight path, it is calculated by interpolation.)

3) The OFP and the composite do not overlap at any point in time for which the composite already has incorporated data from the OFP's radar (the OFP's radar ID is in the list for that particular segment). An OFP cannot possibly share a Platform with another OFP observed from the same radar; a point conflict so described would indicate two such OFPs.

Figure 4.7 gives a before-and-after look at the fusion of corresponding portions of an OFP and a Platform's composite FP whose result is :MATCH. We see that each point in the OFP matches the corresponding point in the composite because their radii intersect. The resulting new composite point for each pair is a weighted average of the locations of the two points.¹ The radius of error for each new point is computed from the new point location and the intersecting radii. What is important to note in this process is that, as each new OFP is fused with a P, the composite FP for that P becomes more defined as the error radii error of its constituent points get smaller.

¹ The "weight" of a composite segment is dictated by the number of radars already represented in that segment.

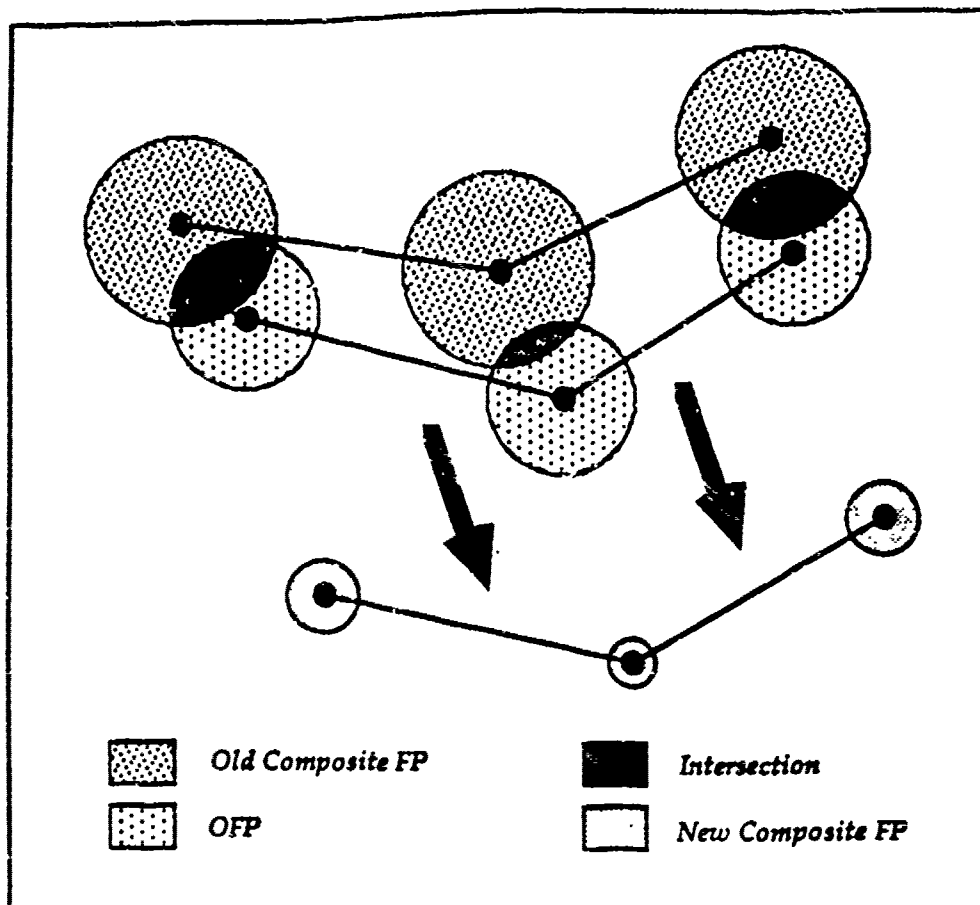


Figure 4.7 Fusing portions of an OFP and a Platform's composite FP to form part of a new composite FP. The points of the new composite are weighted averages of the two corresponding points, and the new radii are calculated from the intersections of the radii of the points.

A fusion result of :NO-MATCH is returned if either of conditions 2) and 3) above is violated. An :UNKNOWN result is returned if neither :MATCH nor :NO-MATCH holds; i.e., the OFP and the composite do not overlap in time for even a single scantime, violating condition 1) above. These three fusion results will be mentioned often in the discussions of the fusion processes, so it is important to understand what each means.

4.1.4.4. The Regular Fusion Process

Figure 4.8 shows the regular process of fusion: the objects involved and the communication paths between them. With reference to this figure, we now turn to the description of the complicated fusion process.

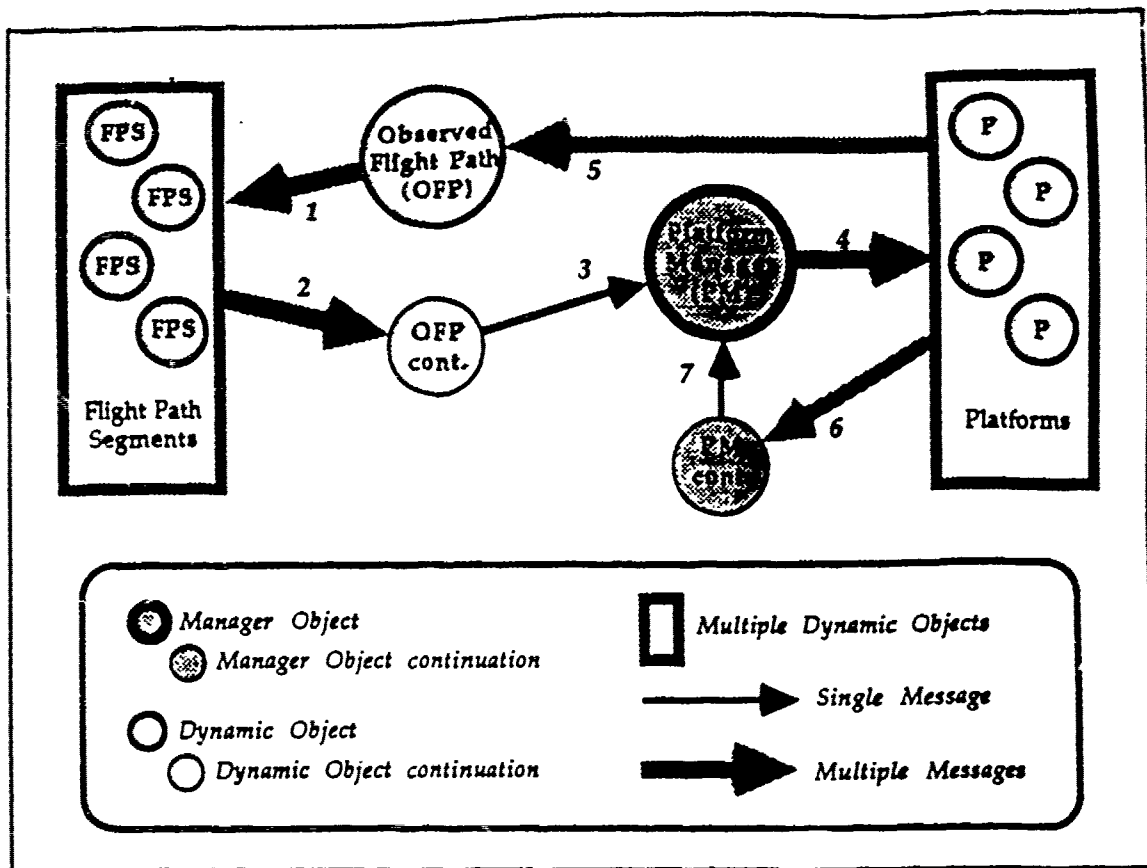


Figure 4.8 The Regular Fusion Process

When an OFP is created or when it is dropped from a Platform (discussed later), it immediately begins the process of fusion:

- 1) Since the OFP does not duplicate FPS data by caching them locally, that information must first be obtained for fusion. The OFP multicasts a *.send-line-estimates* message to all its FPS children [1]. It then spawns a continuation process to collect the replies from the FPSs, giving it also a list of its OFP connections.
- 2) Each FPS responds to the *.send-line-estimates* request from its OFP by returning all its current line estimates to the OFP continuation [2].
- 3) When replies have been received from all of the FPSs, the OFP continuation gathers all of the line estimates together with the OFP's own OFP connections and sends a *.match-OFP-to-platform* request to the appropriate P.M (in charge of aircraft of this type) [3].
- 4) The PM broadcasts a *.fuse-OFP* message to all Ps it has registered, containing the line estimates information provided for the OFP [4]. It then spawns a continuation to collect the fusion replies from the Ps.

- 5) Each P checks to see whether the OFP fuses with its composite-*FP*.¹ If so, the P merges the OFP into its composite and sends an *:add-P-parent* message to the OFP, acknowledging successful fusion [5]. Whatever the fusion result may be, it is relayed to the PM continuation [6].
- 6) When all fusion results are in, the PM continuation examines them and acts accordingly:
 - If there is at least one *:MATCH* result, then fusion was successful and no further action need be taken.
 - If all results were *:NO-MATCH*, then the continuation sends a *:create-Platform* message back to the PM [7], where a new P is created for the unmatched OFP.²
 - If there were no *:MATCH*es but at least one *:UNKNOWN* result, then the fusion search was inconclusive. This is an irregular case treated in Sections 4.1.4.8. and 4.1.4.9.

4.1.4.5. Platform Creation

The creation of a new Platform is requested for an OFP only in the case that the OFP fails to fuse with any existing P during the fusion process (*:NO-MATCH* results received from all). A P creation is handled by the same PM that coordinated the fusion search. The PM learns of the need to create a new P in a *:create-Platform* request from one of the PM continuations spawned to collect the fusion results.³ Before the PM goes ahead and actually creates a new P object, however, it must check one final possibility.

Consider the case of an aircraft that appears in the region and is picked up at the same time by two radar trackers. Assume that the AIRTRAC system is not heavily loaded and that processes are "keeping up," able to handle new data as quickly as they are received. In this case Data Association produces *:create* PORs for the aircraft, one for each radar, at roughly the same data time. Path Association, then, receives these PORs, the appropriate FPMs creates a new FPS for each, and these report to the right FPCs. They, in turn, create new OFPs for the FPSs (assuming no connection possibilities are present), and these OFPs immediately begin the fusion process. The requests for fusion for both radars' flight paths, expectedly reach the responsible PM at nearly the same time. One request must be handled by the PM first, of course. So the fusion search broadcast for the first OFP goes out to all Platforms existing at that time. As soon as the PM

¹ In this discussion an OFP "fuses" with a Platform if the match is consistent with the fusion criteria set forth in Section 4.1.4.3.

² Actually, creation of a new Platform is not this simple. See Section 4.1.4.5. for more details.

³ Platform creation can also be requested by Unfused Flight Path objects (discussed in Section 4.1.4.9.).

continuation is spawned to manage the fusion result collection, the PM turns its attention to the second OFP. Another fusion search is broadcast to the same set of existing Ps, and another continuation is spawned. Soon afterwards, the fusion search for the first OFP ends unsuccessfully, and the PM receives a *:create-Platform* request from the continuation for that search. A new P is created for that OFP and added to the PM's list of existing Ps. Right on the heels of his activity comes a second request for a P creation, this time from the continuation spawned for the second OFP--seems it, too, was unable to fuse successfully with any Ps. The PM then creates another new P for the second OFP, and it is added to the existing P list. And the scenario continues ...

What has happened? Somehow we ended up with two different Platforms for the same aircraft, each housing just a single OFP. Unless one of these Ps is deallocated and its OFP forced to re-enter the fusion process at some later time, there is simply no way that the two OFPs will ever be fused together in the same Platform. This is clearly an undesirable effect--a simple situation made complicated, and NOT because of unreasonable message delays (in fact, the timely message delivery is what *created* the condition).

The problem with this naive Platform creation algorithm is that it neglects the possibility that the PM can be creating new Ps between the time it sets off a fusion search for an OFP and the time it gets back a P-creation request for that OFP--Ps that could and should be checked for fusion with the OFP. We tackled this problem by requiring a PM to check, before it creates a new P, if any other creation has taken place while the search was being performed. If so, it begins the fusion search cycle for the OFP all over again, only this time just involving the recently created Ps. Only when the PM is certain that no intermediate creation has taken place will it proceed with the actual creation of another new Platform.

Unfortunately, this is not the end of our Platform creation difficulties. The proposed method of dealing with false creations will work fine in most instances, but it leaves open the possibility that the fusion-search/creation-request loop will continue indefinitely, especially for complicated scenarios involving a large and steady amount of new track creations. We have overcome this particular bit of nastiness by imposing a *maximum creation attempts* limit for OFPs, another tunable system parameter. If the number of creation requests for a given OFP exceeds this limit, a new Platform will be created no matter what, regardless of any new P activity. The end result of this policy is that, at the risk of incurring some amount of incorrectness (minimal, we hope), we have ensured that all OFPs will eventually be associated with a Platform.

4.1.4.6. Platform Updates

As OFPs receive their own update messages (*:FPS-update*) from their currently *:active* FPS, they immediately feed this track update information along to all their P parents in an *:OFP-update* notification message. The message

contains the latest POR line estimate information for an aircraft's track. When a P receives an update it immediately tries to incorporate the new line estimates into its composite FP. The manner in which this is done is the same as during the original fusion check against the entire OFF; in this case, though, the P need only fuse segments no longer than the POR period. The same fusion criteria are placed upon the new track data. If the P update from an OFF fails to be fused properly because one of its segment points cannot correctly match with the corresponding point in the composite (error radii do not intersect) or because the composite already has data from the same radar source at a scantime covered by the update, then that OFF is considered to have *split* from the P.

4.1.4.7. Platform Splits

What does it mean when an OFF splits from a Platform? In fusing OFF update information by incorporating it into the composite FP, a P makes sure all its OFFs remain consistent with one another. Just because an OFF originally fused successfully with a P doesn't mean it will always remain compatible with the rest of the OFFs in the P; The explanation in the real world is that two or more planes could first appear in the region flying very close to each other--so close, in fact, that their tracks are virtually indistinguishable from one another from the fusion standpoint. The OFFs for those tracks might easily be fused together to form a single P. There is no way to avoid this possibility; we can only expect that the aircraft will eventually move off in different directions, after which an update from one of their OFFs will violate the fusion with composite data from the others, causing a Platform split.

A Platform, like an OFF, does not duplicate track data contained in FPSs; it only maintains its composite FP computed from those data from all its OFFs. When a Platform split occurs, then, it is computationally impossible to "strip off" the single offending OFF from the P, reconstructing the composite to allow the other OFFs to remain. There is no choice but to disband the P entirely, notifying each OFF child, in a *remove-P-parent* message, to disassociate itself with the P. Upon receiving such a notification, the OFF checks to see if it is still fused with any other Ps. If so, then things are fine--in fact, the Platform split has succeeded in eliminating an ambiguous OFF-P relationship. If it does not have another P parent, however, the OFF immediately begins the fusion process anew, just as it did when it was first created. This time, we hope, the greater amount of line estimate information the OFF possesses ought to allow it to unequivocally fuse with the right Platform.

4.1.4.8. Unfused Flight Paths (UFPs)

There are occasions during attempted fusion in which a Platform cannot determine with complete certainty whether an OFF is a logical match with its composite because the two paths do not overlap in time. Most often this is due to message disorder such that the P lags slightly behind the OFF in terms of the

latest data time each has received. Such a case produces an :UNKNOWN result for that P. Often while this is happening, though, the OFP has successfully fused with another P. Since the goal of the fusion process is to unite an OFP with *at least one* P, it becomes unimportant whether or not the :UNKNOWN result is ever resolved for this particular P.

But what happens if no other existing Platforms are able to successfully fuse the OFP, either? The PM continuation, having received all the fusion results (some :UNKNOWN and the rest :NO-MATCH) would notice this fact. The following difficulty arises: The OFP has not yet been associated with a P, something that must be done before the fusion process is complete. But we cannot simply create a new P for the OFP because there's no assurance that the OFP will not be able to fuse at some later time with one of the :UNKNOWN Ps, after more information is obtained. What do we do?

The answer we developed is an object called an *Unfused Flight Path* (UFP), another dynamic object subordinate to a PM. The main function of a UFP is to serve as a higher-level object for an OFP in place of a Platform in the event that the OFP remains unfused following the regular fusion process. It is different from other dynamic objects in the AIRTRAC system in that it is expected to be allocated only for a short period of time, until the OFP can eventually become fused with a P, either by fusing with an existing P or by eliminating all :UNKNOWN cases so that a new P can be created. The motivation behind a UFP is the fact that we expect that after some time both the OFP and the :UNKNOWN Ps will receive additional data through update messages, thus resolving the indeterminate fusion condition during subsequent attempts.

The purpose of a UFP, then, is two-fold.

- 1) It serves as a "placeholder" in the absence of a Platform, a higher-level object to which the OFP can send its update messages. Such messages will contain information about the addition, update, or removal of FPS children and/or OFP connections for the OFP.
- 2) It manages additional fusion processes, so-called *fusion retries*, for the OFP. A fusion retry is attempted whenever the UFP receives new information about the OFP through an update message.

Table 4.4 provides a listing of the various slots of a UFP object, some of which will receive explanation in the following sections.

Table 4.4 Unfused Flight Path

The information in an Unfused Flight Path includes the following:

<i>OFF</i>	name of the Observed Flight Path represented by the UFP
<i>status</i>	status of the associated OFF; keyword, one of (:active :inactive)
<i>radar ID</i>	the identifier of the radar observing the tracks of the OFF
<i>aircraft type</i>	the type of the aircraft
<i>line estimates</i>	cached line estimates of the OFF
<i>creation connections</i>	list of names of earlier OFFs connected to the one the UFP represents
<i>termination connections</i>	list of names of later OFFs connected to the one the UFP represents
<i>failed matches</i>	list of names of Ps whose fusion attempt with the OFF was unsuccessful
<i>fusion retry count</i>	number of (unsuccessful) fusion retries attempted for the OFF
<i>PM</i>	name of Platform Manager in charge of Ps of this aircraft type

4.1.4.9. An Irregular Fusion Process

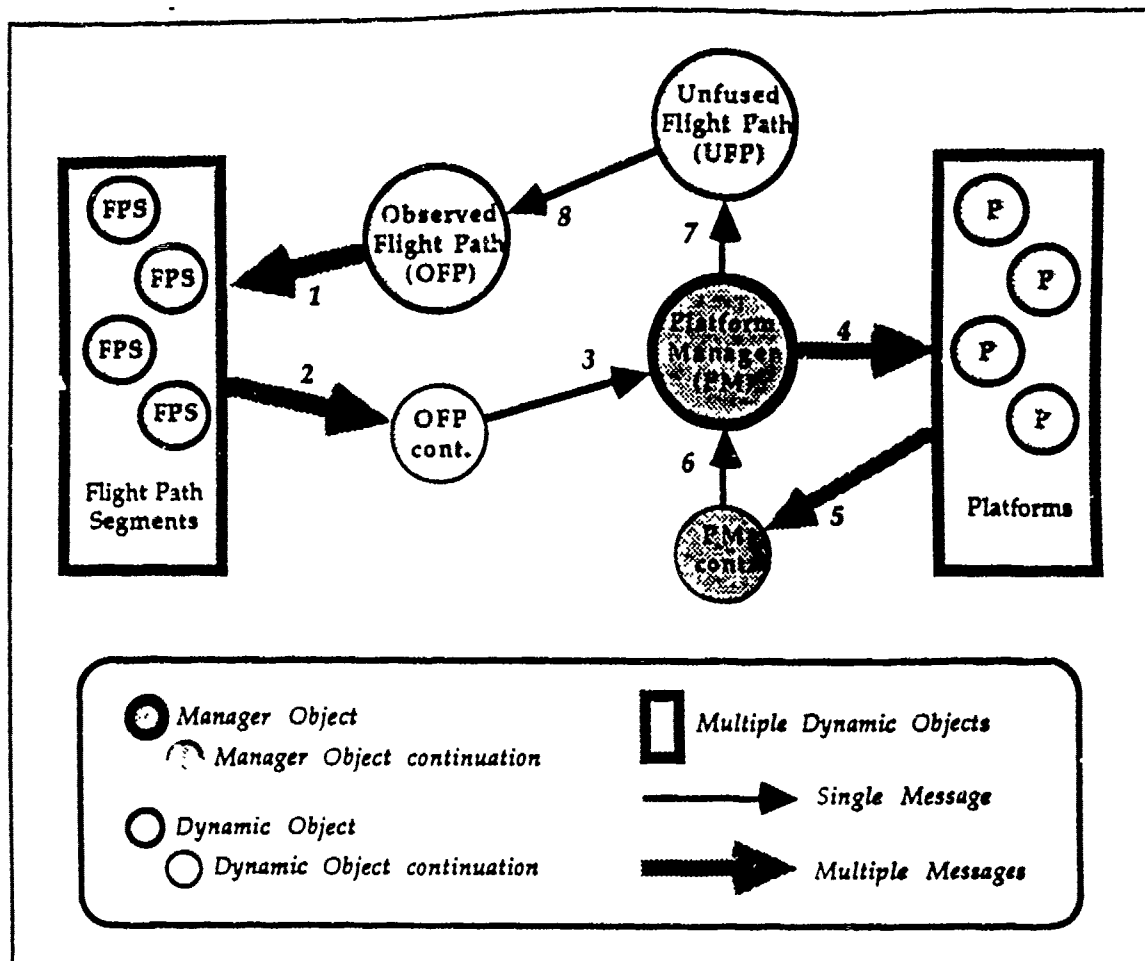


Figure 4.9 The Irregular Fusion Process

Fusion processes are considered *irregular* if they result in the creation of a UFP. The procedure is different from a regular fusion process only in the way the PM continuation handles the fusion results it receives from the Ps. The complete process is as follows, with reference to Figure 4.9, which shows the irregular fusion process diagrammatically. As with the regular process, it begins when an OFP is created or dropped from a Platform.

- 1) through 4) are the same as in the regular fusion process [1,2,3,4].
- 5) The fusion result from each P is sent on to the PM continuation [5]. Since this is an irregular case, this result is never `:MATCH` (no `:add-P-parent` notification message is ever sent to the OFP at this point).
- 6) The PM continuation examines the fusion results when all have arrived and takes appropriate action. We know that in this case there will be no `:MATCH`s but at least one `:UNKNOWN` (if not,

then it is not irregular). The continuation sends a *:create-UFP* message back to the PM including a list of the Ps that returned a *:NO-MATCH* fusion result for the OFP [6].

- 7) Upon reception of the *:create-UFP* request, the PM does just that-- it creates a new UFP and sends it an initialization message that contains the relevant OFP information (line estimates, OFP connections) as well as the list of Ps that failed the original fusion search [7].
- 8) After the UFP is initialized, it sends an *:add-UFP-parent* message to the OFP [8], acknowledging the new association of the OFP with the higher-level UFP.

4.1.4.10. Fusion Retries

The creation of a UFP, again, occurs when the request of fusion from an OFP results in no *:MATCHes* but at least one *:UNKNOWN*, and is made in the belief that the OFP and the *:UNKNOWN* Ps involved will eventually acquire enough update data to "catch up" with each other so that subsequent fusion attempts with those Ps will provide a definitive fusion result. Recall that an OFP treats its relationship with a UFP as if it were part of a true Platform, including forwarding to the UFP all update information it receives from its FPSs. The UFP, by caching all the FPS line estimates of its OFP, is ready at all times to effect another fusion attempt without having to request this track information from the FPSs themselves, as the OFP must do. And indeed this is the major function of a UFP--to manage fusion retries.

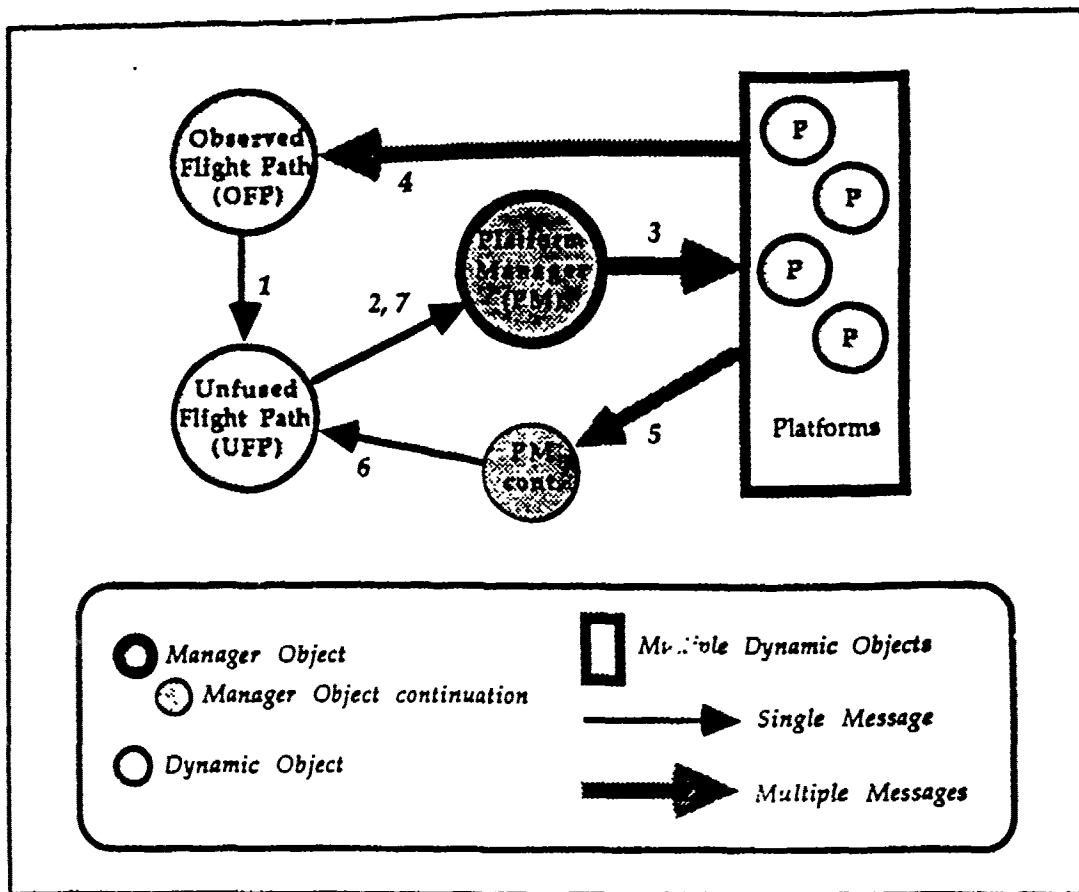


Figure 4.10 Fusion Retries

Figure 4.10 shows a diagram of the fusion retry process. It proceeds as follows:

- 1) Periodically a UFP will receive a new update message from its OFP [1]. When it does, it adds the new FPS line estimates or OFP connections to the previous ones it has stored.
- 2) The UFP then packages up all the line estimates and OFP connections in a *retry-OFP-fusion* message to the PM [2]. Included in this message is the list of Ps with whom the OFP has failed fusion already.
- 3) The PM handles the fusion process much like a regular one, except that it broadcasts the *fuse-OFP* message only to those Ps not included in the list of failed matches provided by the UFP (those that were :UNKNOWN in the previous attempt as well as any new Ps created since the last time, [3]).¹ The PM then spawns a continuation to collect the replies from the Ps.

¹ This is quite an efficient scheme as it limits to only a few the number of Ps needing to perform fusion and report back.

