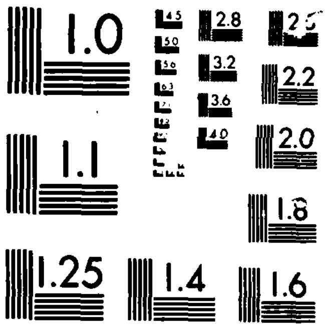
AD-A167 622 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN B W SMITH MAY 85 ARO-18790.17-EL-APP-G DAAG29-82-K-0101 1/4

UNCLASSIFIED F/G 9/2 NL



MICROCOPY CHART

ARO 18790.17-EL
app-G

2

ON THE DESIGN AND MODELING OF SPECIAL PURPOSE PARALLEL PROCESSING SYSTEMS

AD-A167 622

Ph.D. Thesis by:
Bradley Warren Smith

Faculty Advisor:
Howard Jay Siegel

Appendix G for

Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation Final Report

U.S. Army Research Office
Contract No. DAAG29-82-K-0101*

*Chapters 1, 4, and 5 supported by this contract.

86 4 28 180

ON THE DESIGN AND MODELING OF SPECIAL PURPOSE PARALLEL PROCESSING SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Bradley Warren Smith

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 1985

ACKNOWLEDGMENTS

The author wishes to extend his thanks to Dr. James Tilton for providing timing data about the 11/70 and for patiently dealing with the author's ignorance about contextual classification. Special thanks is also extended to Mark A. Yoder, without whom the discussion on the isolated word recognition system could not have been done in real-time and to G. Cooper, E. Coyle, M. Franklin, D. Gannon, G.J. Lipovski, D. Meyer, M. Rodwell, H.J. Siegel, P. Swain, and A. Van Tilborg whose careful readings and comments helped to organize and clarify the ideas presented in this document. Special thanks is extended to D. Curry and K. Rodwell for their time and effort in the final rush in producing this final copy.

Portions of the research presented here were sponsored by the National Aeronautics and Space Administration, under Contract No. NAS729-15466, the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581, the Defense Mapping Agency, monitored by the United States Air Force Command, Rome Air Development Center, under Contract No. F30602-C-0193, the U.S. Army Research Office, Department of the Army, under Contract DAAG29-82-K-0101, and by the National Science Foundation under Grant ECS-81-20896.

Some of this material was/will be presented in:

H.J. Siegel, P.H. Swain, and B.W. Smith, "Remote sensing on PASM and CDC flexible processors," *Multi-Computer Algorithms and Image Processing*, ed. K. Preston and L. Uhr, Academic Press, New York, NY, 1982.

B.W. Smith, H.J. Siegel, and P.H. Swain, "Contextual classification on a CDC flexible processor system," *1981 Machine Processing of Remotely Sensed Data Symp.*, pp. 283-291, June 1981.

B.W. Smith, H.J. Siegel, and P.H. Swain, "Parallel processing concepts for remote sensing applications," *1982 Machine Processing of Remotely Sensed Data Symp.*, Purdue Univ., pp. 520-526, June 1982.

B.W. Smith and H.J. Siegel, "Models for use in the design of macro-pipelined parallel processors," *12th Int'l. Computer Architecture Symp.*, June 1985.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Smith, Bradley Warren. Ph.D., Purdue University, May 1985. ON THE DESIGN AND MODELING OF SPECIAL PURPOSE PARALLEL PROCESSING SYSTEMS. Major Professor: Howard Jay Siegel.

As the capabilities of computing machinery grow, so does the diverse variety of their applications. The feasibility of many approaches to these applications depends solely upon the existence of computing machinery capable of performing these tasks within a given time constraint. Because the majority of the available computing machinery is general purpose in nature, tasks that do not require general purpose facilities, but that do require high throughput, are condemned to execution on expensive general purpose hardware.