- 4) As before, each of the Ps receiving the *:fuse-OFP* checks to see if the updated OFP now fuses with its composite, returning to the PM continuation the appropriate reply [5]. If fusion is successful, the P also notifies the OFP directly in an *:add-P-parent* message [4].
- 5) The PM continuation, upon receiving the fusion replies, forwards a fusion result directly onto the UFP, not the PM. Depending on what the P replies were, the final result is sent in either a *:successful-fusion-retry*, a *:failed-fusion-retry*, or a *:completely-failed-fusion-retry* message [6].
- 6) Here's how the UFP handles each of the three fusion result messages:
 - *successful-fusion-retry* --this result means that at least one P in this fusion attempt successfully fused the updated OFP, and the UFP is no longer needed. It is simply deallocated.
 - *failed-fusion-retry* --the retry was not successful, but there was still at least one *:UNKNOWN* fusion result from the Ps. The UFP adds the names of newly failed Ps to its list of failed matches and increments its *fusion retry count* in preparation for more retries. If, however, this count has exceeded the *maximum-fusion-retries* (another system parameter), the UFP treats the situation like a *completely-failed-fusion-retry*, as below.¹
 - *completely-failed-fusion-retry* --fusion was impossible with all Ps; therefore, the OFP truly deserves its own Platform. The UFP sends a *:create-Platform* message to the PM with the OFP information it has gathered [7]. It then posts a *:remove-UFP-parent* message with the OFP and is deallocated.

4.1.4.11. Fusion in the Example Scenario

How have the results of the complicated fusion processes manifested themselves in our example scenario? Figure 4.11 on the next page provides a graphical display of Platforms in the system at the familiar instant in time, the final output of Path Association. In contrast to the previous screen images, there is only one area in the display; the data from both radar views has been fused together to form a clear picture of the region under observation. The composite flight paths of individual Platforms are drawn as complete lines, with (again) an identifying string near their origin. (By now the reader has surely become well acquainted with the scenario so that an identification of the beginnings and endings of flight paths is unnecessary!)

¹ The *maximum-fusion-retries* cap was added so that UFPs do not hang around in the system indefinitely. It follows from our requirement that all OFP's eventually be associated with a P.

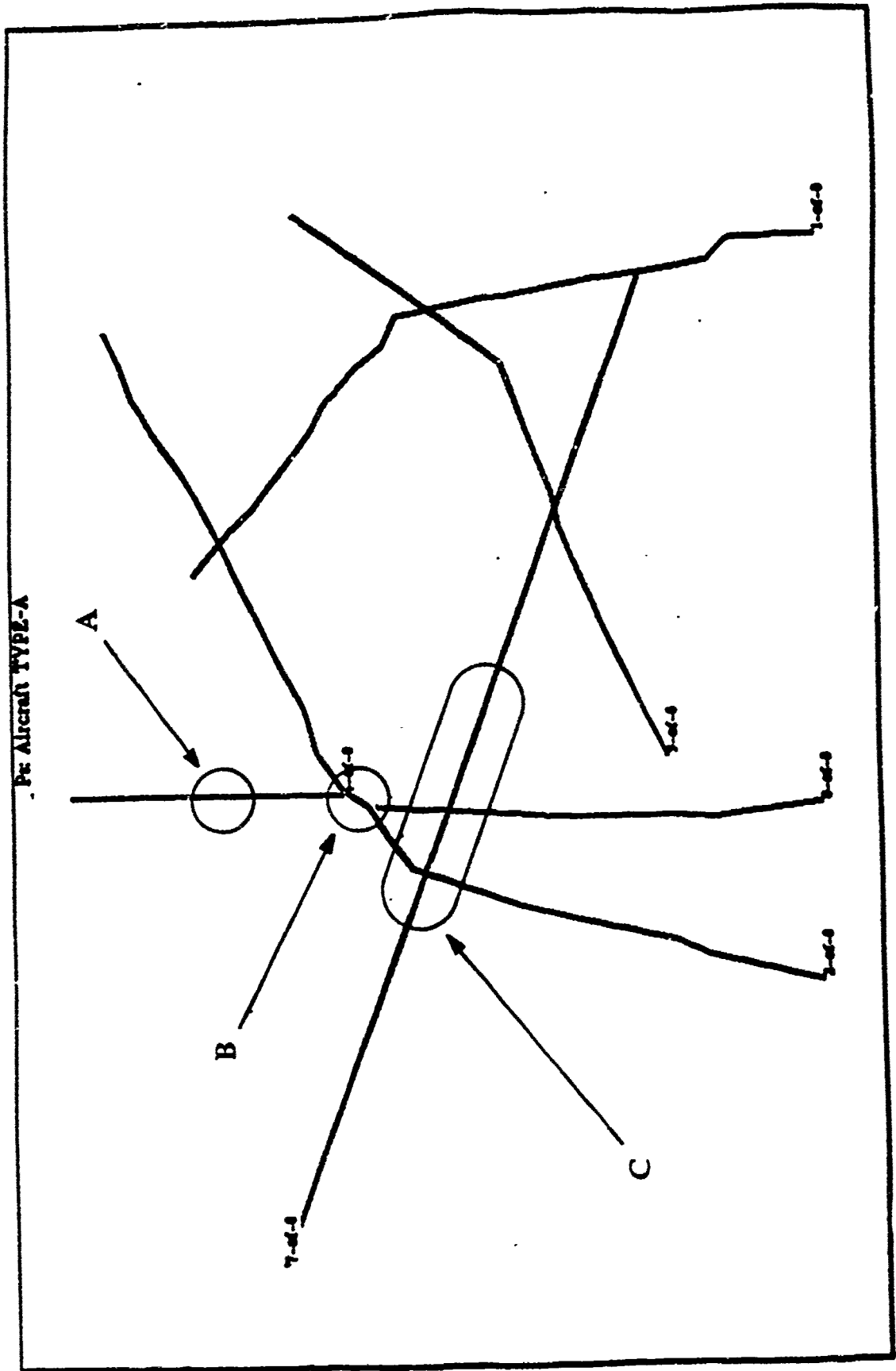


Figure 4.11 Fusion in the Example Scenario

It appears from the display that the fusion process has completely resolved nearly all of the ambiguity that persisted at the OFP level. All but one of the flight paths are well defined from beginning to end, with no hint of the confusion that existed in Figure 4.6 caused by connected OFPs. Let's take a close look at the three areas we have been following (previously viewed from two radar images):

- A) The OFP of radar1 has supplied the data necessary to deal with the missing FPS in the OFP of radar1; together they form a complete, unbroken Platform (labeled 6-of-0).
- B) Here is the only remaining point of ambiguity in the entire scenario. Though all falsely connected OFPs have been resolved, one of the crossing flight paths has remained broken, with a separate Platform covering either side of the cross (0-of-0 and 6-of-0). The explanation of this phenomenon is that both radar1 and radar2 experienced a break in FPSs for that flight path at the crossing point. There was, unfortunately, no radar view that maintained consistent observation of the aircraft across this point, which would have, presumably, been sufficient to resolve the broken path. It so happened that two Platforms were created to fuse the OFPs on both sides of the cross (despite knowledge of the connected OFPs, which apparently was not available at the time of fusion), but since no further information was available to contradict this state of affairs, these two Platforms remained. The result: one platform (2-of-0) correctly spans the crossing in the area, but the two others do not.
- C) Based on the unambiguous OFP of radar2, platform 7-of-0 correctly crosses the entire area. In addition, radar1's view of the area has allowed for the formation of Platform 0-of-0, despite the large gap between OFPs of radar2. The radar shadow of radar2 in this vicinity has been completely overcome.

4.1.4.12. Unresolved ambiguity

From the output of Path Association demonstrated by Figure 4.11, we can see that, for the most part, the Platforms produced by the fusion process very closely model the real-world flight paths of aircraft. However, as exemplified by area B in our example, there may be certain complicated situations in any given scenario which might not be resolved completely by Path Association. Hence, we are left with the possibility that the output from Path Association will still contain ambiguous or incomplete data. It should be pointed out that the existence of such ambiguity is not necessarily the fault of poor processing on the part of Path

Association; rather, it has more to do with the incompleteness of radar data fed to Data Association.

What are we to do, then, about any unresolved ambiguity? This is one way in which Path Interpretation would assist Path Association. Path Interpretation should be able to reason about situations such as the broken Platforms in the crossing area, for instance. By 1) noting that one Platform terminates in the middle of the region, and 2) realizing that the aircraft could not possibly have landed at that point, given the landing speed constraints contained in the aircraft's model, Path Interpretation could deduce that something unusual had occurred. It might then also notice a similar problem with the creation of the other Platform nearby, and, putting all this together, determine that the two Platforms really should be combined into one. Given this feedback from Path Interpretation, Path Association could then correctly adjust the collection of Platforms and proceed as usual.

4.2. System Architecture

Figure 4.12 presents a diagram of the entire system architecture of Path Association. Pictured are all of the various objects (static and dynamic) of the system along with the paths of message-passing communication between them. Though this diagram may give the system a suspiciously data-flowish look, one mustn't be fooled--a lot has been omitted from the figure for clarity's sake. One thing we must remember as we view this diagram is that each object pictured represents a class of objects, not just a single entity. In addition, we must keep in mind that the system is implemented on a multiprocessor architecture of many distributed processor-memory pairs. Hence, all the objects are executing their methods concurrently, leading to a highly complex network of inter-object relationships and communication.

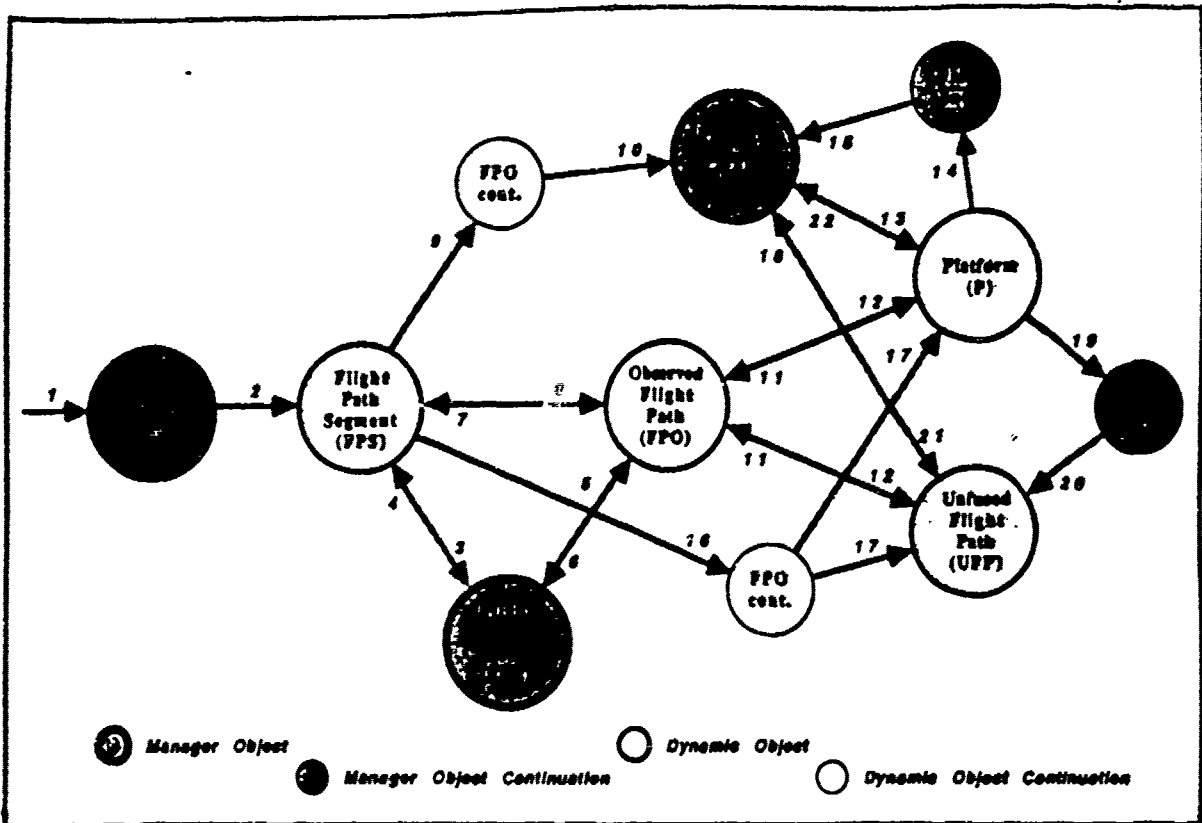


Figure 4.12 Path Association System Architecture

An important high-level hypothesis we are investigating with this type of system software architecture is that the need for synchronization increases (opportunities for concurrency decrease) with the complexity of reasoning. Our experimental evidence supports this claim. While the Data Association experiments demonstrated close to linear speedup over the range of processors, the task it was performing was quite simple in relation to the functions of Path Association. Naturally, the speedup results we have obtained in experiments with Path Association (reported later), though encouraging, are not quite as extraordinary as those of Data Association. Our belief is that the potential for concurrency in Path Association is decreased by the great amount of coordination and synchronization inherent in the connection and fusion stages.

4.3. Design Philosophy

In the course of developing the Path Association module of AIRTRAC we have had to deal with the implications of building a complex application on a distributed, message-passing system that does not guarantee message ordering. Consideration for these issues has necessarily led us to adopt a design philosophy that we believe is especially appropriate for efficient programming of AIRTRAC-like concurrent software systems. Our design principles pervade the entire

implementation of Path Association. Here we list two of the most important, along with a few instances of their realization in Path Association:

1) *Do the best you can with what you have.* Phrased another way: *Don't wait.* An overriding concern in Data Association was making sure data were in order before making assertions about them. Elaborate mechanisms were included to force sequentially of messages at each stage of the design. Recognizing that such an approach potentially sacrifices performance to some degree, we attempted to avoid, whenever possible, requiring the complete ordering of data, even if it meant processing incomplete information. It is important to note that we adhered to this principle only when doing so did not compromise the correctness of our solution. Some examples:

- When an FPM receives a POR with a new track ID, it immediately creates a new FPS for that track, regardless of whether or not that POR's status is *:create*.
- An FPC connects in both directions of time and always associates an FPS with an OFP as soon as possible. In the usual connection case (at creation of an FPS) an FPC searches backward in time to find terminated FPSs that might connect, not worrying about whether all reported FPSs are up-to-date. It deals with the unusual connection case (at termination) by searching forward in time for possible connections to FPSs already created.

Both of these examples show our intention that a object be created as soon as any evidence exists to support such creation.

2) *Treat exceptions as exceptions.* In other words, program for the nominal case. Do not build expensive exception-handling mechanisms into the mainstream processing as this will degrade performance. Rather, include low-cost *exception-recognition* mechanisms that will, if necessary, trigger (admittedly more expensive) operations to resolve the situation. This joins nicely with our first principle by dealing with any condition as easily and as quickly as possible. We notice its evidence in Path Association:

- The connection criteria are intentionally simple and unambiguous (no "degree" of connection probability computed) because the temporal and spatial constraints involved are usually sufficient to handle all cases except an occasional crossing pattern.
- We use a simple fusion algorithm that assumes, basically, that if two paths are near to each other then they must belong in the same Platform. A poor fusion can be amended by the detection of a Platform split. In addition, a UFP is commonly created only in cases of extreme disorder.

The motivation for our design philosophy stems from our experience that, 1) in spite of the obvious potential for disorder in the system, things usually keep up in most cases, and 2) exceptions in our domain (paths crossing or aircraft flying close together, etc.) are rare indeed. It is beneficial, therefore, to program in a style that is not too conservative in order to achieve maximum performance.

4.4. Important Features

The Path Association portion of the AIRTRAC system, which we have described in this chapter in some detail, includes several features worth highlighting:

Parallel search coordination. This is perhaps the most difficult feature of any concurrent application with dependencies among many distributed objects. Path Association has two instances of parallel search: in the connection process at each FPC and during the fusion process managed by the PM. The troubles inherent in coordinating concurrent search in both procedures should be apparent from the detailed description of each process.

Retraction of assertions (nonmonotonicity). The unpredictability of message order in the system and our design principle to go ahead and reason with available data can sometimes lead to false inferences during connection and fusion. Path Association must be able to robustly manage the retraction of these assertions. For example:

- When an FPC notices that a connection between two FPSs calls for a change in connection conditions previously imposed, it must notify and/or create OFPs as needed to establish the correct FPS-OFP relationships.
- A Platform is capable of recognizing when an OFP has been falsely fused into its composite, and takes the steps necessary to repair the incorrect association (a Platform split).

Consistency maintenance of cache data. Path Association must provide appropriate measures to deal with the possibility that high-level objects in the system may not have complete information about the events of their lower-level constituents. Some examples:

- Platforms must be kept up with the latest POR information through forwarded messages from its OFPs so that it can properly execute fusion of other OFP data.
- Before an OFP can report to a PM for fusion, it must collect the latest line estimate information from all of its FPSs so that the Ps can attempt fusion with the most current track data. (Strictly speaking, this is not a matter of the consistency of cached OFP line estimates since an OFP does not store this information, but in the

sense that the OFP provides updated FPS data for Platform use, the principle still applies.)

- Whenever an OFP receives notification (in an *:add-P-parent* or *:add-UFP-parent* message) that it has been associated with a P or a UFP, it forwards its entire store of current FPS line estimate and OFP connections to the higher-level object. The P or UFP then filters out all the information it already has, keeping only the data received by the OFP since the initial fusion request (during which time it had no parent P or UFP to which to report updates). This also eliminates the complication of unknowingly sending update messages to recently deallocated objects before notification of such an event is received by the OFP. (Note that the collection of FPS line estimates is itself subject to cache consistency problems, as mentioned above.)

As a result, Path Association does not simply employ forward-directed, data-driven reasoning. Instead, it uses several feedback paths of communication that add greatly to the complexity of the entire system.

What is new and exciting about all of this? After all, many knowledge-based systems purport to include some or all of the above features. The important consideration to keep in mind is that Path Association must deal with all of these problems in an asynchronous, distributed environment of dynamic entities, subject to the possible indeterminacies caused by message disorder. In this context the design and implementation of Path Association represent a significant challenge.

4.5. Outstanding Issues

Having described the complicated software architecture of Path Association and the design philosophy which dominated our work, we are left with the following important questions:

- What techniques and constructs are useful for development of application software for multiprocessors?
- Can we achieve major speedup of large, symbolic (knowledge-based) applications through concurrency?

Each of these matters will be addressed in turn in the next two chapters.

5. ELMA

This chapter describes work which addresses the first of the issues mentioned in Section 4.5, namely, what techniques and constructs are useful for development of application software for multiprocessors? We discuss programming constructs developed in the course of developing Path Association. The result of this work has come to be referred to as ELMA, an acronym for Extended Lamina for Memory-management Applications.

ELMA consists of additional syntax and constructs for programming the CARE architecture in the object-oriented style at a higher level than LAMINA. From the outset of this project a strong effort has been made to separate application-dependent code from anything which could be generalized. Much of the latter has worked its way into ELMA in the belief that it could be useful to other applications. On the other hand, the task of writing Path Association has been considerably simplified by the presence of these specialized, high-level constructs.

ELMA actually provides a complete programming interface insofar as whole applications are realizable without recourse to LAMINA. In fact, no knowledge of LAMINA is required. We believe the ELMA Programming Interface is considerably more transparent to the programmer than "raw" LAMINA, albeit at some loss of generality.

ELMA provides the CARE programmer with the following:

- Syntax and constructs for managing concurrency and memory usage.
- A library of definitions of special-purpose LAMINA objects, for the programmer to further specialize ("mix in") if required.
- A library of useful abstract data types.

These features are described in the following sections. Appendix 4 contains an annotated example ELMA program. The reader is referred to the ELMA Programmers Guide [Noble 88] for further details.

5.1. Syntax and Constructs

Experience from earlier applications developed on CARE, such as ELINT and Data Association, strongly supports the need for strict control of concurrent object creation. Both these applications, however, present rather ad hoc approaches to implementing such control. ELMA provides the programmer with high-level constructs which remove the burden of writing such control code without unduly sacrificing performance.

Unlike these earlier applications, AIRTRAC involves extensive allocation and deallocation of objects. Both *Observed Flight Path* and *Platform* objects are destroyed when evidence for the existence of their corresponding real-world counterparts disappears. For example, a platform believed to correspond to one aircraft will be deallocated when evidence appears to the contrary; i.e., at least two distinct aircraft tracks are subsequently observed. An important feature of ELMA, therefore, is the facility it provides for managing the storage of dynamic objects at the level of the application.¹ This section describes these and other constructs.

5.1.1. Memory management

ELMA memory management is based on *free pools*. The notion of a free pool is easy to understand in a sequential program. The program maintains a list of (pointers to) commonly used objects (or records) which are allocated in advance. New objects are taken from the pool rather than being allocated from heap storage directly, and returned to pool when deallocated. Apart from syntactic nicety, free pools offer an elegant and simple memory interface that puts the application in control of its own memory management (of pool objects). New storage is allocated by simply returning the address of an empty, already allocated object in the pool.²

In ELMA, a free pool is an object in the local address space of a *manager* object which contains the remote addresses of objects of some type (all the same type), the latter spread out over a predefined pool of sites in the multiprocessor.³ A manager may own multiple free pools, as in the case of Path Association Platform Manager objects, which have free pools of both *Platform* and *Unfused Flight Path* objects. Objects allocated from a free pool of a manager are said to be *subordinate* objects. As in the sequential case, a new free pool object is allocated by returning its (remote) address. However since objects in the pool are separate processes (typically on remote sites), the storage associated with a newly allocated object can only be (re-)initialized by sending a message to that object. This slot initialization message is automatically prepended to any other messages sent to the object, or sent separately if there are no such messages.

¹ Note that processor memory is not simulated explicitly in CARE; memory and memory management are both inherited from the host LISP machine. This makes the simulator inherently non-deterministic since consing can take an indeterminate amount of time. Simulation measurements also derive some randomness from the non-deterministic network routing employed which depends on CARE load conditions.

² Empty free pools can of course be made to grow in size to accommodate requests for new objects.

³ We use the term *site* rather than processor to refer to one CARE processing element, actually two processors sharing local memory.

As for sequential free pools, a new object is created only when the pool is empty. Unlike sequential free pools, in which new storage is usually allocated within local address space and always allocated on the same processor (since there is only one processor!), a new ELMA object can be allocated on any site of the multiprocessor. The programmer specifies a list of permissible site numbers¹ for placement of subordinates of each manager. A function can also be specified which takes this list as its first argument and returns the CARE site for a new subordinate. If the latter is omitted ELMA chooses a site randomly² from the specified list.

Similarly, object deallocation involves sending a *:deallocate-self* message to the subordinate object as well as updating the state of the manager object's free pool object. Furthermore, since the storage for one object can of course be allocated and re-allocated any number of times, there is a need to distinguish between different "incarnations" of a dynamic object to avoid proper handling of messages. Message disordering makes it possible for messages sent to a object to arrive *after* it has been deallocated, i.e., after it has received the *deallocate-self* message. In fact, if the disordering is extremely bad, it is even possible for an incarnation of a subordinate object to receive a message intended for a previous incarnation, since LAMINA messages are sent to a given remote address and this does not change between incarnations.

This problem is solved by taking the obvious approach of associating a *name*³ with each free pool object; every object is renamed each time it is "reincarnated," i.e., re-allocated from the free pool. The ELMA programmer can specify how deallocated objects are to handle messages which are received out of order, on either a object type or object instance basis. Possible actions include:

- always execute
- return to sender
- forward to another object (the forwarding address can be determined dynamically if desired)
- drop and ignore (default)
- signal an error (useful for debugging)

¹ A CARE *site number* is an integer between zero and the size of the CARE grid minus one and is used to reference an array of CARE *sites* which are simulator components.

² By default ELMA employs a random load balancing scheme for dynamic (subordinate) objects. This is reasonable in view of the fact that there is no way of knowing *a priori* whether any given dynamic object will be busier than another. In fact empirical evidence suggests that in the absence of such load knowledge, random allocation is optimal. Note that static (manager) objects are allocated on sites determined by the programmer.

³ In fact, all ELMA objects, including managers, have a unique name. These are simple strings in the present implementation.

ELMA replaces the LAMINA message sending construct with one which takes a destination *name* rather than remote address.¹ The sender is therefore able to specify which particular incarnation of a object the message is intended for.

One final but important thing to note about the free pool mechanism described here is that it competes with application-level methods for the computational resources of the multiprocessor. Unless system resources are available on other sites for subordinate object allocation *and* on the site of the manager object for handling various handshake messages, no performance gain can be expected. The free pool mechanism effectively trades off available resources against object creation latency; providing the resources are available, object creation latency is reduced.²

5.1.2. Smart message sending

Nakano [p. 36, Nakano 87] writes the following comment about the Data Association module.

"A significant part of the [Data Association] LAMINA concurrent program implements techniques to allow a LAMINA object receiving messages from a single sender to handle them as if they were received in the order in which they were sent, without gaps in the message sequence. By doing this we incur a performance cost because the receiver waits for the arrival of the next appropriate message, rather than immediately handling whatever has been received."

Unfortunately such code can be quite convoluted; the presence of additional code for enforcing order detracts enormously from program clarity. The mapping from concept to application code is less apparent and the program is correspondingly more difficult to understand, maintain and debug. Nakano introduced several mechanisms to simplify the message ordering task, which reflects the Data Association philosophy of "let's make sure everything is in order before we attempt this action."

As mentioned earlier, the approach taken in Path Association is fundamentally different.

- First, for reasons described earlier, we have programmed to avoid message waiting whenever possible.

¹ LAMINA *sending* is ELMA *mailing*.

² Reduced because the time for object creation is replaced by the time to send a message and receive an acknowledgement. Since a message is invariably sent to a new object anyway, it is usually a matter of just prepending the additional information necessary to reinitialize the object, rather than sending an additional message. The net cost of this is only slightly more than the cost of sending the original message alone so the "creation" is very cheap.

- Second, when order is essential it is enforced automatically and is transparent to the programmer wherever possible. Needless to say, there are still times when message order is of paramount importance. For example, although the outcome of initiating creation of an object and then sending a message to that object is ill-defined, the programmer's intent is clear enough; have the object handle the message once it is created. ELMA constructs reflect our conviction that the programmer should not have to worry about details such as this. Indeed in this situation ELMA automatically caches the message (and any others) for subsequent re-sending upon receipt of an acknowledgement from the newly created object (cf. example program in Appendix 6).
- Third, general-purpose mechanisms exist for those times when sequentiality is desirable.

5.1.2.1. Automatic deferral

To implement the name-based sending required by the free-pool mechanism, each ELMA manager maintains a *name table*, which translates names to remote addresses. This table is also used to store data about the various objects known by the manager. Managers use this data to automatically defer messages to subordinates still being created. This is transparent to the programmer who can send messages to an object given its name without concern for the status of the object.

5.1.2.2. Combining multiple messages

Sometimes it is convenient to group messages together to guarantee sequential execution of their methods. Of course it is always possible to define a single method which includes the messages in the group but this defeats the purpose of structured programming. ELMA provides the *mailing-together* construct for the desired functionality.

5.1.3. Application-level meters

Although the simulator provides sophisticated hardware-level instrumentation for CARE [Delagi 87a], LAMINA provides nothing similar for the application level, leaving this entirely in the hands of the programmer. ELMA, however, provides numerous meters for recording application-level data which are useful for both debugging and performance analysis. This is a major reason why programming in ELMA is easier than in LAMINA. Methods and functions are included for:

- recording received messages

- recording sent messages
- timing LAMINA methods ("triggers")
- timing LISP methods and functions
- measuring message queue lengths
- measuring dynamic object recycling, i.e., free pool utilization
- recording user-specified events
- measuring user-specified data

These are described in detail in the ELMA Programmers Guide [Noble 88].

5.1.4. Other Constructs

ELMA also includes a variety of utility functions as well as some constructs which are refinements of those found in LAMINA. These are described in detail in the ELMA Programmers Guide [Noble 88].

5.2. Specialized objects

ELMA includes a small library of specialized object types and mixins to facilitate program development through code reusability. An ELMA application is implemented exclusively in terms of managers, subordinates and allocators. These object types are described in this section.

5.2.1. Manager

Managers are objects which are allocated statically¹, i.e., at initialization time, and are typically responsible for tasks involving many (subordinate) objects such as distribution, search, and object creation. Each manager can maintain zero or more free pools of subordinates. The number of managers required depends on the particular application and its input data and must be determined a priori. For example, Path Association has four types of managers in the following numbers.

¹ Managers can also be "ephemeral," that is, created dynamically to perform a specific task. At the present time, however, ELMA does not support non-static managers.

Table 5.1 Path Association Mangers

<i>DAS</i>	one	simulates Data Association by reading PORs from scenario file
<i>FPM</i>	one per aircraft type/radar	distributes PORs to FPSs
<i>FPC</i>	one per aircraft type/radar	connects FPSs into OFPs
<i>PM</i>	one per aircraft type	fuses OFPs into Ps

5.2.2. Subordinate

Subordinates are objects which are created dynamically, i.e., at run time, by managers and typically contain the state of the system. Subordinates can be *allocated* (created), *deallocated*, and *reallocated* many times over in the course of program execution.

5.2.3. Allocator

Allocators are objects which create manager objects at initialization time and start applications. There is only need for one allocator object per ELMA application. The (*start-ELMA*) call creates the allocator and initiates the initialization process, in accordance with initialization data supplied by the application. This initialization data includes the following:

- allocator name
- allocator type
- allocator site number
- start message
- start message recipient (name)
- manager initialization data

The allocator is created on the specified site and given the specified name. Thereafter, the managers specified by the manager initialization data are initialized. When all newly created managers have acknowledged their creation to the allocator, the allocator initiates execution by sending the start message to the start message recipient.

5.3. Abstract data types

ELMA also includes a library of abstract data types (ADTs) such as the following:

- *Table* ADT which can be arbitrarily large, of arbitrary dimensions and indexable on arbitrarily defined keys
- *Ring buffer* ADT for efficient maintenance of time-dependent data, i.e., the addition of new data and timely removal of old, "out of date" data.

Most of these ADTs are so general that they would be a useful addition to any software library. They are included in ELMA simply for the convenience of the programmer. Refer to the ELMA Programmers Guide [Noble 88] for more details.

6. Performance

This chapter describes work which addresses the second of the issues advanced in Section 4.5, namely, can we achieve significant speedup of large symbolic applications through concurrency? In particular, we discuss the design and execution of a series of experiments to determine the performance of Path Association. We show that Path Association achieves an $S_{64,4}$ speedup of 12. This is contrasted against Data Association's linear speedup over an even broader range. We attempt to reconcile our understanding of Path Association with these earlier results and the experimental results discussed herein.

Common terms appearing in this chapter are defined below.

- *Speedup:*

Speedup $S_{n,m}$ is defined as the ratio T_n/T_m , where T_k characterizes the execution speed of a given task on a k-site multiprocessor¹. T_n and T_m represent the same program running on n-site and m-site multiprocessors respectively.

- *Scenario:*

Simulated input data (PORs for Path Association; RTRs for Data Association). The number of simultaneous aircraft in the scenario dictates the amount of data parallelism and thus the load on the system.

- *Exception:*

A scenario situation involving one or more tracks which does not conform to regular behavior, for example, the splitting into two tracks of observations which had previously been detected as only one track.

- *Data rate:*

The rate at which scenario data is actually put into the system, varied in order to load the system and typically faster than that specified by the scenario. A very slow data rate allows the system time enough between inputs to return to quiescence; a very fast data rate can overload the system².

¹ T_k is greater if execution is faster. This subsumes Nakano's definition [Nakano 87], in which the inverse ratio T_m/T_n is the ratio of task execution times, since the latter obviously characterizes task execution speed.

² System overload is characterized by queues of unbounded size.

6.1. Experiment Design

There are a tremendous number of parameters affecting Path Association. These include the number of CARE sites (*grid size*), ELMA free pool sizes (initial size and threshold size for replacement), input scenario (size and exceptions), input data rate, and application-specific parameters such as connection search interval, connection ring buffer length, fusion retry period, maximum creation attempts (of new platforms). The number of observables is even greater: processor utilization, update and creation latencies, message frequency, method and function execution times, free pool utilization, queue lengths and service times, correctness of output, etc.

For our initial experiments we chose to focus on three parameters, namely CARE grid size, input scenario, and input data rate¹. Specifically, we chose to explore the following hypotheses:

- For a given input scenario, performance improves directly with grid size.
- For a given grid size, performance degrades with increasing input scenario exceptions, slightly at first and then markedly. Exceptions require invocation of relatively expensive exception-handling code. Thus while small numbers of exceptions can be tolerated with only slight loss of performance, large numbers usurp resources to the extent that performance is severely degraded.

Figure 6.1 shows a profile of the base input scenario we generated to test AIRTRAC, the so-called CONTROL-10 scenario. There are three dimensions to any scenario: length, width and exceptions. The scenario must be long enough to enable the system to settle down into steady-state behavior. It must be wide enough, i.e., contain enough simultaneously observed aircraft as manifested in PORs per cycle, to provide sufficient data parallelism and thus opportunities for parallel computation. Finally, it needs exceptions to invoke exception-handling code which causes assertion retraction or computational "unwinding."

¹ Results of other experiments in progress will appear in a forthcoming paper.

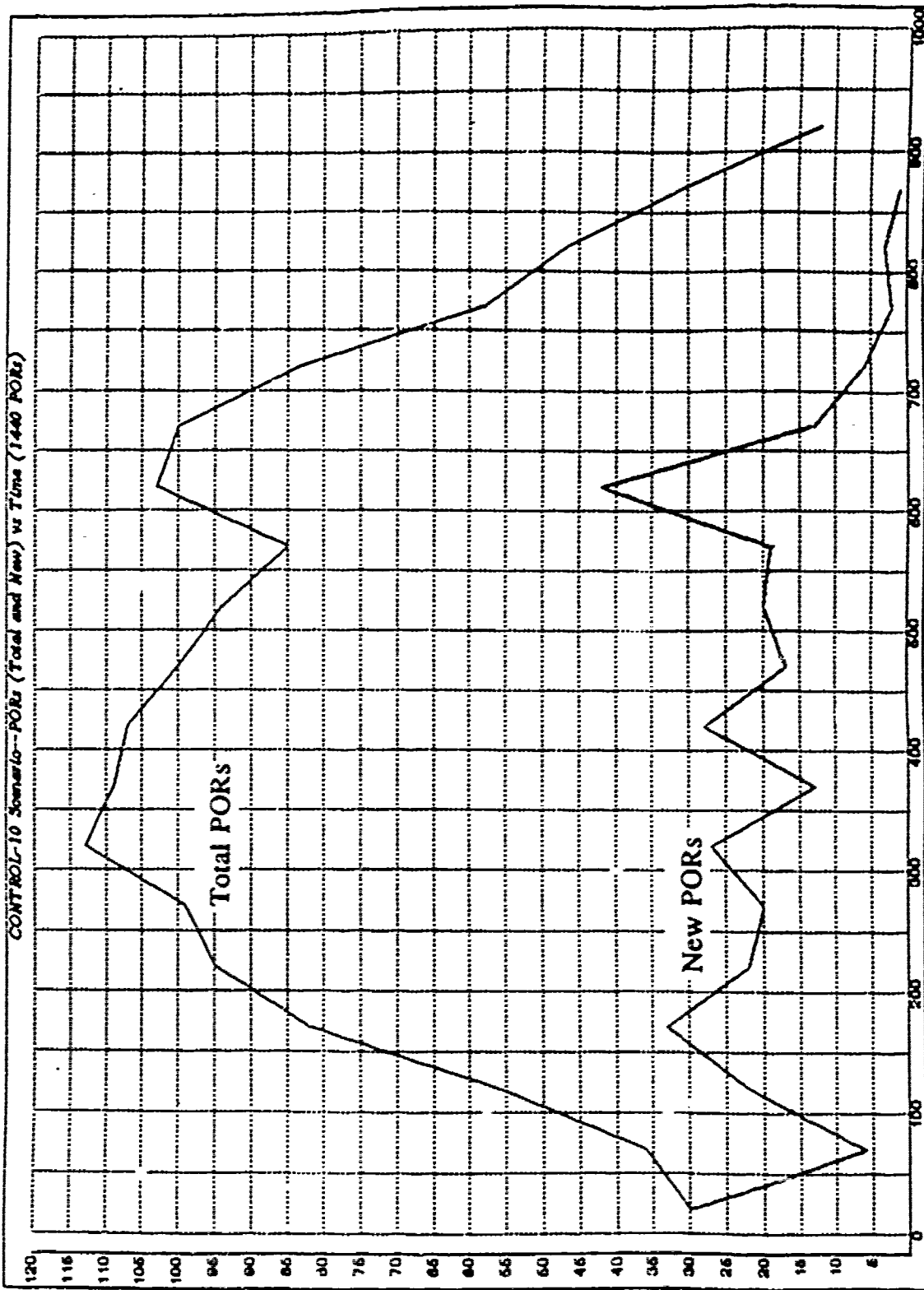


Figure 6.1 Profile of CONTROL-10 scenario

The following characteristics should be noted:

- The scenario contains three types of aircraft which are observed by three different radar sources.
- Scenario duration is 900 seconds, which constitutes 18 POR periods of 50 seconds each¹. Each POR consists of 5 scans of data, i.e., the scan period is 10 seconds.
- After an initial buildup, approximately 100 PORs (never less than 85 and never more than 113) appear each POR cycle²
- Approximately 20 new PORs³ appear each POR cycle
- The number of new PORs surges to twice the average just before the end of the scenario (and although this "bump" is not unrealistic of real-world data, it does complicate measurements)

Two experiments were designed and performed: the *Basic Speedup Experiment* tested the former hypothesis and the *Exceptions Experiment* the latter.

- *Basic Speedup Experiment:*

We ran AIRTRAC for CONTROL-10 on six different grid sizes: 4, 8, 16, 32, 64 and 128.

- *Exceptions Experiment:*

We generated two additional scenarios, CONTROL-10X and CONTROL-10XX, almost identical to CONTROL-10 but containing approximately 10% and 30% exceptions respectively. We then ran AIRTRAC for both of these scenarios on the three largest grid sizes: 32, 64 and 128.

The following section addresses the question of how exactly we measured system performance, i.e., what quantity, T_k , we chose to characterize execution speed.

¹ For trend analysis purposes a longer scenario would have been desirable but simulator garbage collection problems limited the size. One AIRTRAC run for the CONTROL-10 scenario ran for 2 to 3 hours on a TI Explorer II with 16MB of physical memory and 100MB of virtual memory.

² Trial runs indicated that 100 PORs/cycle was adequate for our purposes. This corresponds to approximately 33 simultaneously observed aircraft, since one aircraft is picked up by three different radar sites.

³ A "new" POR corresponds to a previously undetected aircraft.

6.2. Quantitative Performance

6.2.1. Criteria

Performance evaluation of parallel systems with continuous signal data input is a topic which justifiably warrants a separate paper. In the context of AIRTRAC, however, the bottom line is execution speed; the goal is to achieve *speedup* with increasing multiprocessor *grid size*, and preferably linear speedup at that. The approach taken with Data Association was to fix the input data rate (scans of data per second) at some "reasonable" value determined a priori and measure key latencies at that data rate¹. This is not by itself a reliable measure, however, since a given data rate can affect different grid sizes in qualitatively different ways. For example, a data rate sufficient to keep a large grid busy is likely to stress the smallest grids to the extent they will simply be emptying backlogged input queues for the duration of the scenario and beyond. (This behavior is evident in the latency graph for the highest data rate in Appendix 1). This is undesirable since it is the steady-state behavior of the system which is of interest².

Subsequent work refining the ELINT application has led to the notion of *sustainable data rate*, which is defined as the maximum data rate for which designated latencies do not increase over time. Speedup is determined by plotting sustainable data rate versus grid size rather than latency. The choice of latency is quite important, since latencies which reflect sporadic, irregular activity are not representative of the steady-state behavior of the system. Care is also required in selecting latencies so that they indicate the true performance of the whole system, not just a portion or subsystem. For AIRTRAC we monitored five latencies and used three of these--FPS update latency, Platform update latency, and initial fusion latency--to determine the sustainable data rate.

- *FPS update latency:*

Time between a POR for an existing FPS entering the system and being incorporated into that FPS.

- *Platform update latency:*

Time between a POR for an existing Platform entering the system and being incorporated into that Platform.

- *Initial fusion latency:*

¹ Latency is defined as the duration between the time when the system receives a datum (a POR in the case of Path Association) and the time when it actually uses that datum to assert some fact (for example, incorporates the POR data into an FPS).

² The underlying problem is really how to determine steady-state behavior from simulated runs of relatively short duration.

Time between a POR for a new Platform entering the system and the result of the first fusion attempt, i.e., match or otherwise, being made known to the appropriate Platform Manager.

Note that only the initial fusion latency truly reflects system throughput as a whole, since unlike FPS and P updates, only the initial fusion process requires a chain of messages between objects of all types:

FPM->FPS->FPC->FPO->PM->P

Nevertheless, the other two latencies are of interest since they characterize the more common and certainly more regular update tasks. A well-tuned Path Association would ideally sustain approximately the same data rates for all three types of latencies. A large disparity would indicate that part of the system is sustaining the load while another part (or the system as a whole) is not, thus suggesting load imbalance of some sort.

Unfortunately the relatively short length of the input scenario (necessitated by simulator limitations) combined with irregularities in the scenario itself made latency trend analysis extremely difficult (cf. Appendix 1 for example latency graphs). The straightforward linear regression analysis program for determining the trend over time of a latency which had sufficed for ELINT failed for AIRTRAC primarily because of insufficient data. In addition, the surge in new PORs occurring towards the end of the input scenario often caused a misleading rise in initial fusion latencies which further skewed trend analysis results.

Sustainable data rate thus required a new definition for Path Association.

- *Sustainable data rate:*

Sustainable data rate $SDR_{a,b}$ is defined as the input data rate for which absolute latencies are below a threshold of a at least b percent of the time.

This definition meets the objectivity requirement that the sustainable data rate be program determinable, and has the following additional advantages:

- The definition allows for some exceptional latencies in the course of a run providing the system recovers, i.e., the average latency stays low. This permits excessive latencies resulting from short-lived surges in system load, such as those due to the spike in new PORs in the input scenario, to be treated more fairly.
- This definition is also a meaningful and adequate engineering specification for performance.

The following two sections describe experimental results obtained using a criterion of $SDR_{\beta, 0.9}$, for β in Table 6.1. In particular, we measured *sustainable*

POR frequency, defined as the number of PORs per second. For example, a sustainable POR frequency of 10 Hz corresponds to a POR period of 100ms which in turn corresponds to a scan period of 20ms¹.

Table 6.1 Sustainable Data Rate Latency Thresholds (B)

<i>FPS Update</i>	40ms
<i>Initial Fusion</i>	500ms
<i>P latform Update</i>	60ms

6.2.2. Experimental results

6.2.2.1. Basic speedup experiment

Figure 6.1 below shows the results of the basic speedup experiment.

¹ Given a sustainable POR frequency of 10Hz and 100 PORs per cycle in the input scenario, 1000 PORs enter the system each (simulated) second. At this rate the scenario's entire 1500 PORs are pumped into the system in a mere 1.5 seconds. (Due to buildup and builddown, however, the actual time is longer).

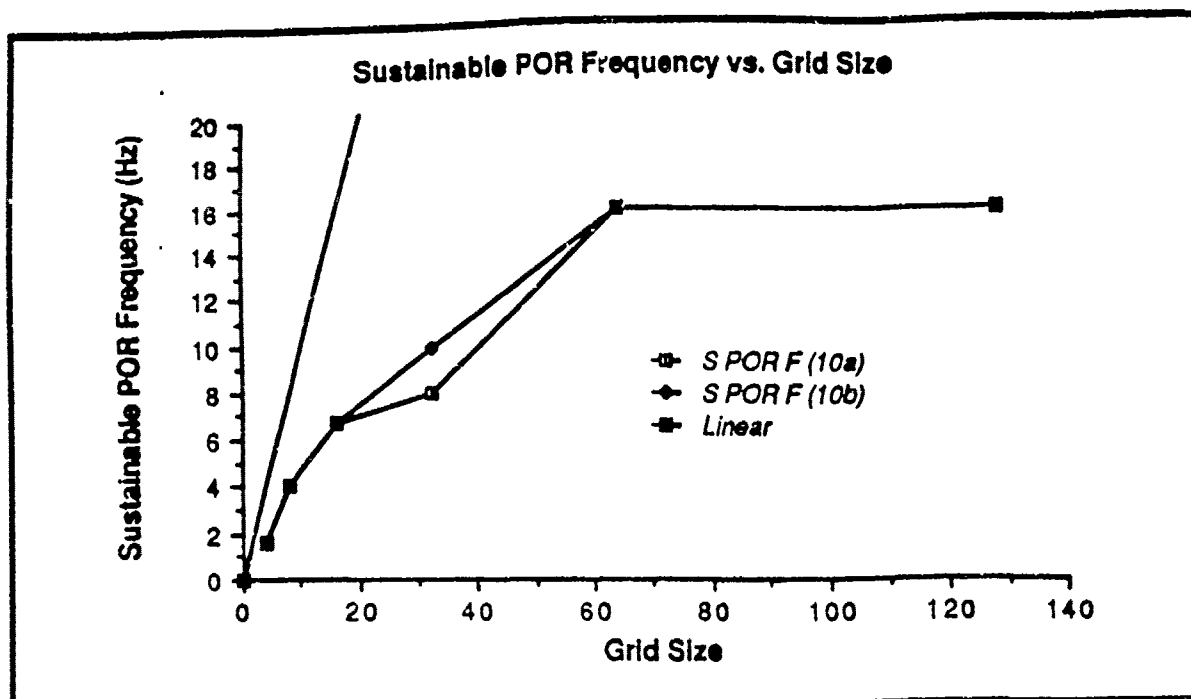


Figure 6.1 Basic Speedup Experiment Results
Sustainable data rates for CONTROL-10 scenario on grids 4, 8, 16, 32, 64, and 128, with alternative load balancing schemes for grid size 32.

The following should be noted:

- Each point, except for grid size 4, represents the minimum sustainable data rate of all three three latencies measured.
- Results for the 4-grid require special explanation. For this small number of sites all managers shared sites with other managers *and* subordinates and were so overloaded that, even for the slowest data rate tested, there was no sustainable data rate for both update latencies, i.e., these latencies exceeded their respective thresholds for all data rates tried. The point for grid size 4 is the sustainable data rate for initial fusion latency alone.
- For grid sizes 4, 8, and 16, managers and subordinates alike share all sites.
- For grid sizes 64 and 128, all managers have dedicated sites and subordinates share the remaining sites.
- For grid size 32, two allocation schemes were tried. The low point, 10a, corresponds to 10 managers with dedicated sites and 12 managers and all subordinates sharing the remaining (22) sites. The high point, 10b, corresponds to dedicated manager sites for all 22 managers with subordinates sharing the remaining (10) sites.

Queue data for 10a shows long queues for some of the managers lacking their own site. The marked increase in performance can clearly be explained by manager bottlenecking in the former scheme, which is mitigated in the latter since previously affected managers no longer compete with subordinates for computational resources.

- Absolute latencies decrease with grid size (as observed for Data Association).
- For grid size 8 and beyond, the sustainable data rate is increasingly dictated by the initial fusion latency. For the grid sizes 64 and 128, the two update latencies are almost an order of magnitude smaller (approximately 10ms) than the specified latency thresholds. Initial fusion latencies, on the other hand, are just under the 500ms threshold.
- $S_{64,4}$ speedup is 12, i.e., a twelve-fold increase in the sustainable data rate accompanies a sixteen-fold increase in grid size from 4 to 64 sites. This is in contrast to Data Association's achieved $S_{64,4}$ speedup of 16, i.e., linear speedup.
- No additional speedup is observed beyond 64 sites. This indicates that Path Association processing this input data (no exceptions) is manager-limited, not subordinate-limited. Manager queue data also supports this conjecture.

6.2.2.2. Exceptions experiment

Figure 6.2 below shows results of the exceptions experiment.

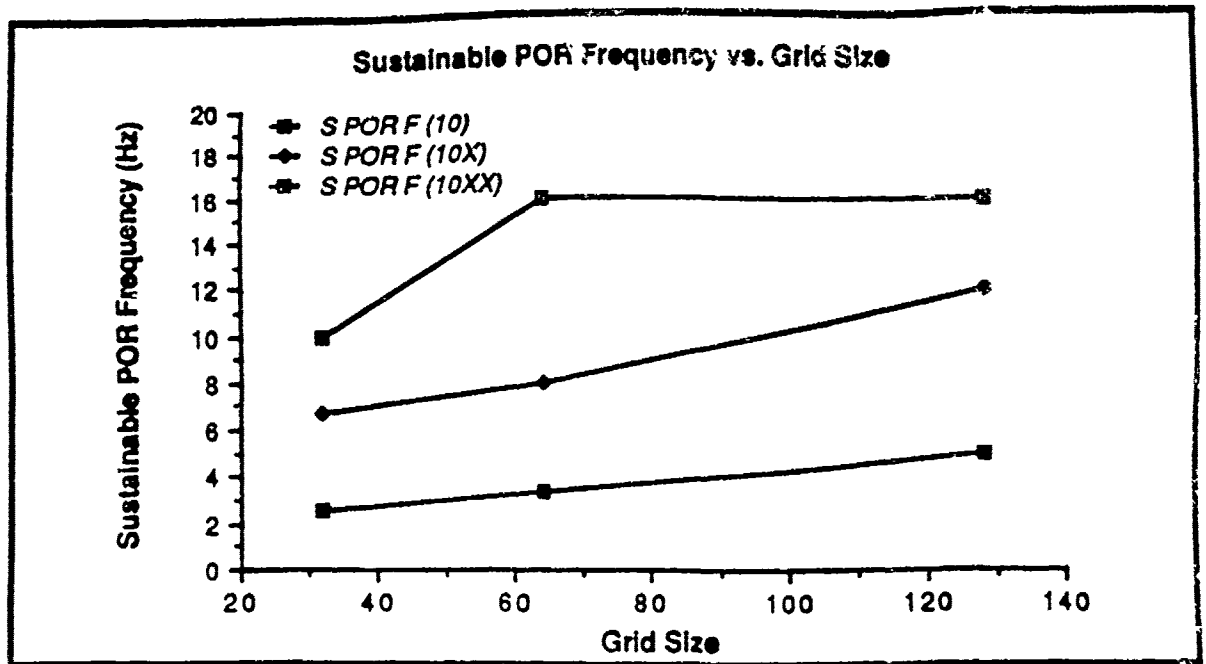


Figure 6.2 Exceptions Experiment Results
Sustainable data rate for three scenarios with increasing numbers of exceptions:
CONTROL-10: 0%; CONTROL-10X: 10%; CONTROL-10XX: 30%.

The following should be noted:

- Performance continues to improve beyond 64 sites for both exception-full scenarios (CONTROL-10X and CONTROL-10XX). This suggests that resources underutilized (sites for dynamic objects) in processing the exception-less (CONTROL-10) scenario are utilized more fully for the exception-full scenarios.
- Performance for CONTROL-10X is only marginally worse than for CONTROL-10 (12Hz vs. 16Hz for a grid size of 128).
- Performance for CONTROL-10XX is significantly worse than either CONTROL-10 or CONTROL-10X and improves less with increasing grid size.

6.2.3. Discussion

- *Load balance:*

How uniformly the computational load is distributed both spatially and temporally over the sites in the multiprocessor.

The basic speedup experiment clearly supports our first hypothesis that (for the given input scenario) performance improves directly with grid size. Two observations, however, indicate that Path Association as implemented is manager-limited. The most obvious of these is the lack of performance gain

beyond 64 sites. Recalling that all managers reside on dedicated sites and that the number allocated is the same for both the 64 and 128 grid (namely 22 sites), the only thing distinguishing the 128 grid from the 64 grid is the presence of an additional 64 sites for dynamic objects. If the system were subordinate-limited, these additional objects would reduce bottlenecks and lead to improved performance on the larger grid.

The second observation is the marked difference in performance for the two manager allocation schemes used for grid size 32. The poor observed performance for the first scheme in which some managers share sites with subordinates indicates the presence of manager bottlenecks. This bottlenecking is alleviated in the second scheme because all managers have their own site.

The exceptions experiment supports our second hypothesis that (for a given grid size) performance degrades with increasing input scenario exceptions. Given our system design in which exception detection is cheap but exception handling relatively expensive, this result is not unexpected. Small numbers of exceptions are tolerated with only slight loss of performance, but large numbers drain computational resources and lead to severe performance degradation. We believe this approach is justified given the domain of aircraft tracking; exceptions of the kind generated for the purposes of this experiment are not commonly observed in practice¹.

Path Association clearly suffers from load imbalance and it is apparent that performance is impaired by a few extremely busy managers. Queue length and service time data indicate that the PMs are the most bottlenecked, followed by certain FPCs; FPMs do not appear to be overloaded. Furthermore, the sustainable data rate is heavily dictated by the initial fusion latency (except for grid size 4) which is further evidence for manager bottlenecking, since a well-tuned, "balanced" Path Association would ideally sustain approximately the same data rates for all three types of latencies. We cannot use this data alone, however, to determine which types of managers are most severely overloaded since we have no latency data which includes connection time and does *not* include fusion time, or vice versa; neither FPS nor P updates are routed via an FPC or PM. Queue data actually indicates that both types of managers for one aircraft type² are considerably busier than those of the other two aircraft types, although the difference across aircraft types is less marked for PMs than for FPCs. In any case, the (unimplemented) solution is to decompose PM and FPC managerial tasks, i.e., further distribute the work of these managers, particularly the work of the PMs.

¹ Unfortunately, the exception handling code took most of the time to design, develop and debug.

² The number of POKs differs for each of type of aircraft; the scenario is in effect the superimposition of three single-aircraft scenarios.

The performance of Path Association is influenced adversely by the need for coordination between managers and subordinates inherent in the connection and fusion processes. This need for coordination, or synchronization, is a serious threat to concurrency. Indeed, a pessimistic conjecture is that concurrency achievable with complex reasoning problems in general will be limited by the synchronization requirements of their sub-parts, i.e., sub-problem dependencies. Further conjecture is fruitless, however, until the known bottlenecks have been first removed from Path Association. Potential remedies are discussed in Chapter 8.