This research describes several tasks that require fast computing machinery. These tasks do not require general purpose facilities in the sense that the computing machinery used will only perform a fixed set of tasks. Some of the tasks are simple in nature, but are required to execute on very large data sets. Other tasks are computationally intensive in addition to possibly involving large data sets. Both simple and complex algorithms are considered. The discussion includes a description of the tasks.

All of the above tasks are useful; however, their value is determined in part by the time required to perform them. This work discusses three architectures for performing remote sensing tasks. These architectures can execute the described tasks more quickly than conventionally available hardware.

The discussion extends to the realm of designing macro-pipelined distributed computer systems for special purpose applications. Nine parameters are introduced along with a proposal for an algorithmic approach to designing a computer system for a special application. The parameters are then applied to an isolated word recognition system.

For may tasks (especially those involving feedback), it is undesirable to use synchronous parallelism. A study, including a probabilistic model, of the effects of using asynchronous stages in the macro-pipeline is presented. Simulation is used to verify the results.

CHAPTER 1
INTRODUCTION

1.1 Overview

For many applications, response time and throughput are of critical importance. Such applications include: defense against incoming missiles, missile guidance, air traffic control, weather analysis, speech recognition, and tomography. The principal goal is to process the data in "relevant" time within some cost criteria. Further, the feasibility of performing many tasks depends on the capability to execute them in a certain amount of time without excessive hardware expense.

General purpose hardware, while less expensive than special purpose hardware, is typically slower than hardware designed for a specific task. The design of special computing facilities can take large amounts of time and manpower, increasing the design overhead of such a system over a general purpose system. Since special purpose computer systems typically do not sell in large quantities, the design cost must be distributed over a relatively small number of units. Thus, the cost of special purpose computer systems can be considerably greater than that of general purpose computer systems. The high cost of special purpose hardware decreases the desirability of algorithms that require special purpose computer systems. Thus, accurate and powerful algorithms may not be used in lieu of less accurate algorithms or, even worse, nothing at all. To help reduce the cost of special purpose systems, computer aided tools can be used to minimize the human intervention needed in the computer design process. These tools would reduce the design time. To achieve this goal, tasks must be modeled as to the type of computational resources they require. Further, presently available hardware, such as small boards and chips, must be modeled according to their computational capabilities. By extending the models to parallel schemes, combination of the two models allows systems to be proposed or built to perform computationally intensive tasks within some time, cost, or other constraint.

This research is divided into four chapters. Chapter 2 considers the application of parallelism to contextual classifiers for image analysis which are being developed to exploit the spatial/spectral content of a picture element (pixel) to achieve higher classification accuracy. Contextual classification requires large amounts of computation, so special hardware is of value. Chapter 2 explores the CDC Flexible Processor (FP) system ([CDC77a],[CDC77b]) and the proposed multimicroprocessor system PASM [SiS81], which are both parallel processing systems that can be applied to image processing tasks. Timings for the FP system to perform contextual classifications, based on a Purdue developed FP system simulator, are presented. For comparison, the same algorithms have been run on a PDP-11/70. The applicability of PASM for implementing the contextual classifier is demonstrated by algorithm complexity analysis. The reduction in execution achieved through the use of these parallel systems is shown.

The research in Chapter 2 has suggested a specific architecture for the application of parallel processing to remote sensing tasks. Chapter 3 proposes such an architecture. It is a large-scale multimicroprocessor structure which could consist of as many as 1024 processors. This type of architecture is extremely well suited to the execution of window and pixel based operations. A number of remote sensing data processing techniques for implementation on a machine with this architecture are discussed. Algorithms considered are: image smoothing, image correlation, and contextual and non-contextual methods of image analysis. This includes both the design of parallel algorithms and the exploitation of appropriate data structures.

In addition to demonstrating how various algorithms can be performed on the parallel architecture, Chapter 3 proposes extensions to the architecture to increase its fault tolerance. Then, a specific implementation of the architecture, called MuRSS, is contrasted to an already existing system called MPP. MuRSS and MPP are compared with respect to speed, processing capabilities, and fault tolerance.

In Chapter 4, an approach to modeling distributed macro-pipelined computer systems is examined. This chapter uses nine parameters to form a model of the characteristics of parallel/distributed algorithms and the environment in which they must execute. These parameters describe the I/O environment, the algorithm, the memory requirements of the algorithm, and the type and amount of arithmetic calculations required by the algorithm to process a normal data set.

In addition, Chapter 4 uses tuples to model the characteristics of computer architectures. These tuples describe the instruction set, the instruction processing times, the size and speed of on-board cache, the data and address widths, the replication of units, the number of stages in pipelined units, and the functional overlap for each unit in the architecture. By combining the tuples with the nine parameters, the execution time of the algorithm modeled by the parameters on the hardware modeled by the tuples can be estimated. The combination of these two models could be used as a basis for computer aided design tools used for special purpose parallel/distributed processors. This chapter uses a layered method of architecture design, in which a task is broken down into sub-tasks. Each sub-task is then assigned to a special purpose processing unit. Such a unit may be either a traditional serial type design or a parallel design.

Chapter 5 extends the work done in Chapter 4 by looking at the effects of both synchronous and asynchronous stages in macro-pipelined machines. Two synchronous schemes (double buffering and triple buffering) are compared to an asynchronous system with respect to throughput and system response time. Theoretical results are presented. A simulator to calculate the throughput and system response time of each system has been developed to verify the theory. The results of the simulation of over 200,000 data sets are presented.

1.2. A Survey of Parallel Architectures for Image Processing

The purpose of remainder of this chapter is to give background information pertinent to the rest of this work. Two taxonomies or hardware description schemes are discussed in Section 1.3. Sections 1.4 and 1.5 describe a number of proposed and implemented parallel and/or distributed processing systems that can be used for image processing. The systems discussed in this chapter are: **CLIP4** - the **C**ellular **L**ogic **I**mage **P**rocessor [Duf82, DuW73, Fou81, Ger83]; **Cytocomputer** - a pipelined image processor [PrD79, Ste80]; **DAP** - the **D**istributed **A**rray **P**rocessor [Ger83, Hun81, Red79]; the **FP** array - CDC's **F**lexible **P**rocessor array [All82, SiS80, SiS82c, SmS81, SwS80]; **MPP** -

the **M**assively **P**arallel **P**rocessor [Bat80, Bat82, Ger83, Pot82a]; **PASM** - the **PA**rtitionable **SIMD/MIMD** system [SiM81a, SiS81, SiS82c, Sie81]; **PICAP** - the **PIC**ture **A**rray **P**rocessor [KrD82, KrG82, Gud81]; and **STARAN** - Goodyear Aerospace's associative processor system [Bat74, Bat76, Bat77b, Bat82, FeF74, Ger83, Thu76, Pot82b].

1.3. Hardware Taxonomies

Currently, there are two classes of computer hardware taxonomies. There are hardware taxonomies that classify (e.g., tiger) and those that describe (e.g., four paws, 16 sharp claws, ravenous meat liking appetite, etc.). The classification taxonomies provide only the most general information, omitting details for ease of use. Several descriptive taxonomies have been developed to accurately describe the architecture of computer hardware. These descriptive taxonomies are often so cumbersome that they cannot be used verbally to convey their thought.

One of the first taxonomies, proposed in [Fly66], is a classification taxonomy. This taxonomy classifies a system based on the number of concurrent instruction and data streams. A machine has either a single stream or multiple streams in this taxonomy.