Processor utilization is relatively poor in Path Association, again due to load imbalance. For example, for grid size 32 and a sustainable POR frequency of 10Hz (1 POR every 100ms), less than 50% of the processors are utilized 80% of the time (with naturally less processor utilization at slower frequencies).

Table 6.2 compares data rates for Path Association and previous LAMINA applications. This table is offered as a rough guide to quantitative performance; care should be taken interpreting the numbers since each application not only runs from different input data but employs a different criterion for sustainability¹. The larger Path Association data rate reflects both a very much wider input scenario, many more LAMINA and LISP objects, and much more computation.

Table 6.2 Data Rates for Various LAMINA Applications

<i>ELINT</i>	< 1ms per scan
<i>Data Association</i>	5ms per scan
<i>Path Association</i>	12ms per scan (60ms per POR)

In conclusion, Path Association achieves monotonically decreasing speedup up to 64 or 128 sites, depending on input data. Although poor contrasted with Data Association's linear speedup up to 100 sites, this result is perhaps reasonable given the the complexity of the connection and fusion processes² and the amount of input data processed. Furthermore, given our current understanding of bottleneck locations, we expect to improve upon this result with future design enhancements.

¹ In fact, Data Association has no notion of sustainable data rate.

² By compiled LAMINA code count alone, Path Association is almost ten times larger than Data Association.

6.3. Qualitative Performance

Qualitative performance naturally requires some metric of quality. For Path Association this is the quality of the resulting Platforms,- but evaluating this is far from trivial. Data Association was completely reimplemented in BB1 which provided a serial implementation to compare results against; output from the serial program was by definition "correct."¹ Path Association has no serial equivalent and a reimplementation does not seem viable on account of its size. The approach to date has been to visually inspect Platform output and check for "expected" connections and fusions which do not materialize. The most that can be said is that Path Association does not fail by these criteria. Connections and fusions are imperfect on occasion, but these results are explainable in terms of the heuristics and parameters governing these operations. The issue of measuring qualitative performance remains one which has yet to be addressed effectively.

¹ BB1 is a serial blackboard system.

7. Lessons

This chapter describes the lessons we have learned in the course of developing Path Association.

7.1. Programming Paradigm

The Path Association system embodies many interesting ideas about parallel programming on CARE-like architectures. Many evolved out of earlier work within the Advanced Architectures Project developing ELINT (in CAOS and LAMINA) and Data Association. The elements of this programming paradigm can be summarized as follows.

- Object-oriented
- Structured in terms of managers and subordinates
- Concurrency through replication and pipelining¹
- Implicit continuations
- Control through handshaking
- High-level constructs (initialization, creation, deallocation, sending)

7.2. Program development

The First Law of Parallel Programming could be stated simply as:

"It will go wrong."

Humans are essentially serial thinkers.² Our intuitions and assumptions can easily lead us astray programming concurrent systems. Only by checking these instincts and taking the time to investigate the full ramifications of concurrency is it possible to achieve successful parallel programs. This is especially true of symbolic applications, such as AIRTRAC, where time/order dependencies are often subtle and yet critical. On several occasions these dependencies were only apparent *a posteriori*. Chiefly for this reason, it has been our experience that approximately one order of magnitude more time is required to develop parallel programs than their serial equivalents. Table 7.1 compares development time for the two implementations of Data Association.

¹ Path Association has only a loose pipeline structure; a better example is Data Association.

² This is not to deny the inherent underlying parallelism. High-level reasoning, on the other hand, seems best characterized as a serial process.

Table 7.1 Data Association Implementation Times

BB1	Two person-weeks
LAMINA	One person-year

There is, however, reason for optimism. Both the ELINT and Data Association applications which consist of less than 50KB of compiled LAMINA code took approximately one person-year to develop. Path Association which consists of over 300KB of compiled code was completed in the same amount of time¹. Clearly, we are doing something better. A contributing factor which should not be understated is what we have learned from previous applications. Probably the biggest contributor, however, has been the intermediate ELMA layer we developed to support the application. Needless to say, considerable time has been gained by being able to program at a higher level of abstraction. Debugging time has also been slashed, not only because many ELMA constructs were debugged independently of Path Association but also because of the various application-level meters provided.

7.3. Load Balancing

Speedup of parallel applications is invariably limited by bottlenecks which arise due to poor load balancing. Results of the basic speedup experiment indicate that Path Association performance is certainly no exception. Two load-balancing issues are involved.

The first involves the placement of newly-created objects. The goal is to distribute such objects as evenly as possible without recourse to a centralized site allocation facility.² Several schemes have been used in the past.

- *statistical*: each site has information about neighboring sites
- *knowledge-based*: uses knowledge about the domain
- *random*

We used the same modified random load balancing used by Data Association [Hailperin 87, Nakano 87] which essentially involves random selection from the set of all sites excluding those used by managers (providing the multiprocessor is large enough). Excluding dynamic objects from manager sites works well, although it is likely that some managers do not require a dedicated site.

¹ Excludes graphics interface, ELMA, analysis code, etc.

² Actually, the goal is to distribute objects such that the work they do is as even as possible.

The second issue involves overloaded objects, such as some Path Association managers. When a site cannot cope with the load of even a single resident LAMINA object, the work of that object must be divided and distributed. This then reduces to the basic problem of decomposing a task into multiple, independent sub-tasks. Unfortunately, there are no clear-cut rules for doing this. Furthermore, the potential need for synchronization among sub-tasks could become a limiting factor which prevents finer granularity and thus better load balancing.

7.4 Performance evaluation

Performance evaluation of continuous parallel systems is difficult. The notion of sustainable data rate has proven to be useful for quantitative performance evaluation. Qualitative performance evaluation is tied more closely to the nature of the application and is less tractable. One way to evaluate qualitative performance is to re-implement the application serially and compare results of the serial and parallel programs.

8. Future Work

With only two series of AIRTRAC experiments performed to date, many interesting experiments and enhancements remain for future work.

Additional experiments worth pursuing with the present system include:

- measure performance gains afforded by free pool mechanism; verify or disprove that ELMA free pools offer performance advantages in addition to the proven syntactic ones.
- measure/analyze timing data to determine the cost of exception recognition and handling; verify or disprove that exception recognition is cheap while exception handling is expensive.
- measure an additional latency, namely initial connection latency. This would enable the cost of connection to be factored out of the initial fusion process.
- measure platform correctness (qualitative performance). UFP timeouts, for example, can be used as a metric, although others will also be needed.
- measure/analyze queue data to determine if there are managers which do not need dedicated sites.
- remeasure sustainable data rates using an alternative definition [Hailperin 88]: given a data rate and a grid size, what is the widest scenario sustained?

Slightly more ambitious but interesting work would entail reimplementing parts of the system in order to alleviate the load imbalance. Alternative designs to be explored include:

- pipelined managerial tasks
- "time-sliced" managers, i.e dynamically created managers which process a particular time interval

Finally, future work, in the form of major software additions, includes:

- extending ELMA to support dynamic managers
- integrating ELMA history/reporting mechanisms into the simulation environment for more interactive performance feedback
- designing and implementing Path Interpretation

9. Summary

This paper has described the development of the Path Association module of AIRTRAC, a knowledge-based application written in LAMINA for the CARE family of multiprocessors. The high-level goal of AIRTRAC is to monitor the flight of aircraft in a particular region of airspace and to interpret and predict their behavior, given continuous tracker data from one or more radar sites within the region.

AIRTRAC has two conceptually distinct, yet very related, parts: the solution to the aircraft tracking problem (the knowledge-based portion), and the realization of that solution in an appropriate software architecture (the multiprocessor portion). We described in great detail the design and implementation of the system and show that it is very difficult to separate the two parts in practice.

We developed a set of high-level programming constructs and library objects in the course of developing the Path Association, collectively referred to as ELMA. By programming at a higher level of abstraction the task of implementing the application was greatly simplified. In this paper we described the salient features of ELMA:

- Syntax and constructs for managing concurrency and memory usage.
- A library of definitions of special-purpose LAMINA objects.
- A library of useful abstract data types.

We next defined and then refined the notion of a *sustainable data rate* for the quantitative performance evaluation of continuous parallel systems. *Sustainable data rate* $SDR_{a,b}$ was defined as the input data rate for which absolute latencies are below a threshold of a at least b percent of the time. We applied this criterion to confirm the following experimental hypotheses concerning Path Association:

- For a given input scenario, performance improves directly with increasing numbers of sites (grid size).
- For a given grid size, performance degrades with increasing input scenario exceptions, slightly at first and then markedly.

We described LAMINA/ELMA programming style and tips for parallel programming CARE-like architectures. We defined and justified the First Law of Parallel Programming, "*It will go wrong.*" and showed how load imbalance limits performance.

We obtained monotonically decreasing speedup up to a maximum of 64 to 128 processors, depending on input data. More importantly, we were able to explain the factors limiting performance and suggest improvements. Although poor compared to Data Association, this result is perhaps reasonable given the complexity of the connection and fusion processes and the amount of input data processed. Future design enhancements should see performance improvements.

The following questions remain unanswered:

- Does the described free pool mechanism offer performance advantages over ad hoc creation?
- How should qualitative performance be evaluated?
- Are there are better approaches to evaluating quantitative performance?
- Will concurrency of complex reasoning problems suffer from subproblem dependencies which limit the granularity of processing load and thus limit load balance?

In conclusion, we believe that the techniques and constructs that we have described and the lessons we learned in the course of developing Path Association will be useful to others working in the field of parallel symbolic computation.

Acknowledgements

We are indebted to all members of the Advanced Architectures Project for nurturing a rich research environment and for their tireless support, especially Harold Brown, who provided valuable (and essential) guidance throughout the project. We would also like to thank Bruce Delagi, Nakul Saraiya, and Sayuri Nishimura who built and maintained the CARE/LAMINA system, James Rice for LISP machine magic, and Hirotoishi Maegawa, Djuki Muliawan, and Max Hailperin for insights provided en route. The Symbolic Systems Resources Group of the Knowledge Systems Laboratory provided excellent support of our computing environment. Finally, special thanks are due to Edward Feigenbaum for his leadership and support of the Knowledge Systems Laboratory and the Advanced Architectures Project which made this research possible.

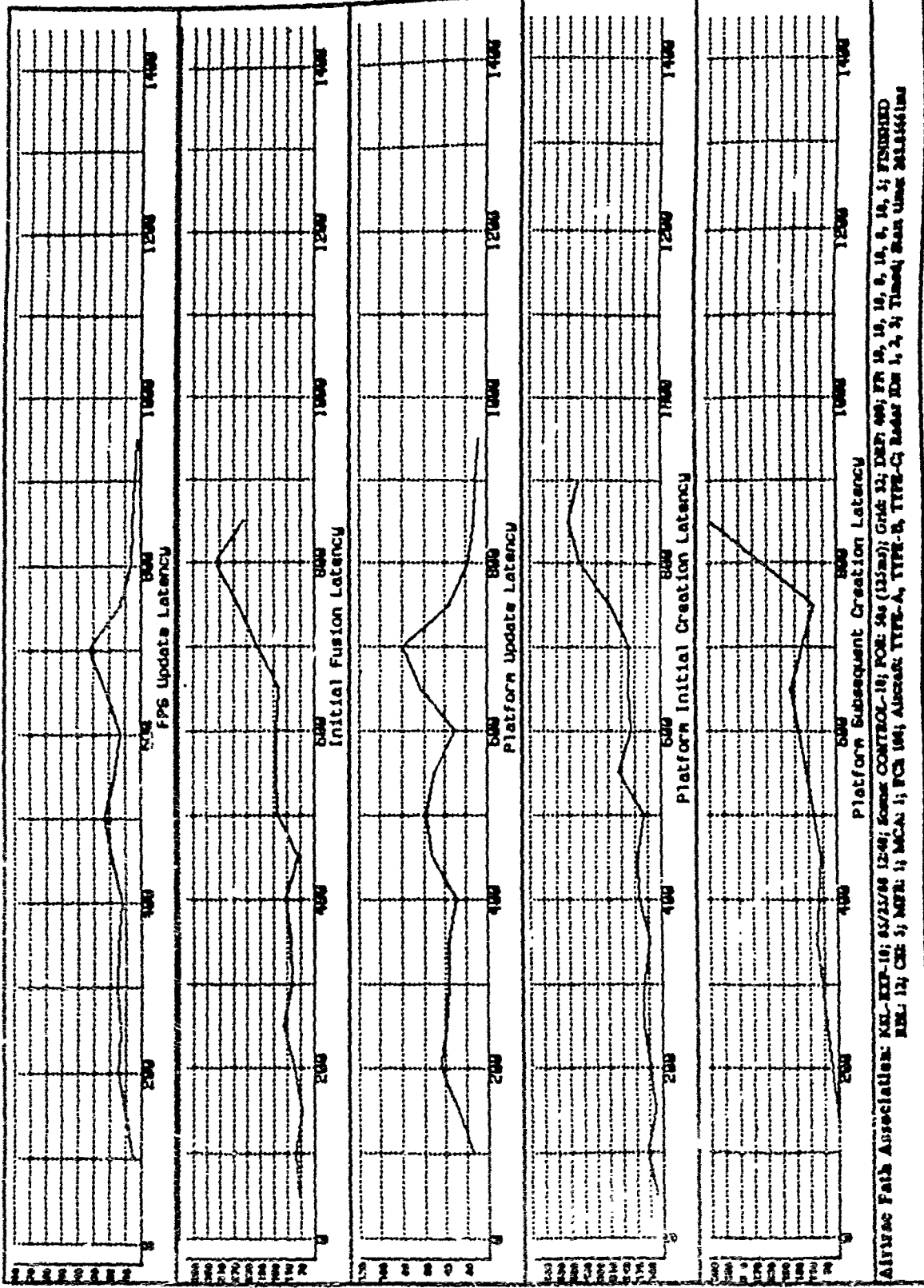
This work was supported by DARPA Contract F30602-85-C-0012 and NASA Ames Contract NCC 2-220-S1.

References

- [Brown 86] H. Brown, E. Schoen, B. Delagi. *An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures*, Report No. STAN-CS-86-1136 (also numbered KSL-86-69), Department of Computer Science, Stanford University, 1986
- [Delagi 87a] B. Delagi, N Saraiya, G. Byrd. *Instrumented Architectural Simulation*, Report No. KSL 86-65, Knowledge Systems Laboratory, Department of Computer Science, Stanford University
- [Delagi 87b] B. Delagi, N Saraiya, G. Byrd. *LAMINA: CARE Applications Interface*, Report No. KSL 86-87, Knowledge Systems Laboratory, Department of Computer Science, Stanford University
- [Hailperin 86] M. Hailperin, private communication, July 1987
- [Hailperin 88] M. Hailperin, private communication, March 1988
- [Nakano 87] R. Nakano, M. Minami. *Experiments with a Knowledge-Based System on a Multiprocessor*, KSL 87-61, Knowledge Systems Laboratory, Department of Computer Science, Stanford University
- [Noble 88] A. Noble. *ELMA Programmers Guide*, Report No. KSL-88-42, Working Paper, Knowledge Systems Laboratory, Department of Computer Science, Stanford University

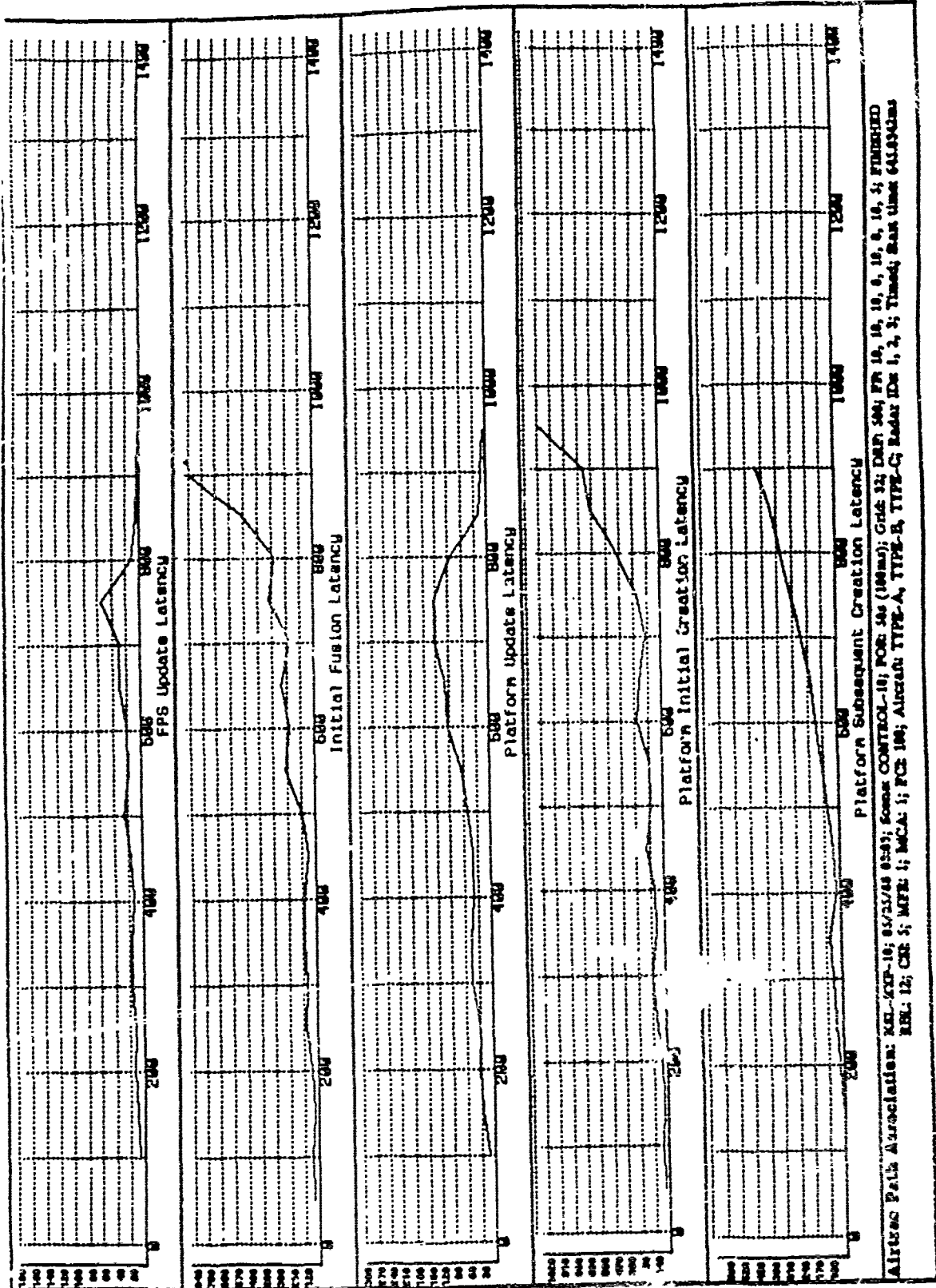
Appendices

Appendix 1 Sample Latency Graphs



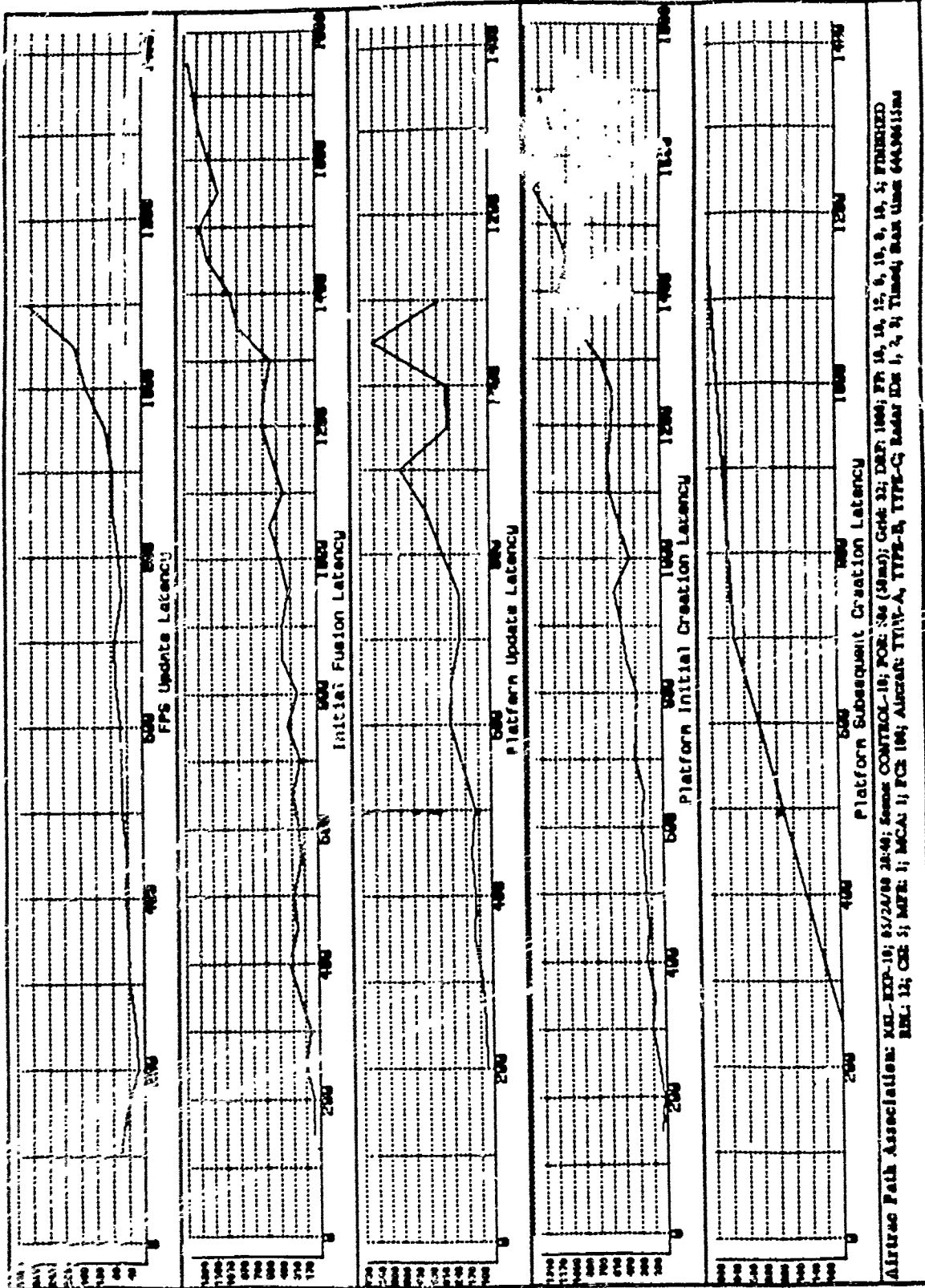
AIRFAC Path Association: KIL-EXP-10; 6/15/08 12:40; Sensor CONTROL-10; POS: 56 (125m); GRN: 12; DRP: 00; FN: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100; MCA: 1; PCA: 100; Allocated: TYPE-A, TYPE-B, TYPE-C; Radar ID: 1, 2, 3; Thermal Scan Line: 2611661m

Safety sustainable, i.e. underloaded



Airtrac Path Association: XEL-21P-10; 01/25/00 03:57; Socks CONTROL-10; FOR: 50 (10000); Gid: 32; DMR: 504; FR: 10, 10, 6, 10, 5, 10, 5; FINISHED
 IBL: 12; CR: 5; MCA: 1; MCA: 1; FCE: 10; Aircraft: TYPE-A, TYPE-B, TYPE-C; Radar: Ids: 1, 2, 3; Thread: Sock Unit: 64100ms

Just sustainable by SDR500ms,0.9 criterion

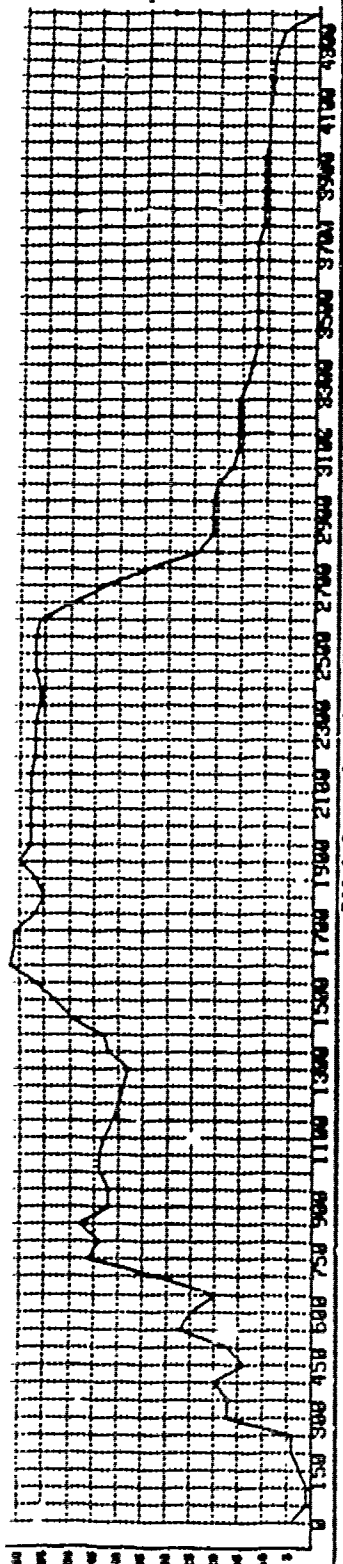


Not sustainable, i.e., overloaded

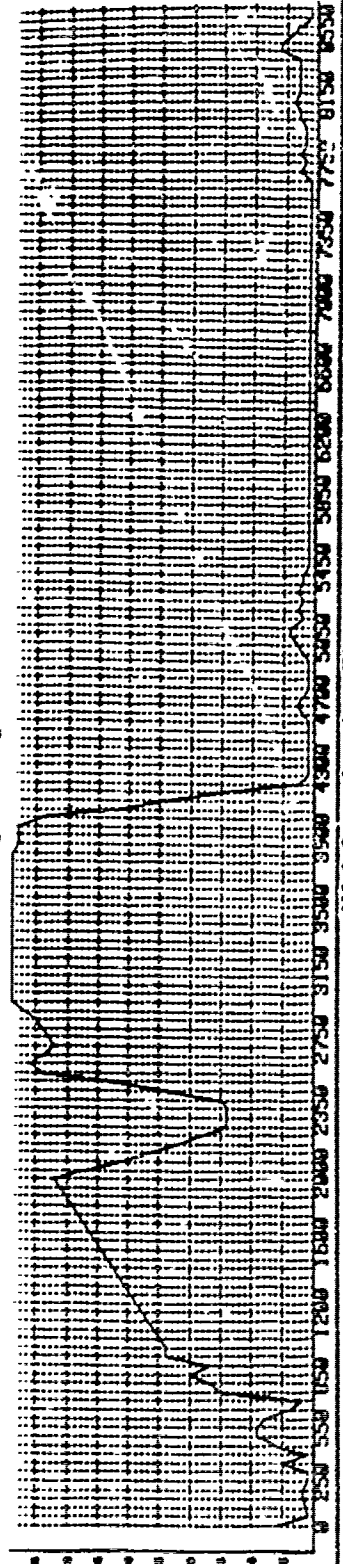
Appendix 2 Sample Queue Graph

Airtac Path Association: XEL-EXP-18; 85/12/88 12:46; SCENE CONTROL-18X; FOR: 365 (17ms); Grid 128; DREF: 3009, 27, 19, 14, 8, 10, 6, 14, 5; FINISHED
 REL: 12; CSE: 5; MFR: 1; MCA: 1; FCE: 169; ALTRAC: TYPE-A, TYPE-B, TYPE-C; Robot ID# 1, 2, 3; Turned Start Times: 773, 806, 1304

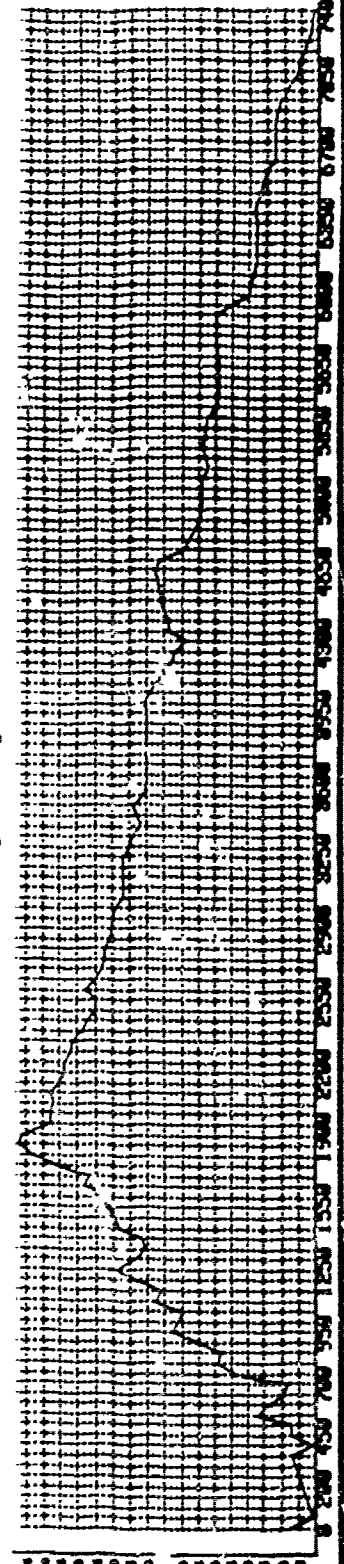
PM-0 Queue Length vs Time



PM-1 Queue Length vs Time



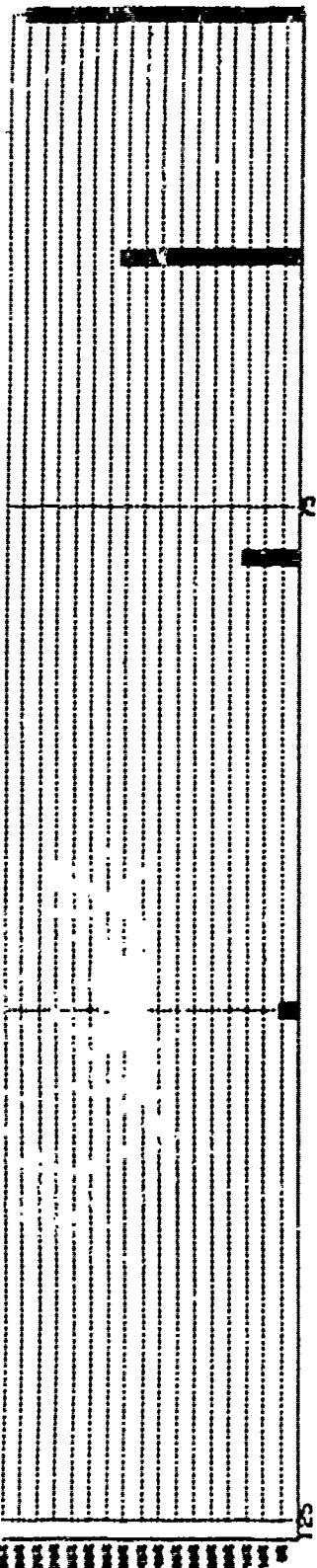
PM-2 Queue Length vs Time



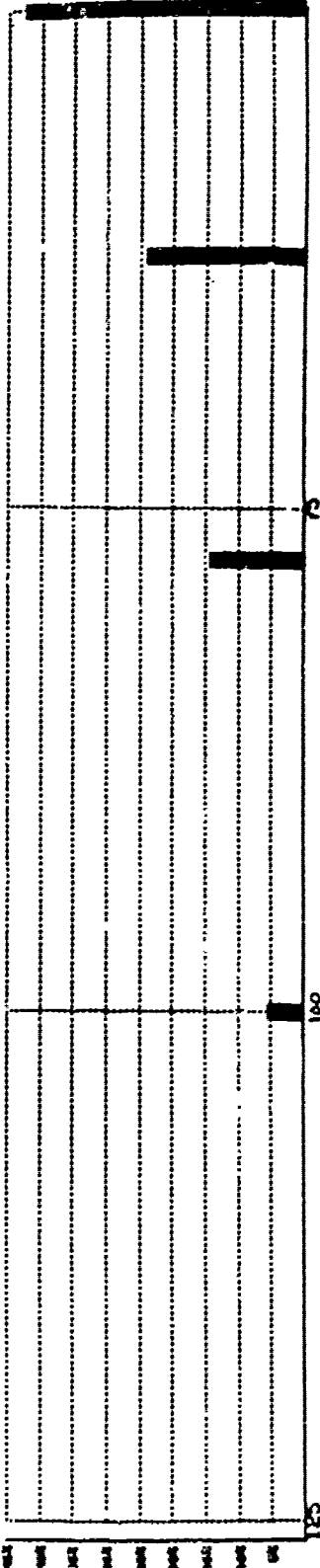
Queue data indicating severe manager overload

Appendix 3 Sample Latency Threshold Graph

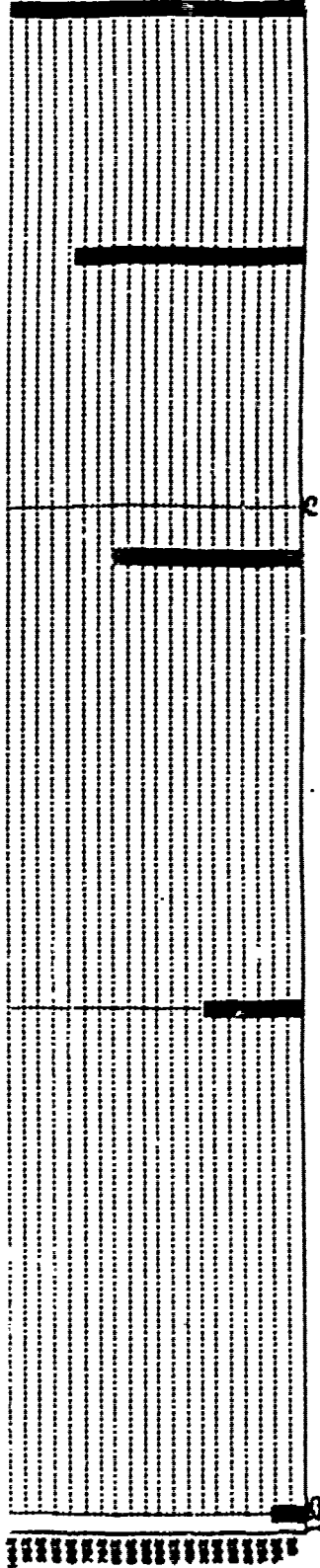
% TIME FPS-UPDATE-LATENCY THRESHOLD EXCEEDED (above 40 ms) VS FOR PERIOD (ms)
 [CONTROL-14; 32 Grid] [125.0, 100.0, 77.5, 62.5, 50.0 ms]



% TIME INITIAL-FUSION-LATENCY THRESHOLD EXCEEDED (above 500 ms) VS FOR PERIOD (ms)
 [CONTROL-10; 32 Grid] [125.0, 100.0, 77.5, 62.5, 50.0 ms]



% TIME PLATFORM-UPDATE-LATENCY THRESHOLD EXCEEDED (above 60 ms) VS FOR PERIOD (ms)
 [CONTROL-10; 32 Grid] [125.0, 100.0, 77.5, 62.5, 50.0 ms]



Initial Fusion Latency SDR500ms,0.9 = 100ms

Appendix 4 Example ELMA Program

```
;;; -*- Mode:Common-Lisp; Package:CARE-USER; Fonts:(CPTFONT TR10B TR10I); Base:10 -*-
```

```
;; Author: Alan C. Noble
```

```
;; This file contains a complete but simple ELMA program,  
;; including SIMPLE simulator interface.
```

```
(defvar *free-pool-init-length* 2)  
(defvar *free-pool-threshold-length* 1)
```

```
;; *****  
;;  
;; PROGRAM EXPLANATION  
;;  
;; *****
```

```
;; Two user object types are defined: BOSS, an ELMA-manager, and CLERK, an ELMA-subordinate.  
;; The (start-ELMA) call initializes the program by creating an ELMA-allocator which in turn  
;; creates a BOSS object. The ELMA-allocator automatically sends a :get-ball-rolling message  
;; to the BOSS when it receives a creation acknowledgement from the latter.  
;; The BOSS allocates three CLERKS, Alfred, Oliver and Barry, and sends them an :apple, :orange  
;; and :banana message respectively. The BOSS then deallocates Oliver and Barry and  
;; then sends them various messages. Since they are deallocated, the incorrect address mechanism  
;; handles these messages and automatically returns, forwards or drops them according  
;; to settings.
```

```
;; *****  
;;  
;; OBJECT DEFINITIONS  
;;  
;; *****
```

```
;; BOSS
```

```
(defflavor boss  
  ()  
  (elma-manager)  
  :settable-instance-variables  
  (:documentation "Boss--doles out work to the clerks")  
  )
```

```
;; CLERK
```

```
(defflavor clerk  
  ()  
  (elma-subordinate)  
  (:documentation "Clerk--does all the work")  
  (:default-init-plist  
   ; record incoming messages for these triggers  
   :notable-triggers '(:apple :orange :banana)  
   ; time the execution of these trigger methods  
   :timed-triggers '(:apple :orange :banana)  
   :incorrect-address-action '( ; default settings for CLERK instances  
                               (:apple . .RETURN) ; i.e., return all apple messages to sender  
                               (:orange . :RETURN) ; i.e., return all banana messages to sender  
                               ; :banana not specified, so dropped by default  
                               )  
  )
```

```

;; *****
;;
;; TRIGGER METHOD DEFINITIONS
;;
;; *****

;; BOSS

(defun boss-init-list-fn (clerk-sites)
  "Returns an init-list for a boss object, in this case the :free-pools
  slot and its value"
  (list :free-pools (list (make-free-pool
                           :node-type 'CLERK ;; each boss has a free pool of clerks
                           :initial-length *free-pool-init-length*
                           :threshold-length *free-pool-threshold-length*
                           :sites clerk-sites))
        )
  )

(defun trigger (boss :get-ball-rolling) ()
  "This trigger gets the ball rolling."
  ;; Allocate three clerks from the pool
  (with-subordinate (alfred)
    (with-subordinate (oliver
                      :after-messages
                      ((oliver `(:pineapple))))
      ;; i.e., send :pineapple message to object named oliver as soon as it is allocated
      )
    (with-subordinate (barry)
      ;; get clerks working
      (mailing alfred :apple nil)
      (mailing oliver :orange nil)
      (mailing barry :banana nil)
      ;; deallocate oliver and barry
      (deallocate oliver )
      (deallocate barry
                  :incorrect-address-action
                  `(:FORWARD :orange (,alfred))) ;; override the default so :orange is forwarded
      ;; send oliver and barry some work and see what happens
      (mailing oliver :apple nil) ;; this should be returned to the boss
      (mailing barry :apple nil) ;; this should also be returned
      (mailing barry :orange nil) ;; this should get forwarded to alfred
      (mailing barry :banana nil) ;; this should get dropped
      )
    )
  )

(defun trigger (BOSS :returned-message)
  ((original-message sent-name new-name original-args))
  "Trigger for handling returned message"
  (debug-format output-stream "~%=> ~a received a RETURNED-MESSAGE" name)
  (debug-format output-stream "~% Message ~a was sent to ~a, now named ~a, with args ~s"
                original-message sent-name new-name original-args)
  )

```

```
;; CLERK
```

```
(deftrigger (CLERK :apple) ()  
  (untimed-format output-stream "~t~a got an APPLE" name)  
  )
```

```
(deftrigger (CLERK :orange) ()  
  (untimed-format output-stream "~t~a got an ORANGE" name)  
  )
```

```
(deftrigger (CLERK :banana) ()  
  (untimed-format output-stream "~t~a got a BANANA" name)  
  )
```

```
(deftrigger (CLERK :pineapple) ()  
  (untimed-format output-stream "~t~a got a PINEAPPLE" Name)  
  )
```

```
;; *****
```

```
;;
```

```
;; SIMULATOR INTERFACE
```

```
;;
```

```
;; *****
```

```
;; The rest of this file defines functions which set up the simulator and start a simulation.
```

```
;; The top level function is START-ELMA.
```

```
(defun elma-example (soptional skey initialize  
  (circuit 'care:octorus-32) (instrument 'care:observer))  
  "Run elma example program.  
  Initialize simulator if initialize is t."  
  (when initialize ;; initialize the simulator  
    (simple :run nil :reset t :flush t :circuit circuit :instrument instrument);  
    (let ((clerk-sites (cl:remove 0 (care-site-numbers)))) ;; use six sites except boss site  
      (start-ELMA  
        :initialization-parameters  
        ((:ALLOCATOR . "the-ALLOCATOR")  
         (:ALLOCATOR-TYPE . ELMA-allocator)  
         (:ALLOCATOR-SITE . 0)  
         (:START-MESSAGE . :get-ball-rolling)  
         (:START-MESSAGE-RECIPIENT . "the-BOSS")  
         (:MANAGER-INITIALIZATIONS ("the-BOSS" ;; name of the manager  
                                     BOSS ;; type of the manager  
                                     0 ;; site of the manager  
                                     boss-init-list-fn ;; function to generate init-list of manager  
                                     (,clerk-sites) ;; args of function  
                                     NIL))  
        )  
      )  
    )  
  )  
  'compile-flavor-methods clerk)  
(compile-flavor-methods boss)
```



```
> (pkg-gr 'o 'care-user)
> (setq *ELMA-debug* t)
> (elma-example)
```

```
Time 4.0987: the-ALLOCATOR CREATED
FOR-EFFECT
Time 7.4321003: the-BOSS CREATED
Time 15.5573: CLERK-0-of-the-BOSS CREATED
Time 10.5774: CLERK-1-of-the-BOSS CREATED
> CLERK-2-of-the-BOSS creation requested
> CLERK-3-of-the-BOSS creation requested
> CLERK-4-of-the-BOSS creation requested
Time 16.2048: CLERK-2-of-the-BOSS CREATED
Time 17.398901: CLERK-3-of-the-BOSS CREATED
CLERK-3-of-the-BOSS got a PINEAPPLE
Time 17.764702: CLERK-4-of-the-BOSS CREATED
CLERK-2-of-the-BOSS got an APPLE
CLERK-4-of-the-BOSS got a BANANA
Time 30.1036: CLERK-4-of-the-BOSS DEALLOCATED-AND-RENAMED
> Incorrect address action for CLERK-6-of-the-BOSS;
  Returning APPLE to sender the-BOSS
> Incorrect address action for CLERK-6-of-the-BOSS;
  Forwarding ORANGE to (CLERK-2-of-the-BOSS)
> Incorrect address action for CLERK-6-of-the-BOSS;
  Dropping BANANA
CLERK-3-of-the-BOSS got an ORANGE
Time 31.748: CLERK-3-of-the-BOSS DEALLOCATED-AND-RENAMED
> Incorrect address action for CLERK-5-of-the-BOSS;
  Returning APPLE to sender the-BOSS
> CLERK-4-of-the-BOSS deallocated and renamed CLERK-6-of-the-BOSS
CLERK-2-of-the-BOSS got an ORANGE
==> the-BOSS received a RETURNED-MESSAGE
  Message APPLE was sent to CLERK-4-of-the-BOSS,
  now named CLERK-6-of-the-BOSS, with args NIL
> CLERK-3-of-the-BOSS deallocated and renamed CLERK-5-of-the-BOSS
==> the-BOSS received a RETURNED-MESSAGE
  Message APPLE was sent to CLERK-3-of-the-BOSS,
  now named CLERK-5-of-the-BOSS, with args NIL
NIL
```

Knowledge Systems Laboratory
Report No. KSL 88-42

August 1988

ELMA Programmers Guide

by

Alan C. Noble

WORKING PAPER

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

*This work was supported by DARPA Contract F30602-85-C-0012
and NASA Ames Contract NCC 2-220-S1.*

1 Introduction

ELMA (Extended LAMINA for Memory-management Applications) is a high-level parallel programming interface to the CARE family of distributed-memory multiprocessors. It is built on top of LAMINA, the basic language interface to CARE¹ which provides primitive mechanisms and language syntax for expressing and managing concurrency and locality. ELMA is an extension of object-oriented LAMINA² and ELMA objects are in fact specializations of LAMINA objects.

As in other object systems, LAMINA objects encapsulate state (instance variables) and behavior (methods). Methods are invoked by message sending but unlike the case of sequential systems this involves transmitting a packet containing the message from one object to another, typically on different *sites*, or processor/memory units. Message sending is non-blocking and the time required for communication is thus visible to the LAMINA programmer. Methods run atomically within processes which are usually restartable but not resumable. An object and its methods can be considered a non-nested monitor; exclusion is guaranteed by the fact that only one method is ever scheduled to run at a time, and then runs to completion. The time required to create a LAMINA object is also visible to the programmer. The reader is referred to *LAMINA: CARE Applications Interface*,³ for a detailed description of LAMINA.

ELMA is tailored to CARE applications which involve extensive dynamic object creation and deallocation and thus require some form of memory management. Its syntax and constructs facilitate programming in the object-oriented style at a higher level than LAMINA. This makes program development easier in ELMA than in LAMINA for this class of applications. ELMA is a complete programming interface; strictly speaking, no knowledge of LAMINA is required to mount an application written in ELMA on CARE.

ELMA provides the CARE programmer with the following:

- A library of definitions of specialized objects, for the programmer to further specialize ("mix in") if required.
- Syntax and constructs for managing concurrency and memory usage.
- A library of useful abstract data types.

These features are described in the following sections.

¹CARE is a distributed-memory message-passing architecture, simulated by a highly-instrumented system called SIMPLE. Refer to: B. Delagi, N. Saraiya and G. Byrd, *Instrumented Architectural Simulation*, Report No. KSL-86-65, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

²LAMINA actually supports three styles of programming, namely functional, shared-variable, and object-oriented.

³B. Delagi, *LAMINA: CARE Applications Interface*, KSL Report No. 86-87, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

2 Specialized objects

ELMA includes a small library of specialized object types and mixins to facilitate program development through code reusability. An ELMA application is implemented exclusively in terms of managers, subordinates and allocators. These object types are described in this section.

2.1 Manager

Managers are objects which are allocated statically, i.e., at initialization time, and are typically responsible for tasks involving many (subordinate) objects such as distribution, search, and object creation. Each manager can maintain zero or more free pools of subordinates. The number of managers required depends on the particular application and its input data and must be determined a priori.

The generic ELMA manager is the type ELMA-manager.

```
(defflavor ELMA-manager
  ()
  (acknowledge creation-mixin
   ;; managers acknowledge creation on init-ack-stream
   free-pool-owner-mixin
   ;; managers maintain free pool objects
   ELMA
  )
  (:documentation "For ELMA manager nodes"))
```

2.2 Subordinate

Subordinates are objects which are created dynamically, i.e., at run time, by managers and typically contain the state of the system. Subordinates can be allocated (created), deallocated, and reallocated many times over in the course of program execution.

The generic ELMA subordinate is the type ELMA-subordinate.

```
(defflavor ELMA-subordinate
  ()
  (free-pool-node-mixin
   ELMA)
  (:documentation "For ELMA subordinate nodes"))
```

2.3 Allocator

Allocators are objects which create manager objects at initialization time and start applications. There is only need for one allocator object per ELMA application. The `start-ELMA` call creates the allocator and initiates the initialization process, in accordance with initialization data supplied by the application. This initialization data includes the following: allocator name, allocator type, allocator, site number, start message, start message recipient (name), and manager initialization data.

```
start-ELMA &key initialization-parameters &optional init-list [function]
```

`start-ELMA` starts an ELMA application. *initialization-parameters* is an alist of the form:

```
( (:ALLOCATOR . name-of-allocator)
  (:ALLOCATOR-TYPE . type-of-allocator)
  (:ALLOCATOR-SITE . site-of-allocator)
  (:START-MESSAGE . start-message)
  (:START-MESSAGE-RECIPIENT . start-message-recipient)
  (:MANAGER-INITIALIZATIONS manager-initializations)).
```

An object of type *type-of-allocator* is created with the name *name-of-allocator* on *site-of-allocator*, a *site number* between zero and the total number of sites in the multiprocessor (or CARE "design") less one. Thereafter, the managers specified by the *manager-initializations* are created. When all newly created managers have acknowledged their creation to the allocator, the allocator initiates program execution by sending the *start-message* (with no arguments) to the *start-message-recipient*. The generic ELMA allocator is the type `ELMA-allocator`. The *type-of-allocator* must be `ELMA-allocator` or a specialization of `ELMA-allocator`.

init-list (default `nil`) is a list of alternating slot keywords and values with which to initialize the allocator object. This list should not include `:start-message`, `:start-message-recipient` or `:manager-initializations` since these are specified by *initialization-parameters*. It is intended that this list be used to initialize user-defined slots, not the standard `ELMA-allocator` ones.

```
(defflavor ELMA-allocator
  ()
  (static-allocator-mixin
   ELMA)
  (:documentation "For ELMA static allocator nodes"))
```

Example:

```
(start-ELMA :initialization-parameters
  '((:ALLOCATOR . "the-ALLOCATOR")
    (:ALLOCATOR-TYPE . ELMA-allocator)
    (:ALLOCATOR-SITE . 0)
    (:START-MESSAGE . :get-ball-rolling)
    (:START-MESSAGE-RECIPIENT . "the-BOSS")
    (:MANAGER-INITIALIZATIONS
     ("the-BOSS" ;; name of the manager
      BOSS ;; type of the manager
      0 ;; site of the manager
      boss-init-list-fn
      ;; function to generate init-list of manager
      (,clerk-sites) ;; args of function
      NIL))) )
```

This call initializes a simple application in which there is only one manager, named "the-BOSS." First an ELMA-allocator with the name "the-ALLOCATOR" is created on site number ⁴ zero, i.e., CARE *site* (1 1). As soon as "the-ALLOCATOR" is created, it creates, also on site number zero, an object of type BOSS named "the-BOSS." "the-ALLOCATOR" applies the function boss-init-init-function to (,clerk-sites) to get the init list to create "the-BOSS." "the-ALLOCATOR" automatically sends a :get-ball-rolling message to "the-BOSS" once it has received the mandatory creation acknowledgement from the latter.

3 Syntax and Constructs

3.1 Free Pools

Experience from early applications developed on CARE, such as ELINT and AIRTRAC Data Association, strongly supports the need for strict control of concurrent object creation. Both these applications, however, present rather ad hoc approaches to implementing such control. ELMA provides the programmer with the high-level constructs `with-subordinate` and `with-named-node` for object creation and `deallocate` for object deallocation. These constructs also provide a means of dynamic object memory management, based on the use of free pools.

⁴A CARE *site* is a simulator component, whereas a *site number* is an integer between zero and the total number of sites in the multiprocessor less one. There is a unique mapping from site number to site

The notion of a free pool is easy to understand in a sequential program. The program maintains a list of (pointers to) commonly used objects (or records) which are allocated in advance. New objects are taken from the pool rather than being allocated from heap storage directly, and returned to pool when deallocated. Apart from syntactic nicety, free pools offer an elegant and simple memory interface that puts the application in control of its own memory management (of pool objects). New storage is allocated by simply returning the address of an empty, already allocated object in the pool.

In ELMA, a free pool is a LISP object in the local address space of a manager object which contain pointers⁵ to objects of some type (all the same type), the latter spread out over a predefined pool of sites in the multiprocessor. The function `make-free-pool` returns a free-pool object.

```
make-free-pool &key node-type sites &optional (initial-length *default-free-pool-length*)  
(threshold-length *default-free-pool-length*) (site-function #'random-fp-site) [function]
```

A manager may own multiple free pools or none. In other words the ELMA-manager `free-pools` slot can be a list of zero or more free-pool objects, instantiated by `make-free-pool`. Objects allocated from a free pool of a manager are said to be subordinate objects. As in the sequential case, a new free pool object is allocated by returning its (remote) address. However since objects in the pool are separate processes (typically on remote sites), the storage associated with a newly allocated object can only be (re-)initialized by sending a message to that object. This slot initialization message is automatically prepended to any other messages sent to the object, or sent separately if there are no such messages. As for sequential free pools, a new object is created only when the pool is empty. Unlike sequential free pools, in which new storage is usually allocated within local address space and always allocated on the same processor (since there is only one processor!), a new ELMA object can be allocated on any site of the multiprocessor. The programmer specifies a list of permissible *site numbers* (the *sites* argument in `make-free-pool`) for placement of subordinates of each manager. A function can also be specified (the *site-function* argument) which takes this list as its first argument and returns the CARE site for a new subordinate. If the latter is omitted ELMA chooses a site randomly from the specified list using *random-fp-site*.

3.2 Remote Addresses and Names

In LAMINA, objects are referenced exclusively by means of *remote addresses*, which can be regarded as inter-site pointers. Each object's remote address is unique and invariant since, once created, an object is never relocated to another site. Remote addresses are also referred to as *handles*.

In addition, ELMA gives each object a unique *name*. Static objects, such as managers have exactly one name for the entire duration of the program. Dynamic objects such as

⁵These pointers are *remote addresses*, since they can reference objects on other sites.

subordinates, however, can have many different names in the course of being deallocated and reallocated during program execution. With each reallocation, or *reincarnation*, a dynamic object acquires a new name, although its remote address is of course unchanged. By using names, it is possible to distinguish between different incarnations of an object possessing the same remote address.

Each ELMA object maintains an *address table* which maps names to remote addresses. This is usually a many-to-one mapping. Names are simple strings in the present implementation of ELMA.

3.3 Object Creation

with-named-node (*node-name node-handle* &optional &key *node-type before-form init-list after-messages*) [macro]

Allocates a new object, or *node*, from the free pool. A new object is created only if the free pool is empty.

1. The name of the new node is bound to the *node-name* symbol.
2. The remote address, or *handle*, of the new node is bound to the *node-handle* symbol.
3. *node-type* is the type of the new node. The default is the type of the node of the first free pool in this manager's free-pools slot.
4. *before-form* is evaluated within the current context before allocating or creating the node.
5. *init-list* is the init-list of the new node, a list of alternating slot keywords and values (identical to that specified for LAMINA's creating function.)
6. *after-messages* are sent (by *mailing-together*) as soon as the new object has been created.