A machine that executes a **S**ingle **I**nstruction stream on a **S**ingle **D**ata stream is called an **SISD** machine. Some examples of SISD machines are the IBM 370/155, the DEC PDP-11/70, and the DEC VAX-11/780. Machines that execute a **S**ingle **I**nstruction stream on **M**ultiple **D**ata streams are called **SIMD** machines. Some examples of SIMD machines are CLIP4, ILLIAC IV [Bar68, Bou72, Sto80], MPP, PASM (in SIMD mode), PICAP I, and STARAN. In such systems, a control unit broadcasts the same instruction to all processors, and all enabled processors execute the same instruction simultaneously, each processor on its own data stream. Machines that execute **M**ultiple **I**nstruction streams on **M**ultiple **D**ata streams are called **MIMD** machines. Some examples of MIMD machines include the CDC Flexible Processor Array, PASM (in MIMD-mode), PICAP II, and Cytocomputer [LoM80]. A machine that executes a **M**ultiple **I**nstruction on a **S**ingle **D**ata stream is called an **MISD** machine. Macro-pipelined machines fall into this category. The design of such machines is the topic of discussion for Chapter 4.

The classes of machines in this taxonomy are very broad. For example, MPP, whose **P**rocessing **E**lements (**PEs**) operate on one bit of data at a time falls into the same class (SIMD) as ILLIAC IV, whose PEs operate on 64 bits of data simultaneously. In addition, this taxonomy gives no indication of the relative size of a machine. For example, PASM (in MIMD mode), which could consist of up to 1024 PEs, is in the same class as the CDC FP array, which can consist of up to 16 PEs. Several taxonomies have been proposed to narrow the classes, at the expense of simplicity. Flynn's taxonomy, however, still remains the simplest and most widely used.

In contrast to Flynn's taxonomy, which categorizes computers according to their instruction and data streams, the classification taxonomy in [Kuc78] proposes to classify hardware according to the instruction stream(s), instruction type, execution stream(s), and execution type. As in Flynn's taxonomy, the instruction and execution streams can be either single or multiple. The instruction and execution types can be either scalar or array.

The number of instruction streams is determined by the number of concurrently executable programs. For a program to be executable, it requires a program location counter to point to the next instruction to be executed.

If the arguments to any machine language instruction (operands) are arrays, the instruction type is array. If no machine language instruction can accept an array (vector) as an argument, the instruction type is scalar. For example, consider the instruction:

move a,m

If "a" is a single element and "m" is a memory location this instruction type is scalar. Systems that have scalar type instructions include: the AMD 9511A [Amd82], the CDC FP array [CDC77a, All82], the CDC 6600 [Che80], CLIP4 [Duf82, Fou81], ILLIAC IV [Bar68, Bou72], MPP [Bat80], PASM [SiS81, Sie82], and STARAN [Bat76, Bat77b]. For the instruction:

move a,m,1000

if "a" is the base address of an array, "m" is a memory location, and 1000 is the number of bytes to be moved, then the instruction is implicitly performing an array operation. For this latter case, the instruction type is array. For a system to have array type instructions, it must include at least one array instruction. Systems that have array type instructions are: OMEN [Thu76], VAMP [Che80, Thu76], and the TI-ASC [Che80]. An example of a chip that has an array type instruction is the Zilog-Z80 [SiS83].

The number of execution streams is determined by the variety of operations that can be performed simultaneously by the system. Either a system can perform a single operation or multiple operations at once. Multiple copies of a single operation count as a single operation. Systems that fall into the single execution stream category are all systems in the SISD and SIMD classes of Flynn's taxonomy that allow no overlapping of different instructions (e.g., no overlap of control unit and PE operations). An example of a machine that has a single instruction stream of scalars with a multiple execution stream is the CDC 6600. The CDC 6600 has two multipliers, the execution of which can be overlapped with the addition unit. From a single job stream, both an addition and multiplication can be taking place at the same time, although they cannot be initiated simultaneously, thus, there exists multiple execution streams. Another example of a machine that has a single instruction stream of scalars with a multiple execution stream is the VAX 11/780 with the floating point accelerator. A VAX 11/780 can overlap slower floating point operations with integer instructions, giving multiple executions simultaneously. Without the floating point accelerator, the VAX cannot overlap operations in any way, thus the system must wait for the result of any operation before continuing. Thus, the VAX without the floating point processor is an example of a system that has a single instruction and single execution stream.