Example:

```
(defflavor foo-type
  (slot-1
   slot-2
   slot-3
   slot-4)
  (ELMA-subordinate)
  (:documentation "Example subordinate definition"))
```



```

(with-named-node
  (foo foo-handle
    :node-type
      'foo-type
    :before-form
    (progn
      (do-something)
      (do-something-else)
    :init-list
    (list
      :slot-1 slot-1-value
      :slot-2 slot-2-value
      :slot-3 slot-3-value)
    :after-messages
    (
      (foo
        (list (:message-for-foo some-args)))
      (name
        (list
          (:message-for-creator some-more-args)
          (:another-message-for-creator some-args)))
      (bazz
        (list (:message-for-bazz foo foo-handle)))
    )
    (some-function foo)
    (something-else-to-do-within-the-scope-of-foo))

```

The (with-named-node ...) of this example would exist within a method or function of a manager containing a free pool of *foo-types* (and possibly others). The forms (do-something) and (do-something-else) are evaluated first. An object is then allocated from the free pool of objects of type *foo-type* and the name and handle (remote address) of this object are bound to *foo* and *foo-handle* respectively. The slot-1, slot-2 and slot-3 slots of the new object named *foo* are initialized as specified and the slot-4 slot is made unbound. Finally when *foo* is created, it executes *:message-for-foo*, sends back to its manager (which is named *name* since *after-message* targets are bound within the scope of the manager) two messages, and sends to the object named *bazz* a *:message-for-bazz* message which takes *foo* and *foo-handle* as arguments. In the meantime, the creating manager is evaluating the forms (some-function ..) and (something-else-to-do-within-the-scopeof-foo).

```

with-subordinate (node-name &optional &key node-type before-form
  init-list after-messages) [macro]

```

with-subordinate is identical to *with-named-node* except that only the name of the new node is returned, not its handle.

In ELMA, the time for object "creation" is the time to send a message and receive an acknowledgement. Since a message is invariably sent to a new object anyway, it is usually a matter of just prepending the additional information necessary to reinitialize the object, rather than sending an additional message. The net cost of free-pool allocation is thus only slightly more than the cost of sending the original message which means that "creation" is very cheap. The caveat is that the ELMA free pool mechanism competes with application-level methods for the computational resources of the multiprocessor. Unless system resources are available on other sites for subordinate object allocation and on the site of the manager object for handling various handshake messages, no performance gain can be expected. The free pool mechanism effectively trades off available resources against object creation latency; providing the resources are available, object creation latency is reduced.

3.4 Object Deallocation

As for creation, object deallocation involves sending a `:deallocate-self` message to the subordinate object as well as updating the state of owning manager, specifically the free-pool object. The storage for one object can of course be allocated and re-allocated any number of times. ELMA uses *names* to distinguish between different incarnations and thus avoid mishandling of messages. For example, message disordering makes it possible for messages sent to an object to arrive after it has been deallocated, i.e., after it has received the `:deallocate-self` message. In fact, if the disordering is extremely bad, it is even possible for an incarnation of a subordinate object to receive a message intended for a previous incarnation, since LAMINA messages are sent to a given remote address and this does not change between incarnations. The ELMA programmer can specify how deallocated objects are to handle messages which are received out of order, the so called *incorrect address action*, on either an object type or object instance basis. Note that a subordinate can only be deallocated by its manager.

`deallocate names &optional incorrect-address-action` [function of elma-manager]

Deallocates free pool nodes named *names*, either a list of names or a single name. *incorrect-address-action* is a list of lists of the form

((action1 messages1 &optional forwarding-names1)
(action2 messages2 &optional forwarding-names2) ...)

which specifies the incorrect address action for the node being deallocated, i.e. what to do when a message is received by an object other than the intended one. Actions may be one or more of the following:

`:DOIT` Go ahead and execute the message even if it is out of order.
`:DROP` Ignore the message.
`:RETURN` Return the message to sender.
 The sender must have a `:returned-message` trigger.

:ERROR Signal a resumable error when messages arrive out of order
This is useful for debugging.

:FORWARD Forward messages to names.
(forwarding-names may be specified in this case)

trigger Execute *trigger* instead of messages.

function Execute the trigger and args returned by funcalling *function*.
function should take 3 args: target-name,
original-message and original-message-args.

A subordinate can request its own deallocation by invoking `request-deallocation`.

`request-deallocation` &optional *other-messages* [function of elma-subordinate]

`request-deallocation` sends a `:request-deallocation` message to the owning manager, which results in the subordinate being deallocated. *other-messages* is a list of messages for the manager to execute immediately after deallocating the subordinate. It is a list of message names and arguments in mailing-together format.

`set-incorrect-address-action` *action messages* &optional (*incarnation name*)
forwarding-names-and-handles [function of elma-subordinate]

`set-incorrect-address-action` defines the incorrect address action for messages targeted to the object named *incarnation*. *Messages* is a single message or list of messages. *Action* may be one or more of the following:

:DOIT Go ahead and execute the message even if it is out of order.

:DROP Ignore the message.

:RETURN Return the message to sender.
The sender must have a `:returned-message` trigger.

:ERROR Signal a resumable error when messages arrive out of order

:FORWARD Forward *messages* to *forwarding-names-and-handles*.
(*incarnation* and *forwarding-names-and-handles* may be specified,
or specified later by means of a `set-forwarding-addresses`)

trigger Execute *trigger* instead of messages

function Execute the trigger and args returned by funcalling *function*.
function should take 2 args: target-name original-message-args.

`set-forwarding-addresses` &rest *forwarding-names-and-handles*
[function of elma-subordinate]

`set-forwarding-addresses` sets forwarding addresses to the names in *forwarding-names-and-handles*. In other words, all messages for which `incorrect-address-action` is `:FORWARD` which are sent to the calling subordinate after it has been deallocated are forwarded to names.

`:set-forwarding-addresses` *incarnation forwarding-names-and-handles*
[trigger of elma-subordinate]

`:set-forwarding-addresses` sets forwarding addresses for *incarnation* to the names in *forwarding-names-and-handles*, an alternating list of names and handles. In other words, all messages bound for the object named *incarnation* for which `incorrect-address-action` is `:FORWARD` are forwarded to names. Whereas the function `set-forwarding-addresses` is invoked by the subordinate itself, the `:set-forwarding-addresses` message invoking this trigger is typically sent by the owning manager.

3.5 Message Sending

ELMA replaces the LAMINA message sending construct with the `mailing` construct which takes a destination *name* rather than a remote address. The sender is thus able to specify which particular *incarnation* of an object the message is intended for, and ELMA can take appropriate action at the receiving end if the name is incorrect.

`mailing` *target-names trigger value &rest lamina-keywords* [function]

`mailing` sends a *trigger* message with *value* to *target-names*. It is like LAMINA's sending except that *target names* are specified rather than their remote addresses. Remote addresses can be specified, but this makes it impossible to check whether or not the recipient is the intended target. Messages mailed to `oneself` are executed within the current context rather than being actually packaged and sent to the operator.

Sometimes it is convenient to group messages together to guarantee sequential execution of their methods. Of course it is always possible to define a single method which includes the messages in the group but this defeats the purpose of structured programming. Instead ELMA provides the `mailing-together` construct.

`mailing-together` *target-names messages-and-args &rest lamina-keywords* [function]

`mailing-together` is like `mailing`, but *messages-and-args* is list of the form `((message1 args1) (message2 args2) ...)` which are sent to targets in one packet and executed *serially* and *atomically* by the target. Messages mailed to `self` are executed within the current context rather than being actually packaged and sent to the operator.

Example:

```
(mailing-together foo (list (list :message-1 args-for-message-1)
                             (list :message-2 args-for-message-2))
                   :after 20)
```

This sends a packet containing `message-1` and `message-2` and their arguments to `foo` after a delay of 20ms. Upon receipt of the packet, `foo` executes `message-1` and then `message-2`, atomically. The `mailing-together` construct thus provides a means of guaranteeing sequential order of execution.

`sending-together` *targets messages-and-args &rest lamina-keywords* [function]

`sending-together` is like LAMINA's `sending`, but *messages-and-args* is list of the form `((message1 args1) (message2 args2) ...)` which are sent to targets in one packet and executed *serially and atomically* by the target. Messages mailed to self are executed within the current context rather than being actually packaged and sent to the operator. The use `mailing-together` is strongly recommended over `sending-together`.

`mailing-self` *trigger value &rest lamina-keywords* [function]

`mailing-self` sends *trigger* message with *value* to self. Instead of a real mailing, however, the method is simply executed within the present context.

As mentioned previously, each ELMA object maintains an *address table* of all objects it knows about. Apart from storing remote addresses, manager address tables store status data which is used to automatically defer messages to subordinates still being created. This is transparent to the programmer who can send messages to an object given its name without concern for the status of the object. For this reason it preferable to use `mailing` or `mailing-together` over `sending` or `sending-together`.

3.6 Meter Functions

ELMA includes numerous functions for metering the application. These functions are useful for both debugging and performance analysis.

3.6.1 Timing Functions

Triggers which are included in the `timed-triggers` slot of an ELMA object are timed automatically whenever `*elma-time*` is `t`. Functions and LISP methods, however, must be explicitly wrapped in a `timing` macro at each point in the program they are to be timed.

`timing` [macro]

`timing` times the execution of a function call or ZetaLisp send. Note that `timing` currently only wraps around one form.

Examples:

```
(timing  
  (send foo :bar (1 2 3)))  
;; times how long it takes foo to execute the :bar method
```

```
(timing  
  (bazz '(1 2 3)))  
;; times the function bazz
```

```
*trace-if-segment-takes-longer-than* [variable; default = 10000]
```

A trace message is printed if this is non nil and if a function or send takes longer to execute than this number of milliseconds.

```
:method-timing-data method [method]
```

:method-timing-data returns the timing data for *method*. namely the frequency of execution and total execution time. *method* can be a trigger (one of `timed-triggers`), function, or method name.

3.6.2 Recording Functions

ELMA automatically records incoming messages, outgoing messages, and measures dynamic (free pool) object recycling, message queue lengths and free pool lengths when the variables `*elma-record-history*`, `*elma-record-mailings*`, `*elma-count-recycling*`, `*elma-count-queues*`, and `*elma-count-free-pool-lengths*`, respectively are t. In addition, the following functions are useful for saving arbitrary information on the history list.

```
:record event &optional &rest other-info [method]
```

:record records when *event* occurred.

```
:record-msg message args [method]
```

:record-msg records when *message* was received.

```
increment-count action [inline function]
```

increment-count increments the count associated with the symbol *action* in the `count-data` slot. `count-data` is an association list ((`action-1 . count-1`) .. (`action-n . count-n`)) ELMA instance variable. It is also used by ELMA to store information such as object recycling

data. If *action* is not currently in the list, *increment-count* adds it and sets the associated count to 1.

3.7 Setup Function

`adjust-elma-parameters` &optional *batch* [function]

When *batch* is nil (the default), `adjust-elma-parameters` pops up a menu and prompts the user to set ELMA parameters, otherwise it simply makes dependent parameters consistent with the present parameter values.

<i>ELMA Parameter</i>	<i>Type</i>
<i>Description or ELMA action if flag is t</i>	
<code>*elma-memory-management*</code> Conses objects to a static area	Boolean
<code>*elma-time*</code> Times selected triggers, messages and functions (Specify actual triggers with <code>timed-triggers</code> slot)	Boolean
<code>*elma-count-queues*</code> Counts queue lengths	Boolean
<code>*elma-count-recycling*</code> Measures node recycling	Boolean
<code>*elma-count-free-pool-lengths*</code> Measures free pool lengths	Boolean
<code>*replace-upon-removal-from-free-pool*</code> Enables replacement of allocated free pool nodes (If nil, allocated nodes are only replaced when the pool is empty)	Boolean
<code>*elma-record-history*</code> Records history (of messages received and recorded events) (Specify actual triggers with <code>notable-triggers</code> slot)	Boolean
<code>*elma-record-mailings*</code> Records mailings	Boolean
<code>*elma-debug*</code> Prints diagnostic messages, i.e., <code>elma-formats</code>	Boolean
<code>*elma-history-time-span*</code> Time span of ELMA measurements	Number
<code>*elma-history-time-quantum*</code> Time quantum of ELMA measurements	Number
<code>*ELMA-simulation-to-domain-time-conversion-function*</code> Function to convert simulation time to domain time (This is only used by <code>time-lines</code> and need not be specified if <code>*elma-count-queues*</code> and <code>*elma-count-free-pool-lengths*</code> are nil)	Function

Note that domain time is the time of input data as specified by the data set. Simulation time is time according to which input data is put into the simulator.

3.8 Other Functions

ELMA also includes a variety of utility functions as well as some constructs which are refinements of those found in LAMINA.

3.8.1 Name Functions

`remote-address-of node-name` [function]

`remote-address-of` looks up *node-name* in the node's address-table and returns the corresponding remote address and status of the node.

`add-to-address-table &rest rest` [function]

Adds (alternating) node names and handles to the address table

`update-address-table &rest rest` [function]

Adds (alternating) node names and handles to the address table, unless they are already present.

`:add-to-address-table rest` [trigger]

Adds list of alternating node names and handles to the address table.

`:update-address-table rest` [trigger]

Adds list of alternating node names and handles to the address table. An error results if entries do not exist for all names.

`add-requests-to-address-table &rest rest` [function]

Adds (alternating) node names and handles of requested nodes to the address table

`creator-name name` [function]

Returns the name of the creator of the object named *name*.

`creator name` [function]

Returns the type and number of the creator of the object named *name*.

3.8.2 Stream Functions

`new-streams n` [function]

Returns a list of *n* new streams.

`with-multiple-messages (list-of-values group-stream &optional expected-message
(filter-function #'cons))` [macro]

This construct spawns a (continuation) process to wait for messages of type *expected-message* (specified just for checking purposes) on *group-stream*. It "adds" the values returned from each stream to *list-of-values* using the *filter-function*. The default is simply for all values to be consed onto *list-of-values*. Refer to *LAMINA: CARE Applications Interface* for a detailed explanation of the continuation mechanism.

3.8.3 Format Functions

`debug-format &rest rest` [inline function]

Like `format`, but executed within a `without-lock`, i.e., untimed by the simulator, and only if the global variable `CARE-USER::*debug*` is `t`.

`elma-format &rest rest` [inline function]

Like `format`, but executed within a `without-clock`, i.e., untimed by the simulator, and only if the global variable `CARE-USER::*elma-debug*` is `t`.

`untimed-format &rest rest` [macro]

Like `format`, but executed within a `without-clock`, i.e., untimed by the simulator.

3.8.4 Simulator Functions

`now` [macro]

Returns the current simulation time in milliseconds.

`care-sites` [function]

Returns a list of CARE sites for the current design.

`care-site-numbers &optional (grid-size (length care:***all-sites-vector***)` [function]

Returns a list of CARE site numbers for *grid-size*.

3.8.5 Arithmetic Functions

`average &rest averagends` [function]

Returns the average of *averagends*.

`average-if-numeric &rest averagends` [function]

Returns the average of all numeric *averagends*.

`max-if-numeric &rest rest` [function]

Returns the maximum of all numeric *rest*.

`min-if-numeric &rest rest` [function]

Returns the minimum of all numeric *rest*.

`ceiling-to-number dividend &optional (number 1) (divisor 1)` [function]

Like ceiling, but to the nearest multiple of *number*.

`round-to-number dividend &optional (number 1) (divisor 1)` [function]

Like round, but to the nearest multiple of *number*.

`floor-to-number dividend &optional (number 1) (divisor 1)` [function]

Like floor, but to the nearest multiple of *number*.

4 Programmer Provided Methods

4.1 Optional Methods

`:initialize` [method of application's allocator type]

When the programmer defines an *:initialize* method for the allocator type used by the application, i.e., *allocator-type* in the *start-ELMA initialization-parameters*, it is invoked prior to creation of the managers specified in *manager-initializations*. The value returned by this method is appended to the init list arguments explicitly provided in *manager-initializations* for each manager.

Example:

```
(defflavor my-allocator
  ()
  (ELMA-allocator)
  (:documentation "Allocator for my application"))

(defmethod (my-allocator :initialize)
  "This is a silly example which simply returns the CARE
  site numbers. In practice something more useful could
  go here."
  (care-site-numbers)
  )

(defun boss-init-list-fn (clerk-sites all-sites)
  "Returns init list for BOSS object"
  .....
  )

(start-ELMA :initialization-parameters
  '((:ALLOCATOR . "the-ALLOCATOR")
    (:ALLOCATOR-TYPE . ELMA-allocator)
    (:ALLOCATOR-SITE . 0)
    (:START-MESSAGE . :get-ball-rolling)
    (:START-MESSAGE-RECIPIENT . "the-BOSS")
    (:MANAGER-INITIALIZATIONS
     ("the-BOSS" ;; name of the manager
      BOSS ;; type of the manager
      0 ;; site of the manager
      boss-init-list-fn
      ;; function to generate init-list of manager
      (,clerk-sites) ;; args of function
      NIL))) )
```

The boss-init-list-function is called with the value returned by the :initialize method appended to (list clerk-sites). In other words, instead of evaluating (boss-init-list-fn clerk-sites), ELMA evaluates (boss-init-list-fn clerk-sites care-site-numbers) to get the init list of "the-BOSS."

4.2 Required Methods

:returned-message (*original-message sent-name new-name original-args*) [trigger]

Any node which could receive a returned message (due to :RETURN being specified for incorrect-address-action of a node to which it sends messages) should have a trigger as follows:

```
(deftrigger (some-node :returned-message)
  ((original-message sent-name new-name original-args))
  "Trigger for handling returned messages"
  ;; code for handling the returned message
  (debug-format t "%Returned-message: ~a ~a ~a ~s"
    original-message sent-name new-name original-args)
  )
```

5 Abstract Data Types

ELMA also includes a collection of abstract data types (ADTs).

5.1 Table

The `table` is a generic table flavor which can be arbitrarily large, of arbitrary dimensions and indexable on arbitrarily defined keys. A table can have multiple axes, where each axis corresponds to a dimension. Each axis can be of fixed or variable length. A variable length axis can be implemented as either an association list or a hash table (default is alist). The function `make-table` returns a table. Refer to Appendix B for examples.

```
make-table &key axes &optional label init-values [function]
```

5.2 Ring Buffer

The `ring-buffer` is a generic ring buffer flavor for the efficient maintenance of time-dependent data, i.e., for managing the addition of new data and timely removal of old, "out of date" data. Refer to `elma:structures;ring-buffer.lisp` for details.

5.3 Time Line

The `time-line` is a flavor for recording data over time. The history is discretized according to domain time. Refer to `elma:main;elma.lisp` for details.

6 ELMA Flavor and Specializations

This section lists the flavor definitions for the ELMA flavor and its specializations. These can be found in `elma:main;elma.lisp`. (Load `sys:site;elma.translations` to get the logical names)

6.1 ELMA Base Flavor

The ELMA flavor is the essential component of any node in an ELMA application. Substitute it wherever the LAMINA or ORDERED-SELF-STREAM flavor would have been used in a LAMINA application.

6.1.1 ELMA Components

```
(defflavor ELMA
  ()
  (reinitable-mixin
   multi-message-mixin
   history-mixin
   defer-mixin
   name-mixin
   ordered-self-stream)
  (:documentation "ELMA is the base flavor for
   memory-management LAMINA applications.")
)
```

Each component mixin contributes the following behavior.

- `reinitable-mixin` makes slots re-itable, which makes it possible to reuse free pool nodes.
- `history-mixin` maintains user-specifiable history of each node.
- `multi-message-mixin` handles CARE postings which contain multiple messages.
- `defer-mixin` permits postings to nodes still being created by deferring them temporarily.
- `name-mixin` gives node a unique name and maintains an address table permitting other nodes to be referenced by name.
- `ordered-self-stream` gives node essential lamina node behavior with incoming tasks ordered by an insertion sort.

6.1.2 ELMA Slots

The following are the most important slots of the ELMA flavor. The history and data slots accumulate information about each object, and can be inspected/read to debug and/or analyze the program.

`name` name of object

`stream` remote address of object

`notable-triggers` incoming trigger messages for which time, message name and message arguments are to be saved in the history (when `*elma-record-history*` is t).

`timed-triggers` trigger methods to be timed (when `*elma-time*` is t).

`initial-addresses` list of alternating names and remote addresses to be stored in the `address-table` when the object is created.

`output-stream` output-stream used for output.

`address-table` table of remote addresses that object knows about (indexed on name,

`history` list of triggers executed by the object and other user-specified events.

`mailing-history` list of messages mailed by this object (when `*elma-record-mailings*` is t).

`timing-data` data accumulated for timed-triggers and other timed methods and functions (when `*elma-time*` is t).

`count-data` data accumulated when using `increment-count` or for object recycling data (when `*elma-count-recycling*` is t).

`queue-data` time-line data accumulated (when `*elma-count-queues*` is t).

6.2 Acknowledge Creation Mixin

```
(defflavor acknowledge-creation-mixin
  (init-ack-stream)
  ;; node acknowledges on this stream when created
  ()
  :settable-instance-variables
  (:documentation "ELMA node posts an acknowledgement
    to its init-ack-stream when created.")
  (:required-flavors elma)
  )
```

6.3 Static Allocator Mixin

```
(defflavor static-allocator-mixin
  (manager-initializations
    creation-acks
    ;; the number of acknowledgements expected at initialization
    start-message
    ;; the name of the start message to be sent when all managers
    ;; have been created
    start-message-recipient
    ;; the recipient of the above start message
  )
  ()
  :settable-instance-variables
  (:documentation "For creating manager nodes at initialization time.")
  (:required-flavors elma)
)
```

6.4 Manager Mixins

```
(defflavor free-pool-allocator-mixin
  (free-pools) ;;list of free pools maintained by this node
  ()
  :settable-instance-variables
  (:required-flavors elma)
  (:documentation "Enables elma nodes to maintain and
    allocate free pool objects")
)
```

```
(defflavor free-pool-reclaimer-mixin
  ()
  ()
  (:documentation "Enables free pool allocators to deallocate free pool nodes.")
  (:required-flavors free-pool-allocator-mixin elma)
)
```

```
(defflavor free-pool-owner-mixin
  ()
  (free-pool-allocator-mixin
    free-pool-reclaimer-mixin)
  (:documentation "Enables elma nodes to allocate
```

```

        and deallocate free pool nodes.")
    (:required-flavors elma)
  )

```

6.5 Subordinate Mixins

```

(defflavor free-pool-allocatable-mixin
  (owner-stream)
  ;; created object acknowledges its creator on this stream
  ()
  :settable-instance-variables
  (:documentation "Makes elma nodes allocatable from a free pool")
  (:required-flavors elma)
)

```

```

(defflavor free-pool-reclaimable-mixin
  ((subordinate-status :CREATED)
   (incorrect-address-action nil)
   (incorrect-address-system-action '(:deallocate-self . :doit)
                                     (:re-init . :doit)))
  (cached-tasks)
  (forwarding-data nil)
  )
  ()
  (:documentation "Makes free pool nodes reclaimable")
  (:required-instance-variables owner-stream)
  (:required-flavors free-pool-allocatable-mixin elma)
)

```

```

(defflavor free-pool-node-mixin
  ()
  (free-pool-allocatable-mixin
   free-pool-reclaimable-mixin)
  (:documentation "Makes elma nodes allocatable and reclaimable
                  from a free pool")
  (:required-flavors elma)
)

```

6.6 Free-pool Flavor

```

(defflavor free-pool

```



```

((node-type)
 ;; the node in the free pool
 (initial-length *default-free-pool-length*)
 ;; the initial length of the free pool
 (threshold-length *default-free-pool-length*)
 (unallocated-nodes)
 (requested-nodes)
 (allocated-nodes)
 (deallocated-nodes)
 (count 0)
 (sites nil) ;; the sites available for free pool nodes
 (site-function 'random-fp-site)
 ;; name of function which returns a site number on which to
 ;; allocate a new node
 )
(copy-self-mixin)
:settable-instance-variables
(:documentation "The free-pools slot of a free-pool-allocator is a
list of these objects. Use make-free-pool rather
than make-instance."))

```

Acknowledgements

The author wishes to thank Chris Rogers who, as the first true ⁶ ELMA programmer, provided valuable feedback in the face of much flakiness and non-documentation, and still succeeded in implementing the largest CARE application ⁷ yet completed within the Knowledge Systems Laboratory at Stanford University.

The author also wishes to thank Schlumberger Technologies Laboratories for providing the resources with which this document was produced.

This work was supported by DARPA Contract F30602-85-C-0012 and NASA Ames Contract NCC 2-220-S1.

⁶Being a programmer, rather than an implementor/programmer

⁷The AIRTRAC Path Association system. Refer to: A.C. Noble and E.C. Rogers, *AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor*, KSL Report No. 88-41, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Appendix A Example Program

An example program can be found in `elma:examples;practicum example.lisp`. Type in the following to execute it:

1. Enter `(make-system 'CARE)` followed by `(make-system 'ELMA)` at a LISP-listener window.
2. Enter `(load "elma:examples;practicum-example")`
3. Enter `(elma-example :initialize t)`
(On subsequent runs just enter `(elma-example)`)

The listing is as follows:

```
;; -*- Mode:Common-Lisp;
   Package:CARE-USER;
   Fonts:(CPTFONT TR10B TR10I);
   Base:10 -*-

;; Author: Alan C. Noble

;; This file contains a complete but simple ELMA program,
;; including SIMPLE simulator interface.

(defvar *free-pool-init-length* 2)
(defvar *free-pool-threshold-length* 1)

;; *****
;;
;; PROGRAM EXPLANATION
;;
;; *** *****

;; Two user object types are defined: BOSS, an ELMA-manager,
;; and CLERK, an ELMA-subordinate. The (start-ELMA) call initializes
;; the program by creating an ELMA-allocator which in turn
;; creates a BOSS object. The ELMA-allocator automatically sends a
;; :get-ball-rolling message to the EJSS when it receives a creation
;; acknowledgement from the latter. The BOSS allocates three CLERKS,
;; Alfred, Oliver and Barry, and sends them an :apple, :orange
```

```

;; and :banana message respectively. The BOSS then deallocates Oliver
;; and Barry and then sends them various messages. Since they are
;; deallocated, the incorrect address mechanism handles these messages
;; and automatically returns, forwards or drops them according
;; to settings.

```

```

;; *****
;;
;; OBJECT DEFINITIONS
;;
;; ***** *****

```

```

;; BOSS

```

```

(defflavor boss
  ()
  (elma-manager)
  :settable-instance-variables
  (:documentation "Boss--does out work to the clerks")
)

```

```

;; CLERK

```

```

(defflavor clerk
  ()
  (elma-subordinate)
  (:documentation "Clerk--does all the work")
  (:default '-init-plist
    ;; record incoming messages for these triggers
    :notable-triggers '(:apple :orange :banana)
    ;; time the execution of these trigger methods
    :timed-triggers '(:apple :orange :banana)
    :incorrect-address-action
    '(;; default settings for CLERK instances
      (:apple . :RETURN) ;; i.e., return all apple messages to sender
      (:orange . :RETURN)
      ;; i.e., return all banana messages to sender
      ;; :banana not specified, so dropped by default
    )
  )
)

```

```

;; *****
;;
;; TRIGGER METHOD DEFINITIONS
;;
;; *****

;; BOSS

(defun boss-init-list-fn (clerk-sites)
  "Returns an in' list for a boss object, in this case the :free-pools
slot and its val
  (list :free-pools
        (list
         (make-free-pool
          :node-type 'CLERK ;; each boss has free pool of clerks
          :initial-length *free-pool-init-length*
          :threshold-length *free-pool-threshold-length*
          :sites clerk-sites))
         )
        )
)

(defun trigger (boss :get-ball-rolling) ()
  "This trigger gets the ball rolling."
  ;; Allocate three clerks from the pool
  (with-subordinate (alfred)
    (with-subordinate (oliver
                       :after-messages
                       ((oliver '(:pineapple))))
      ;; i.e., send :pineapple message to oliver as
      ;; soon as it is allocated
      )
    (with-subordinate (barry)
      ;; get clerks working
      (mailing alfred :apple nil)
      (mailing oliver :orange nil)
      (mailing barry :banana nil)
      ;; deallocate oliver and barry
      (deallocate oliver )
      (deallocate barry
       :incorrect-address-action
       '(:FORWARD :orange (,alfred)))
      ;; override the default so :orange is forwarded
      )
      ;; send oliver and barry some work and see what happens
    )
  )
)

```

```

(mailing oliver :apple nil) ;; this should be returned to the boss
(mailing barry :apple nil) ;; this should also be returned
(mailing barry :orange nil) ;; this should get forwarded to alfred
(mailing barry :banana nil) ;; this should get dropped
)
)
)
)

(deftrigger (BOSS :returned-message)
  ((original-message sent-name new-name original-args))
  "Trigger for handling returned message"
  (debug-format output-stream "%=> ~a received a RETURNED-MESSAGE" name)
  (debug-format output-stream
    "% Message ~a was sent to ~a, now named ~a, with args ~s"
    original-message sent-name new-name original-args)
  )

;; CLERK

(deftrigger (CLERK :apple) ()
  (untimed-format output-stream "%~a got an APPLE" name)
  )

(deftrigger (CLERK :orange) ()
  (untimed-format output-stream "%~a got an ORANGE" name)
  )

(deftrigger (CLERK :banana) ()
  (untimed-format output-stream "%~a got a BANANA" name)
  )

(deftrigger (CLERK :pineapple) ()
  (untimed-format output-stream "%~a got a PINEAPPLE" Name)
  )

;; *****
;;
;; SIMULATOR INTERFACE
;;
;; *****

```

```
;; The rest of this file defines functions which set up the simulator
;; and start a simulation.
;; The top level function is start-ELMA.
```

```
(defun elma-example (&optional &key initialize
                    (design 'care:octorus-32)
                    (instrument 'care:observer))

  "Run elma example program.
  Initialize simulator if initialize is t."
  (when initialize ;; initialize the simulator
    (simple :run nil
           :reset t
           :flush t
           :design design
           :instrument instrument))
  (let ((clerk-sites (cl:remove 0 (care-site-numbers))))
    ;; i.e., use all sites except boss site
    (start-ELMA
     :initialization-parameters
     '( (:ALLOCATOR . "the-ALLOCATOR")
        (:ALLOCATOR-TYPE . ELMA-allocator)
        (:ALLOCATOR-SITE . 0)
        (:START-MESSAGE . :get-ball-rolling)
        (:START-MESSAGE-RECIPIENT . "the-BOSS")
        (:MANAGER-INITIALIZATIONS
         ("the-BOSS" ;; name of the manager
          BOSS ;; type of the manager
          0 ;; site of the manager
          boss-init-list-fn
          ;; function to generate init-list of manager
          (,clerk-sites) ;; args of function
          NIL))
       )
     )
  )
)
```

Appendix B

Table Examples

Example 1: 2D Table, with initial values

```
(setq foo
  (make-table
    :label "handles"
    :axes (list '(:name :aircraft-type
                  :type :variable
                  :keys (:TYPE-A :TYPE-B))
               '(:name :radar-id
                  :type :variable
                  :keys (:RADAR-1 :RADAR-2 :RADAR-3))
            )
    :init-values '((handle-A-1 handle-A-2 handle-A-3)
                  (handle-B-1 handle-B-2 handle-B-3)))
)
```

;; OR, alternative specification using a linear list of values
;; (but get the order right!)

```
(setq foo
  (make-table
    :label "handles"
    :axes (list (list :name :aircraft-type
                      :type :fixed
                      :keys '(:TYPE-A :TYPE-B))
                (list :name :radar-id
                      :type :fixed
                      :keys '(:RADAR-1 :RADAR-2 :RADAR-3))
            )
    :init-values '(handle-A-1 handle-A-2 handle-A-3
                  handle-B-1 handle-B-2 handle-B-3)
  )
)
```

;; Referencing values:

```
(send foo :value :TYPE-A :RADAR-2) ==> handle-A-2
```

;; Setting values:

```
(send foo :set 'new-handle-A-2 :TYPE-A :RADAR-2) ==> new-handle-A-2
```

```
(send foo :value :TYPE-A :RADAR-2) ==> new-handle-A-2
```

;; Removing values

```

(send foo :remove :TYPE-A :RADAR-2) ==> nil
(send foo :value :TYPE-A :RADAR-2) ==> nil
;; Adding values
(send foo :add 'another-handle-A-2 :TYPE-A :RADAR-2) ==> 'another-handle-A-2
(send foo :value :TYPE-A :RADAR-2) ==> 'another-handle-A-2
;; Returning and removing values (only for variable type tables):
(send foo :remove-and-return :TYPE-A :RADAR-2) ==> 'another-handle-A-2
(send foo :value :TYPE-A :RADAR-2) ==> nil
;; Axis information:
(send foo :nth-axis 0) ==> :aircraft-type
(send foo :nth-axis 1) ==> :radar-id

```

Example 2: 1D Table, no initial values, used as a LIFO queue

```

(setq foo
  (make-table
    :label "Names"
    :axes '((:name :track-id
             :type :variable)))
)

(send foo :add 'name-0 0) ==> name-0
(send foo :add 'name-1 1) ==> name-1
(send foo :add 'name-2 2) ==> name-2
(send foo :add 'name-3 3)
(send foo :pop) ==> name-3
(send foo :pop) ==> name-2
(send foo :remove-and-return 0) ==> name-0

```

Example 3: 3D table, with initial values:

```

(setq foo
  (make-table
    :label "Test things"
    :axes (list (list :name :aircraft-type
                     :type :variable
                     :keys '(:TYPE-A :TYPE-B :TYPE-C))
               (list :name :radar-id
                     :type :variable
                     :keys '(:RADAR-1 :RADAR-2 :RADAR-3))
               (list :name :handle-or-handle
                     :type :variable

```



```

        :keys '(:name :handle))
    )
:init-values '(((name-A-1 handle-A-1)
                (name-A-2 handle-A-2)
                (name-A-3 handle-A-2))
              ((name-B-1 handle-B-1)
                (name-B-2 handle-B-2)
                (name-B-3 handle-B-3))
              ((name-C-1 handle-C-1)
                (name-C-2 handle-C-2)
                (name-C-3 handle-C-1)))
)
)

;; Adding values
(send foo :add 'handle-D-1 :TYPE-D :RADAR-1 :handle) ==> 'handle-D-1
(send foo :add 'name-D-2 :TYPE-D :RADAR-2 :name) ==> 'name-D-2
(send foo :remove :TYPE-D :RADAR-2 :name) ==> 'name-D-2
(send foo :value :TYPE-D :RADAR-1 :handle) ==> 'handle-D-1

;; Referencing major axis values
(send foo :major-axis-values) ==> '(:type-D :type-A :type-B :type-C)
(send foo :entries) ==> 19
;; Removing all table values for one major axis value (only for :VARIABLE type
tables)
(send foo :major-axis-remove :TYPE-B)
(send foo :major-axis-values) ==> '(:type-D :type-A :type-C)
(send foo :entries) ==> 13

```

Example 4: Variable Type Table with only some axis values specified

```

(setq foo
  (make-table
    :label "FPS stuff"
    :axes (list '(:name :fps-name
                  :type :FIXED
                  :keys (fps-0 fps-1))
                '(:name :fps-attribute
                  :type :FIXED
                  :keys (:time :fpo-parents :postion :velocity)))
  )
)

```

```

;; add some stuff for fps-0
(send foo :add 'time-0 'fps-0 :time)
(send foo :add 'fpo-parents-0 'fps-0 :fpo-parents)
(send foo :add 'fpo-position-0 'fps-0 :postion)

;; add some stuff for fps-1
(send foo :add 'time-1 'fps-1 :time)
(send foo :add 'velocity-1 'fps-1 :velocity)

;; Reference values:
(send foo :value 'fps-0 :time) ==> time-0
(send foo :value 'fps-1 :velocity) ==> 'velocity-1
;; Get major axis values
(send foo :major-axis-values) ==> (FPS-0 FPS-1)

```

Example 5: 2D Table

```

(setq foo
  (make-table
    :label "More stuff"
    :axes (list '(:name :aircraft-type
                  :type :FIXED
                  :keys (:TYPE-A :TYPE-B :TYPE-C :TYPE-D))
               '(:name :radar-id
                  :type :VARIABLE))
  )
)

(send foo :add 'A-1 :TYPE-A :RADAR-1)
(send foo :add 'A-2 :TYPE-A :RADAR-2)
(send foo :add 'B-3 :TYPE-B :RADAR-3)
(send foo :value :TYPE-B :RADAR-3)
;; or, referencing with numeric indices (only for fixed axes)
(send foo :value '(:nth 0) :radar-1) ==> A-1
(send foo :value '(:nth 1) :radar-1) ==> nil
(send foo :value '(:nth 1) :radar-3) ==> B-3
(send foo :value :TYPE-E :RADAR-1) ==> error
(send foo :value '(:nth 1) '(:nth 0)) ==> nil

```

Example 6: 1D Table, with dictionary axis

```

(setq foo

```

```

(make-table
  :label "name"
  :axes '(:name :fps-name
          :type :dictionary
          :keys ("FPS-0" "FPS-1" "FPS-2"))
  :init-values '(FPS-0 FPS-1 FPS-2)
)
)

```

```

(send foo :add 'FPS-3 "FPS-3")
(send foo :add 'FPS-5 "FPS-5")
(send foo :remove "FPS-0")
(send foo :remove-and-return "FPS-3") ==> FPS-3
(send foo :entries)
(send foo :pop-end)

```

Example 7: 3D Table, all axes fixed, without initial values

```

(setq table
  (make-table
    :label "FPC names and handles"
    :axes (list '(:name :aircraft-type :type :FIXED
                  :keys (:TYPE-A :TYPE-B))
                '(:name :radar-id :type :FIXED
                  :keys (:RADAR-1 :RADAR-2 :RADAR-3))
                '(:name :name-or-handle :type :FIXED
                  :keys (:name :handle)))
    )
  )
)

```

```

;; using numeric keys
(send table :add 'A-2-handle '(:nth 0) '(:nth 1) '(:nth 1))
;; above is equivalent to (send table :add 'A-2-handle :TYPE-A 2 :handle)
(send table :add 'A-2-name '(:nth 0) '(:nth 1) :name)
(send table :add 'A-1-handle '(:nth 0) '(:nth 0) :handle)
(send table :remove '(:nth 0) '(:nth 1) :handle)
(send table :remove '(:nth 0) '(:nth 0) :handle)
(send table :nth-axis-mth-key 2 1) ==> :handle
(send foo :valid-keys) ==> (:TYPE-A :TYPE-B)

```

Parallel Execution of OPS5 in QLISP

by

Hiroshi G. Okuno and Anoop Gupta**

*

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

and

Electrical Communications Laboratories
Nippon Telegraph and Telephone Corporation
3-9-11 Midori-cho, Musashino
Tokyo 180 Japan

**

CENTER FOR INTEGRATED SYSTEMS
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

Production systems (or rule-based systems) are widely used for the development of expert systems. To speed up the execution of production systems, a number of different approaches are being taken, a majority of them being based on the use of parallelism. In this paper, we explore the issues involved in the parallel implementation of OPS5 (a widely used production-system language) in QLISP (a parallel dialect of Lisp proposed by John McCarthy and Richard Gabriel). This paper shows that QLISP can easily encode most sources of parallelism in OPS5 that have been previously discussed in literature. This is significant because the OPS5 interpreter is the first large program to be encoded in QLISP, and as a result, this is the first practical demonstration of the expressive power of QLISP. The paper also lists the most commonly used QLISP constructs in the parallel implementation (and the contexts in which they are used), which serve as a hint to the QLISP implementor about what to optimize. We also discuss the exploitation of speculative parallelism in RHS-evaluation for OPS5. This has not been previously discussed in the literature.

Parallel Execution of OPS5 in QLISP

Abstract

Production systems (or rule-based systems) are widely used for the development of expert systems. To speed-up the execution of production systems, a number of different approaches are being taken, a majority of them being based on the use of parallelism. In this paper, we explore the issues involved in the parallel implementation of OPS5 (a widely used production-system language) in QLISP (a parallel dialect of Lisp proposed by John McCarthy and Richard Gabriel). This paper shows that QLISP can easily encode most sources of parallelism in OPS5 that have been previously discussed in literature. This is significant because the OPS5 interpreter is the first large program to be encoded in QLISP, and as a result, this is the first practical demonstration of the expressive power of QLISP. The paper also lists the most commonly used QLISP constructs in the parallel implementation (and the contexts in which they are used), which serve as a hint to the QLISP implementor about what to optimize. We also discuss the exploitation of speculative parallelism in RHS-evaluation for OPS5. This has not been previously discussed in the literature.

1. Introduction

There are several different programming paradigms that are currently popular in Artificial Intelligence, examples being production systems (or rule-based systems), frame-based systems, semantic-network systems, logic-based systems, blackboard systems. Of the above, production systems have been widely used to build large expert systems [10, 14]. Unfortunately, production systems run quite slowly, and this has especially been a problem for applications in the real-time domain. Production systems must be speeded-up significantly if they are to be used in new increasingly complex and time-critical domains. In this paper, we focus our attention on a specific production-system language, OPS5, that has been widely used to build expert systems and whose performance characteristics have been extensively studied. We also focus on parallelism as a means to speed-up the execution of OPS5.

The parallel execution of the OPS5 production-system language has been studied by several groups [4, 8, 11, 13]. Their general approach consisted of two steps. (i) the design of a dedicated parallel machine suitable for execution of OPS5; and (ii) the mapping of the OPS5 compiler and run-time environment on to the parallel hardware. In these implementations, the second step (the mapping step) involves parallel encoding of OPS5 using hardware specific and operating-system specific structures. In this paper, we explore how this mapping step may be done in a high-level parallel dialect of Lisp, called QLISP. The main advantages of encoding using a high-level programming language are: (i) Increase in portability, since the code does not depend on machine specific features; (ii) Greater flexibility and expressive power of the high-level language results in faster turn-around time, fewer errors, and more readable and modifiable code. The main disadvantage, of course, is that the encoding may not be as efficient as hand-coded hardware-specific encodings. We normally do not worry about such issues for uniprocessors -- language compilers for uniprocessors are good enough -- but the disadvantage is significant for parallel implementations where the technology is not as far advanced. There is one more strong motivation for doing a parallel implementation of OPS5 while remaining within Lisp (unlike most previous parallel implementations). This is that OPS5 is often used as an embedded system within larger AI systems, and the fact that the rest of these systems are encoded in Lisp. If OPS5 is also encoded in Lisp, then it makes the task of interfacing much simpler.

There are several parallel Lisp languages, for example, Multilisp [5, 6, 7] and QLISP [3], that are available for speeding up Lisp programs by using multiple processors. Since QLISP is based on the Common Lisp [12], it provides very powerful facilities to the user. Multilisp is based on a functional programming subset of Lisp.

Another distinguishing features of QLISP is that control mechanisms to access shared data or global data are embedded in Lisp primitives. Other parallel Lisp languages use some data structures for locking, such as semaphores. QLISP enables the user to write parallel programs without paying much attention to the consistency of shared or global data. One of the main purposes of this research is to explore the expressive power of QLISP by implementing a large program in it. Ours is the first large ("real") program implemented in QLISP, so this constitutes the first practical demonstration of the expressive power of QLISP. We also list the most commonly used QLISP constructs and the contexts in which they are used, which can serve as a guide for optimizing the implementation of the QLISP language. A language where it is easy to express parallel constructs, but which does not offer better performance is not of much use.

The approach we take for parallelizing OPS5 is based on that of the Production System Machine (PSM) project at Carnegie-Mellon University [4]. The PSM project studied how the speed-up from parallelism increases as one goes from coarse-granularity (rule-level) to fine-granularity (intra-node) parallelism. We implement each of their schemes and show that it is relatively easy to encode these parallel schemes within QLISP. We also show some interesting ways in which to exploit conflict-resolution parallelism and speculative parallelism¹ in RHS evaluation using QLISP.

This paper is organized as follows. Section 2 presents some background information about the OPS5 language, the Rete algorithm used to implement OPS5, and about QLISP. Section 3 describes how we do a parallel implementation of OPS5 using QLISP and the various issues involved. Finally, Section 4 is devoted to a discussion and conclusions.

2. Background

2.1. The OPS5 Production-System Language

An OPS5 [1] production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of assertions called the *working memory*. The assertions in the working memory are called *working memory elements*. Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left-hand side* of the production), and a set of *actions* corresponding to the *then* part of the rule (also called the *right-hand side* of the production). The left-hand side and the right-hand side are separated by the "-->" symbol. The actions associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-1 shows two simple productions named *p1* (with three condition elements) and *p2* (with two condition elements).

```
(p p1 (C1 ^color <x> ^size 12)      (p p2 (C2 ^price 38 ^color <y>)
      (C2 ^price 38 ^color <x>)      (C4 ^color <y>)
      (C3 ^color <x>)                -->
      -->                             (modify 1 ^price 50) )
      (remove 2) )
```

Figure 2-1: Example of productions

The production system *interpreter* is the underlying mechanism that determines the set of satisfied productions and controls the execution of the production system program. The interpreter executes a production system program

¹The parallel computations of a program can be divided into two categories, *mandatory computations* and *speculative computations* [7]. The former means that all computations executed in parallel are necessary, while the latter means that some computations executed in parallel may not be necessary.

by performing the following *recognize-act* cycle:

- **Match:** In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a *conflict set* is obtained, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- **Conflict-Resolution:** In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.
- **Act:** In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

Each working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator \wedge . The values are symbolic or numeric constants. Each conditional element in the LHS consists of a class name and one or more terms. Each term consists of an attribute prefixed by \wedge , an operator, and a value. An operator is optional and its default value is $=$. Other operators are $<$, $<=$, $>$, $>=$, $<>$ and $<=>$. A value is either a constant or a variable. A variable is represented by an identifier enclosed by $<$ and $>$. A variable can match any value, but all occurrences of the same variable in the LHS of a rule should match the same value. Conditional elements may not contain all pairs of attribute-value present in a working memory element. If a conditional element is preceded by $-$, it is called a negated condition element. The match for a rule succeeds only if there is no working memory element matching its negated condition element.

The RHS of a production can contain any number of actions. Actions can be classified into:

- **Working memory operations:** These are make, remove, and modify.
- **I/O operations:** These are openfile, closefile, and write.
- **Binding operations:** These are bind and cbind.
- **Miscellaneous operations:** These are default, call, halt, and build.

The above action types often take functions as arguments. Some such functions are $'$ (quote), substr, genatom, compute, litval, accept and acceptline.

2.2. The Rete Match Algorithm

Empirical study of various OPSS programs shows two interesting characteristics; *temporal redundancy* and *structural similarity* [2]. Temporal redundancy refers to the fact that a rule-firing makes only a few modifications to the working memory and most working-memory elements remain unchanged. Structural similarity refers to the fact that all productions are not totally distinct, and that there are many similarities between the condition elements of different productions. The Rete match algorithm exploits these two features to speed up the match phase of the interpreter.

The Rete algorithm uses a special kind of data-flow network compiled from the left-hand sides of productions to perform match. The network is generated at compile time, before the production systems is actually run. Figure 2-2 shows such a network for the two productions shown in Figure 2-1. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top-node down along these paths. The nodes with a single predecessor (near the top of the figure) are the ones that are concerned with individual condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left-hand sides require identical nodes, the algorithm shares part of the network rather than building duplicate nodes.

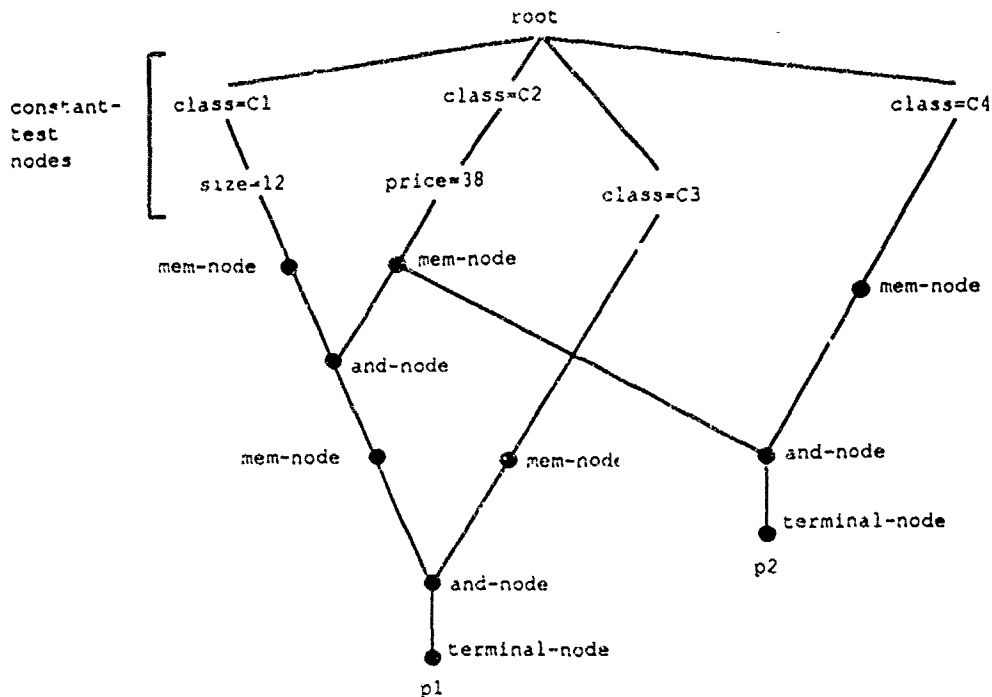


Figure 2-2: The Rete network

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as state within the nodes. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network updating the state stored within the network. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes are called *tokens*, which consist of a *tag* and an *ordered list of working-memory elements*. The tag can be either a +, indicating that something has been added to the working memory, or a -, indicating that something has been removed from it. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. These are:

1. **Constant-test nodes:** These nodes are used to test if the attributes in the condition element which have a constant value are satisfied. These nodes always appear in the top part of the network. They have only one input, and as a result, they are sometimes called *one-input nodes*.
2. **Memory nodes:** These nodes store the results of the match phase from previous cycles as state within them. The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. For example, the right-most memory node in Figure 2-2 stores all tokens matching the second condition-element of production p2.

At a more detailed level, there are two types of memory nodes -- the α -mem nodes and the β -mem nodes. The α -mem nodes store tokens that match individual condition elements. Thus all memory nodes immediately below constant-test nodes are α -mem nodes. The β -mem nodes store tokens that match a sequence of condition elements in the left-hand side of a production. Thus all memory nodes immediately below two-input nodes are β -mem nodes.

3. **Two-input nodes:** These nodes test for joint satisfaction of condition elements in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives on the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives on the right input of a two-input node.

There are also two types of two-input nodes -- the *and-nodes* and the *not-nodes*. While the *and-nodes* are responsible for the positive condition elements and behave in the way described above, the *not-nodes* are responsible for the negated condition elements and behave in an opposite manner. The *not-nodes* generate a successor token only if there are no matching tokens in the memory node corresponding to the negated condition element.

4. **Terminal nodes:** There is one such node associated with each production in the program, as can be seen at bottom of Figure 2-2. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

2.3. QLISP - Parallel Lisp Language

QLISP is a queue-based parallel Lisp proposed by Dick Gabriel and John McCarthy [3] and is being implemented on an Alliant FX/8 shared-memory multiprocessor by Stanford University and Lucid Inc. QLISP is similar to MultiLisp [5, 6, 7], but language constructs incorporate important mechanisms for parallel computation such as spawning and locking. The spawned processes are put in the system queue and given to a processor by the scheduler to evaluate it. The key ideas in QLISP were derived by reexamining Common Lisp [12] from the perspective of parallel processing, and by striving to make the minimal number of extensions to Common Lisp. Some QLISP primitives are summarized in the following subsections.

2.3.1. QLET

The *qlet* form executes its local binding in parallel.

```
(qlet predicate ((var value)*) (form)*)
```

The *qlet* form is a construct to evaluate all *values* in parallel². However, its computational semantic depends on the result of *predicate* which is evaluated first in the *qlet* form.

- If the result of *predicate* is nil, the *qlet* form acts exactly as the *let* form.
- If the result of *predicate* is neither nil nor eager, a process for each *value* is spawned and the process evaluating a *qlet* form is suspended. When all the results of *value* are available, each result is bound to each *var* and the process evaluating a *qlet* form resumes its computation; that is, the body of a *qlet* form is evaluated.
- If the result of *predicate* is eager, a special value, *future*³, is bound to each *var* and the body of a *qlet* form is evaluated immediately. A *future* is associated with a process which evaluates a *value* eventually. In the execution of the body, if the value is not supplied yet, the process executing the body is suspended till the value is available.

Two kinds of parallel fibonacci functions are shown in Fig. 2-3.

The first one calculates a fibonacci number by spawning a process to calculate every fibonacci number of a smaller number. There may occur a combinatorial explosion of processes if *n* is a large number. For example, the number of spawned processes is 176, 21890 and 242784 for *n* = 10, 20 and 25, respectively. The second fibonacci function spawns a process only if the depth of the nesting is less than the value of **cut-off**. The *qlet* predicate

²Since the *pcall* form in MultiLisp evaluates arguments of a function in parallel, it will be easily implemented by *qlet* in QLISP

³The mechanism of *eager* is an implicit implementation of the *future* form in MultiLisp, or the *lazy evaluation*.

```

(defun fib (n)
  (cond ((< n 2) 1)
        (t (qlet t ((f1 (fib (- n 1)))
                    (f2 (fib (- n 2))) )
                (+ f1 f2) ))))

(defun fib-c (n)
  (labels ((fib-cutoff (n depth)
            (declare (special *cut-off*))
            (cond ((< n 2) 1)
                  (t (qlet (< depth *cut-off*)
                          ((f1 (fib-cutoff (- n 1)) (1+ depth))
                           (f2 (fib-cutoff (- n 2)) (1+ depth)) )
                          (+ f1 f2) )))))
    (fib-cutoff n 0) ))

```

Figure 2-3: Two parallel Fibonacci functions - Example of qlet

enable the user to control the spawning of processes. Needless to say, an appropriate value for **cut-off** should be determined by the tradeoff between the cost and benefit of spawning.

2.3.2. QLAMBDA

The *lambda* form in the Common Lisp creates a closure which is used to share variables among several functions or as an anonymous function. The *qlambda* form creates a *process closure*.

(qlambda predicate lambda-list {form})*

A process closure is used not only to share variables among several process closures but also to control an exclusive invocation of the same process closure. That is, only one application of a process closure is evaluated and other applications of the same process closure are suspended. The evaluation of a process closure depends on the value of *predicate* which is evaluated at the time of evaluation of the *qlambda* form, that is, creation of a process closure.

- If the result of *predicate* is *nil*, the *qlambda* form acts exactly as the *lambda* form. That is, a lexical closure is created.
- If the result of *predicate* is neither *nil* nor *eager*, a process closure is created. When it is applied with arguments, a separate process is spawned for evaluation. If more than one applications occur, only one applications are evaluated and others are blocked. This is an implicit locking mechanism.
- If the result of *predicate* is *eager*, a process closure is created and spawned immediately without waiting for any arguments.⁴

A process closure may be used as an anonymous process, of which application is evaluated as a separated process. The *spawn* form is a shorthand form to do it; that is,

(spawn {form})* is the same as *((qlambda t () {form}*) .)*

In a sequential construct such as *block*, all forms may be evaluated in parallel by *spawn*. A set of functions to update of the conflict-set is shown in Fig. 2-4. The global variable **conflict-set-lock** holds a *qlambda* closure to control the exclusive access to the variable **conflict-set** which holds the list of production instances. The idea to provide an exclusive access to **conflict-set** is to execute an update operation by using the

⁴This curious mechanism can be used to write a parallel Y operator, that is, for all *f*, $Y(f)=f(Y(f))$, in QLISP. However, other useful applications are not yet known.

same `qlambda` closure. The lock is released when `register-cs` returns a value immediately or when `sort-conflict-set` updates the `*conflict-set*` or executes a sorting by spawning a subprocess by `qlet` with the predicate `eager`.

```
(proclaim (special *conflict-set-lock* *conflict-set*))

(defun ops-init ()
  (setq *conflict-set-lock*
        (qlambda t (body) (apply (car body) (cdr body))) ))

(defun insertcs (name data rating)
  (funcall *conflict-set-lock*
           (list 'register-cs
                 name data (cons (sort-time-tag data) rating) t )))

(defun removecs (name data rating)
  (funcall *conflict-set-lock*
           (list 'register-cs
                 name data (cons (sort-time-tag data) rating) nil )))

(defun register-cs (name data key flag)
  (cond ((null *conflict-set*)
         (setq *conflict-set*
               (create-new-cs-element key nil name data flag) ))
        (t (sort-conflict-set name data key flag *conflict-set*) )) )
```

Figure 2-4: Locking for Conflict-set

2.3.3. CATCH and THROW

A pair of `catch` and `throw` provides a way to do a non-local exit in the Common Lisp.

```
(catch tag form) and (throw tag value)
```

In QLISP, it provides not only a means of non-local exit but also a mechanism to control subprocesses spawned during the evaluation of `form` in the `catch` form. If the `catch` gets a value by the normal termination of `form` or a throwing, the `catch` kills all processed spawned during the execution of the `form`. If the value contains a future, the associated processes are not killed. Note that the execution of a process spawned at a value-ignoring position of a sequential construct is aborted.

2.3.4. QCATCH

The `qcatch` form is similar to the `catch` form, but the control of spawned processes is different.

```
(qcatch tag form)
```

If the evaluation of the `form` terminates normally and the `qcatch` gets a value, the `qcatch` waits for all the processes spawned during the execution of the `form` to terminate. Therefore, processes spawned at a value-ignoring position will be evaluated before terminating the `qcatch` form. If the execution of the `form` is aborted by a throwing, the `qcatch` kills all spawned processes beneath it.

2.3.5. UNWIND-PROTECT

The `unwind-protect` form is useful to do some cleanup jobs no matter what the `unwind-protect` form is terminated.

```
(unwind-protect protected-form {cleanup-form} *)
```

The `unwind-protect` form is very important in QLISP world in order to make the data consistent, because processes can be killed by the `catch` even if no throwing occurs.

2.3.6. Others

The *suspend-process* and *resume-process* forms are used for the user to control the scheduling of processes. The *wait* and *no-wait* are used to control the termination of a process spawned at a value-ignoring position of sequential constructs.

3. Parallel execution of OPS5 programs

As stated in Section 2.1, the OPS5 interpreter repeatedly executes a match -- conflict-resolution -- act cycle. In this section, we discuss how parallelism may be exploited in executing each of the three phases. Most of the discussion focuses on the match phase, as the match phase takes 90% of the time in the interpreter.

3.1. Parallelism in Match Phase

In this section, we explore how parallelism may be exploited to speed up the match phase. We present several different algorithms. We start with a coarse-granularity algorithm and slowly move towards finer granularity. In particular, we explore parallelism at three levels of granularity -- rule-level parallelism, node-level parallelism, and intra-node parallelism. All of the above algorithms are based on the Rete algorithm described in Section 2.2. What changes from one parallel algorithm to the other is the kinds of node activations that are allowed to be processed in parallel. The granularities we choose to discuss here correspond to those discussed in [4].

Before exploring the above schemes further, a word about the different kinds of node activations in the Rete network. Activations of constant-test nodes (shown in top-part of network in Figure 2-2) require just a simple test and are fairly cheap to execute. We call these *ctest* activations. It is usually not worth it to spawn a process to execute an individual *ctest* activation, because the overhead of spawning is larger than the work saved.

The second kind of node activations are the *memory-node* activations. These require that a token be added or deleted from the memory node, and can be expensive because a delete request may require searching through all the tokens stored in that memory node. The third kind are the *two-input node* activations, that require searching through the opposite memory-node to find all matching tokens (tokens with consistent variable bindings). These are also fairly expensive. We normally lump the processing required by the two-input node and the associated memory nodes together into a single task/process, because the two are closely interrelated (the two-input activation examines the memory node) and separating them incurs a large synchronization overhead. One also has to be careful about the sequence in which the above node activations are executed. For example, the Rete algorithm sometimes generates *conjugate* tokens, where exactly the same token is first scheduled to be added to the memory node and later deleted. The final result should be that the state of the memory node remains unchanged. However, in parallel implementations it is easily possible that the scheduler decides to pick the delete request before the add request, and if not handled properly, the final state of the memory node may have an extra token. To process conjugate pairs correctly, each memory node has an *extra-deletes-list* to store a deleted token whose target token has not arrived yet.

Finally, there are *terminal-node* activations that insert or delete instantiations/tokens into the conflict-set. Here also the problem of conjugate tokens can occur. The details for terminal-node activations are discussed later in Section 3.2.

For all the parallel implementation discussed in this paper, we use a common strategy for handling the *ctest* activations. (We present this strategy here, before discussing the differing strategies for the remaining types of activations.) This strategy is that multiple activations of the root node are processed using separate processes (i.e., activations corresponding to different changes to working memory are processed in parallel). However, all successors of the root node or the *ctest* nodes are evaluated using the following rule. If the successor node is also a *ctest* node then evaluate it sequentially within the same process, otherwise fork a separate process to do the

evaluation. The code for such an evaluation policy is shown in Figure 3-1.

```
(defun match (token root-node)
  (qllet 'eager
    ((foo (dolist (node (successor root-node))
      (cond ((c-test? node) (c-test token node))
            (t (qllet 'eager
              ((foo (eval-node token node)) ))))))))

(defun c-test (token node)
  (cond ((do-c-test token node)
    (eval-node-list token (successor node)) )))

(defun eval-node-list (token node-list)
  (cond ((null node-list)
    (t (let ((node (pop node-list)))
      (qllet (cond ((lock-node-p node) 'eager)
        (t t) )
        ((foo (eval-node token node))
          (bar (eval-node-list token node-list)) ))))))))

(defun eval-node (token node)
  (cond ((funcall (function node) token (arguments node))
    (eval-node-list token (successor node)) )))
```

Figure 3-1: QLISP code to evaluate Rete nodes in parallel.

3.1.1. Rule-level Parallelism

Rule-level parallelism is a very natural form of parallelism in production systems. Here the match for each individual rule is performed in parallel. In the context of our Rete-based implementation, this requires that we introduce *lock nodes* at points where a ctest node leads into a memory-node. All lock nodes before memory-nodes of the same rule use an identical lock, and those before memory-nodes of distinct rules use distinct locks. Figure 3-3 shows how the original Rete network of Figure 2-2 is modified to exploit rule-level parallelism. (Identical locks are shown grouped together in figure.) The locks are implemented using qlambda closures, and the code for one such lock node is shown in Figure 3-2. As discussed earlier, a QLISP closure ensures that only one process can be actively executing inside the closure. The proposed locks then ensure that all activations corresponding to a single rule are executed in sequence, which is the desired semantics for rule-level parallelism.

```
{qlambda-closure successor-node) ;; structure of lock node
{qlambda t (token node) (funcall (eval-node token node))} ;; qlambda closure
```

Figure 3-2: Code for the lock node.

Finally, we need to provide locks before the tokens enter the conflict-set, since the conflict-set is a global data structure and multiple processes should not be modifying it at the same time.

Using rule-level parallelism, previous studies [4] show that only about 3-fold speed-up can be obtained. This is (i) because the number of rules that require significant processing is small and (ii) because even amongst these *affected* rules there is a large variation in the processing requirements. To reduce this variation in the processing times, we now discuss exploiting parallelism at a finer granularity where the processing for a single rule can be done in parallel.

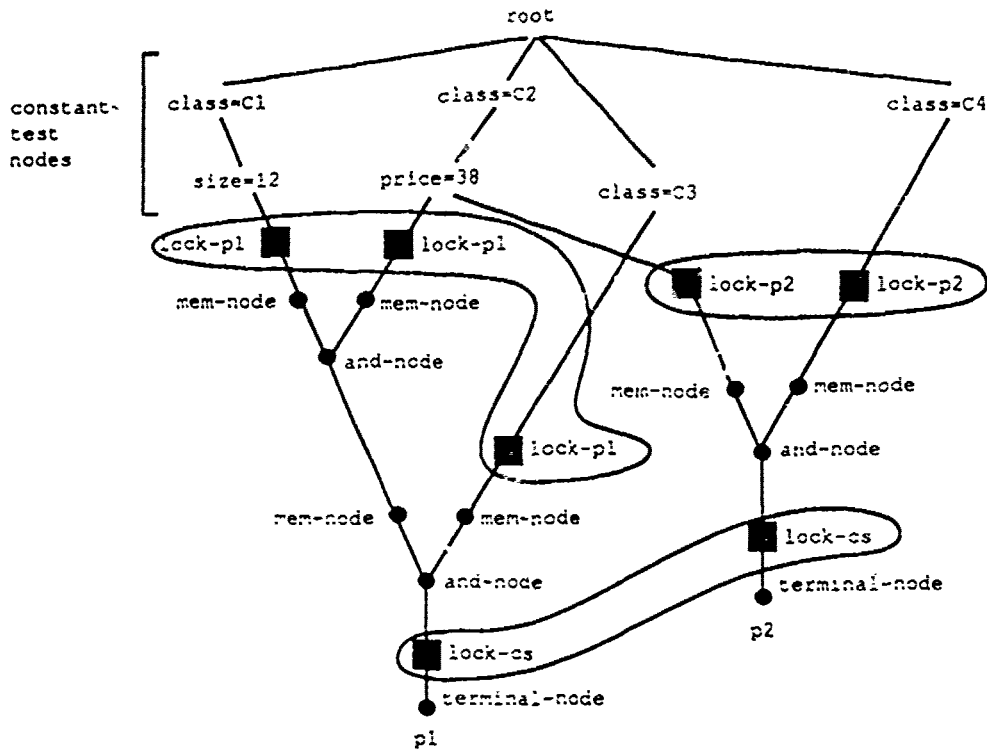


Figure 3-3: Modified Rete Network for Rule-level parallelism

3.1.2. Node-level Parallelism

When using node-level parallelism [4], any distinct two-input nodes can be evaluated in parallel⁵. To implement node-level parallelism, lock nodes are placed before each two-input node and its associated memory nodes as shown in Figure 3-4. The structure of a lock node is the same for node-level and rule-level parallelism. However, the value of the qlset predicate are different for evaluating different types of node activations. The predicate is τ for evaluating a memory-node and a two-input node, but it is 'eager' for evaluating successor nodes below a two-input node. That is, the execution of a two-input node is terminated by a future and the lock is released.

Note that if some two-input node generates multiple tokens, the next two-input node becomes a bottleneck. This is because only one activation of a given two-input node can be processed at the same time.

3.1.3. Intra-node Parallelism

The intra-node parallelism [4] exploits maximal parallelism present in the Rete algorithm. If multiple tokens arrive at a two-input node, then these multiple activations of the two-input node are processed in parallel. However, we have to be very careful about how we access the memory nodes: (i) it is not desirable to have multiple processes modifying the same memory node; and (ii) the correct operation of the Rete algorithm requires that the opposite memory-node should not be modified while processing a two-input node activation. To ensure the correct operation, we adopt the solution proposed by Gupta in [4]. We use a common hash-table for all tokens stored in the memory nodes of the Rete network. Tokens are put into hash-table buckets based on the node-id of the associated

⁵According to the result of the simulations of PSM, the speedup of node-parallelism is about 5-fold.

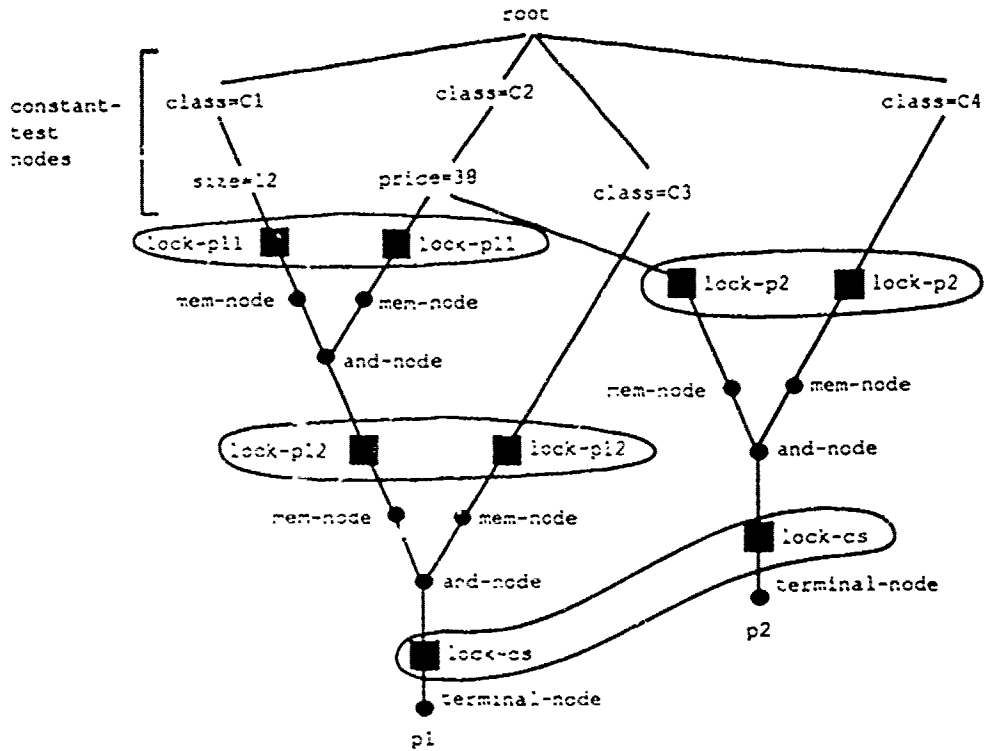


Figure 3-4: Modified Rete Network for Node-level parallelism

two-input node and some values that are tested from the token. The buckets in this hash-table are controlled by locks that are implemented as lambda closures. Figure 3-5 shows the structure of this hash table. This scheme works because the probability that multiple tokens would hash to the same bucket is considered small. If they do hash to the same bucket then they have to be processed sequentially.

In the above scheme, the Rete network reverts back to its original structure as shown in Figure 2-2 (except that locks are needed for executing the terminal nodes). All the remaining locks that were earlier associated with the Rete network are no longer present. Locking has now moved to hash-table buckets.

lock	left-hash-table		right-hash-table	
	token-list	extra-deletes-list	token-list	extra-deletes-list

Figure 3-5: Hash table for memory nodes

3.2. Conflict-Resolution Parallelism

During the conflict-resolution phase one of the several production instantiations in the conflict-set is selected for execution. The method by which this production instantiation is selected is called the conflict-resolution strategy. OPS5 provides for two conflict-resolution strategies -- LEX (lexical) and MEA (means-ends-analysis). The two differ in the way a key is constructed for sorting various instantiations. The key for LEX consists of the sorted time-tag values of the working-memory elements in the instantiation. The key for MEA consists of the time-tag of the first working-memory element in the instantiation, followed by the sorted time-tag values of the remaining working-memory elements in the instantiation.

To perform conflict resolution, normally, the conflict-set is maintained as a sorted list of production instantiations. Executing conflict-resolution in parallel imposes the following requirements:

- We must allow multiple instantiations to be inserted into or deleted from the conflict-set in parallel.
- We must allow for conjugate pairs of instantiations, that is, where the delete request for an instantiation is received before the add request.
- We would like to have the highest priority instantiation available to the RHS evaluation process as soon as possible, although the rest of the conflict-set data structure is not completely sorted.

To handle the first requirement, we build an asynchronous systolic priority queue structure in software [9] using QLISP. In this structure, inserts and deletes are input at the head of the priority queue. These then asynchronously filter down until they find the right position in the sorted queue. A delete may annihilate an element if it is already present. If a delete does not find a corresponding element already there (conjugate token problem), it locates itself at the right location in the queue with a special flag, and waits for the corresponding add request to come by later. An insert behaves similarly. The key point is that the highest priority instantiation is always available at the head of the queue, even if elements are still percolating down in the lower priority regions of the queue. The data structure that we use for a single instantiation in the priority queue is shown in Figure 3-6 and some related code is shown in Figure 2-4.

```

conflict-set-element =
  (key next-element positive-instance-list negative-instance-list)

where next-element = (qlambda-closure . conflict-set-element)
  key = (sorted-time-tag-of-Instance-element . rating-of-production)
  positive-instance-list = (positive-instance ...)
  extra-deletes-list = (extra-deletes-instance ...)
  positive-instance = ((flag . simplified-form) production . instance-element-list)
  extra-deletes-instance = (production . instance-element-list)

```

Figure 3-6: Representation of a production instance

The time to calculate the maximum element in the above scheme is $O(k)$, where k is the number of changes to the conflict-set per recognize-act cycle. Since k is around 5 for most systems this is not a problem. The time to finish sorting, however, can be much larger. This time is $O(N \times k)$, where N is the total number of elements in the conflict-set, which can be much larger. This is not optimal for sorting, but it is good for getting the highest priority element. The highest priority element is used in the speculative execution of the RHS.

3.3. Speculative Execution of RHS

In the normal execution of a rule-based system, one would wait until conflict-resolution finishes completely before starting to execute the RHS of the highest priority rule. However, in a parallel implementation, this may imply too sequential a behavior. Even if RHS execution takes only 10% of the time, this limits the maximum speed-up to 10-fold. As a solution, we propose the speculative evaluation of RHS in this paper. By speculative evaluation of RHS we mean the following. While the match and conflict-resolution are still going on, we make a guess about the highest priority rule. (This in our case is simply the rule currently at the head of the conflict-set.) We start evaluating the RHS of this rule, i.e., gathering up the changes it would make to working memory in a list (without actually changing the working memory). If our guess is proved wrong, that is whenever there is a change in the rule at the head of the conflict-set, we simply create a new process to evaluate the RHS of this new rule. We currently do not abort the previously evaluating RHS because aborting is not easy to implement in QLISP. Furthermore, it is possible that the evaluated RHS of the non-highest rule may come in useful on a later cycle.

The OPSS QLISP system provides a new action command `sfcall`, side-effect-free call which execute a user-defined routines written in QLISP or in Lisp. These user-defined routines should not refer any global data which may be modified by other routines, because the system assumes that simplification should be valid at any time and independent from any global context. The algorithm of simplification is sketched below:

1. Check the type of operations.
2. If a working memory operation, calculate all arguments and make a token.
 - If make, make a token of add and replace the original action with it.
 - If remove, make a token of delete and replace the original action with it.
 - If modify, make a token of delete and a token of add and replace the original action with them.

However, if an action contains a function such as `accept`, `acceptline`, these functions are not executed. Only omitted attribute-value pairs are supplied and the original action is replaced with a new action which has all attribute-value pairs.

3. If a side-effect-free call `sfcall`, do it.
4. Otherwise, process next action.

This simplification is quite similar to the argument evaluation for a Lisp function with keyword arguments of the Common Lisp. The simplification routine is invoked when the maximum production instance of conflict-set is changed and stores a simplified form to the simplified form slot of the instance. Note that this simplified form is valid for any time, because it is calculated with using only local values which is specified in an instance. Conjugate pairs may create unnecessary processes, but the current implementation does not abort them, because such an aborting mechanism is not easy to implement and the number of conjugate pairs are not expected to be large.

4. Discussion

In this paper, we present the details of an implementation of the OPSS production-system language using QLISP, a parallel dialect of Lisp. We would like to make the following observations:

- The number of modifications needed to the original lisp code for OPSS were minimal to exploit the different kinds of parallelism. For example, to exploit the three kinds of parallelism described for match, less than 100 lines of code (out of a total of about 3000 lines in the original code) had to be modified or added. We believe that such a high-level programming approach provides very powerful and flexible tools for research in parallel programming.
- The QLISP constructs that we used most frequently in our parallel implementation are "(qlet 'eager ...)" to spawn new processes and "(qlambda t ...)" process closures for locks. The code sections that are locked and the processes that are spawned consist of a few lines of lisp code with some but not much recursion or iteration. On average, we expect the individual tasks to take about 1 millisecond of

computation time on a 1 MIPS machine. This requires that the process creation overhead, the locking overhead, and the scheduling overhead for the spawned tasks be significantly less than 1 millisecond, if the suggested implementations are to be useful. If the overheads are much larger, then all the advantages of parallel execution will be subsumed by the overhead.

- We are currently using a QLISP simulator to obtain some performance numbers. Our implementation is running, and we have just started getting some performance numbers. Unfortunately, the simulator does not model the underlying hardware very accurately, so we still do not have a good idea about the true overheads involved. However, for reasons mentioned in the next point, this may not be a big problem in practice.
- The parallel constructs provided by QLISP (qlet, qlambda, ...) take a predicate that controls whether a parallel process is actually spawned or not. This convenient run-time method of controlling the granularity at which parallelism is exploited is a very powerful mechanism. It makes it extremely easy to modify code to adjust to different implementations with differing overheads. It is also convenient to adjust the granularity depending on the load present on the parallel machine.
- As stated in the beginning of this paper, another advantage of implementing OPS5 in QLISP, instead of in Pascal or C, is that it is easy to embed the OPS5 system within other AI systems (which normally use Lisp). Furthermore, if there are complex functions in the RHS of rules, then these functions can also use the parallel constructs available in QLISP, which is not possible in previously proposed parallel implementations of OPS5.
- As a final means for improving performance for existing OPS5 systems we are planning to directly compile OPS5 into QLISP code, instead of using an interpreter as we currently do.

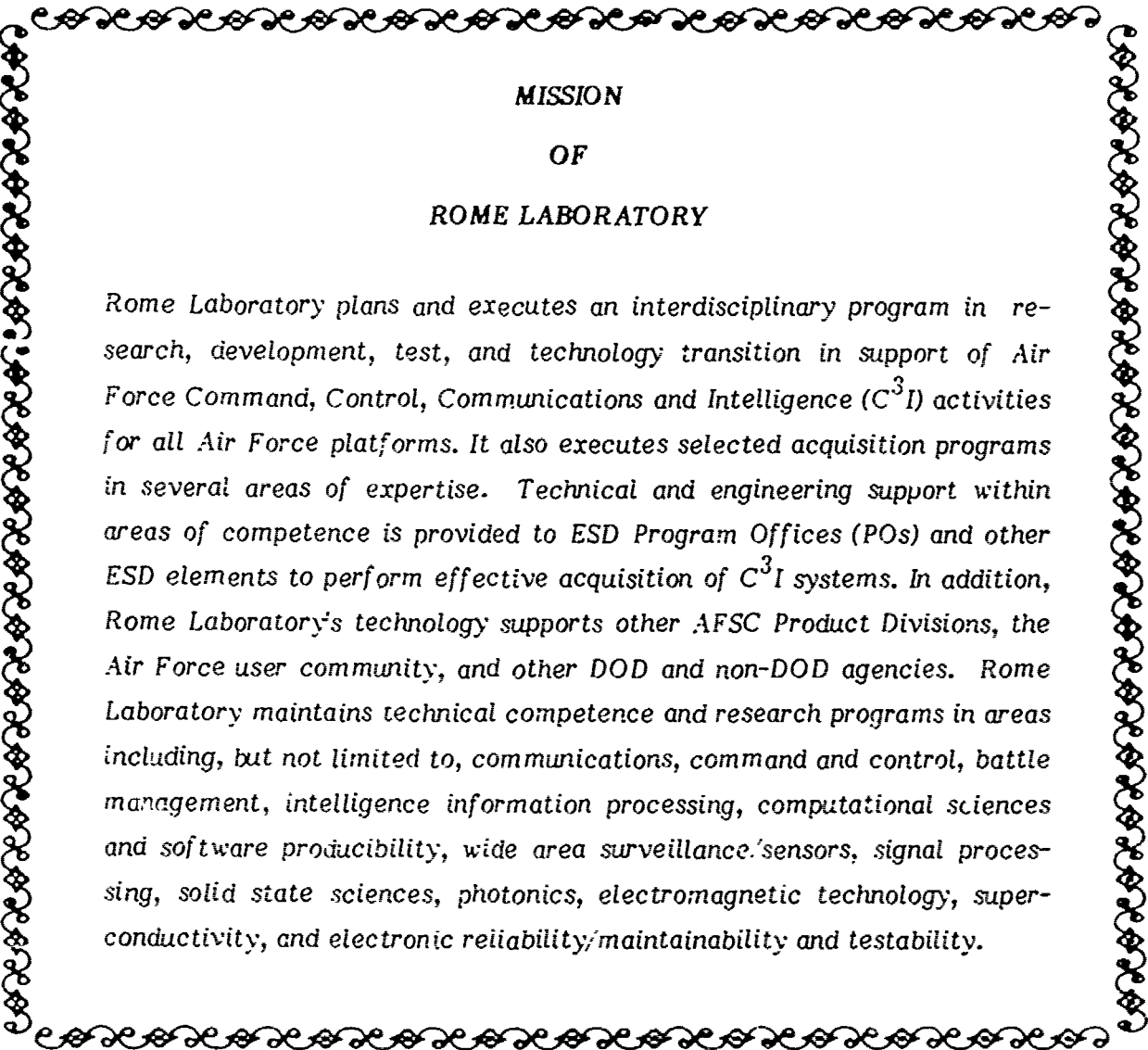
Acknowledgments

The authors would like to thank Prof. Edward Feigenbaum for supporting this work at Knowledge Systems Laboratory at Stanford University. We would also like to thank members of the QLISP group at Stanford. We would especially like to thank Joe Weening for help with the QLISP simulator.

The computing facilities used in doing this work and writing this paper are supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Anoop Gupta is also supported by a faculty grant from Digital Equipment Corporation.

References

1. Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5. An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.
2. Forgy, C. L. On the Efficient Implementations of Production Systems. PhD thesis, Technical Report CMU-CS-79, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, February, 1979.
3. Gabriel, R.P. and McCarthy, J. Queue-based multiprocessor Lisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984.
4. Gupta, A. Parallelism in Production Systems. PhD thesis, Technical Report CMU-CS-86-122, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, March, 1986.
5. Halstead, R. MultiLisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984.
6. Halstead, R. "MultiLisp: A Language for Concurrent Symbolic Computation". *ACM Transaction on Programming Languages and Systems* 7, 4 (October 1985).
7. Halstead, R. "Parallel Symbolic Computing". *IEEE Computer* 19, 8 (August 1986), 35-43.



MISSION
OF
ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.