The execution type is either scalar or array and is determined by the number of operands to which a machine language instruction can be applied simultaneously. A system where a single machine language instruction operates on multiple operands, like the ILLIAC IV SIMD machine, which issues scalar instructions that act upon 64 operands, is said to have an array execution type. If no machine language instruction can act on multiple operands simultaneously, the execution type is scalar.

The nomenclature is formed by describing the instruction stream and type with the execution stream and type. Systems such as the PDP-11/70, which have a Single Instruction stream that performs Scalar instructions on a Single Execution stream of Scalars are classified as: **SISSES**. ILLIAC IV, which has scalar type instructions fetched by one control unit and broadcast to 64 execution units, is classified as: **SISSEA** (assuming no instructional overlap is allowed). The CDC 6600 has a single instruction stream of scalar instructions that control a multiple execution stream of scalars and is classified as **SISMES**. The TI-ASC has a single instruction stream of array instructions that controls a multiple execution stream of array operations is classified as **SIAMEA**. Table 1.3.1 [SiS83] shows what machines fall into which classes. Kuck's scheme is a more precise classification taxonomy; however, it is also more cumbersome to use.

The descriptive taxonomy set forth in [HoJ81] describes the architecture of a machine in an algebraic style suitable to printing and entry into a computer. A SISD computer in this notation would be described as:

$$C=I[E-M]$$

This means that the computer (C) is composed of a single instruction unit controlling an execution unit (E) and a memory unit (M). There are twenty rules that govern symbols, their use, and how they are connected. A synopsis of this notation appears in both [HoJ81] and [SiS83].

Other descriptive taxonomies are set forth in [Gil83] and [BeN71]. These notations, while similar to the notation set forth in [HoJ81], have one important conceptual difference. The notation in [HoJ81] is specifically two dimensional, i.e., the architecture of the system can be described in a two dimensional manner. The notations in [Gil83] and [BeN71] are three dimensional in nature, making them very difficult to parse. A discussion of each of the taxonomies appears in [SiS83], along with several examples. In general, Flynn's hardware classification scheme will be used here. A special descriptive

Table 1.3.1
Kuck's sixteen categories of computer architectures [SiS83].

		SINGLE EXECUTION		MULTIPLE EXECUTION	
	TYPE	SCALAR	ARRAY	SCALAR	ARRAY
SINGLE INSTRUCTION	SCALAR	PDP 11/45	ILLIAC IV STARAN (PASM) (TRAC)	CDC 6600 CPU	OMEN-60
	ARRAY	ZILOG Z80	CYBER 203/205	NONE KNOWN	CRAY-1 BSP CDC 7600 TIASC
MULTIPLE INSTRUCTION	SCALAR	CDC 6600 PPU	NONE KNOWN	BURROUGHS FMP DATA FLOW (PASM) (TRAC)	DENELCOR HEP PASM (TRAC)
	ARRAY	UNDESIRABLE DESIGN	NONE KNOWN	NONE KNOWN	PEPE CDC NASF TRAC PUMPS

taxonomy is needed and is proposed in Chapter 4. There, computer hardware needs to be described by its capacity and speed of execution in such a manner that timing information can be simply obtained.

For the application in Chapter 4 that Flynn's taxonomy does not provide enough information about system architecture to be of use. The taxonomy in [Gil83] limits the level of description of a system in addition to not specifically stating how a system's resources are to be connected. A more explicit representation of the overall system architecture can be found in [BeN71]; however, this description is two dimensional. Thus it is inconvenient to store in a computer, and quite difficult to analyze. Finally, it is undesirable to apply the taxonomy set forth in [HoJ81] because the depth of the description is arbitrary. Therefore, different people can differently describe the same machine. Thus, while all of the above taxonomies are of importance, none is directly applicable to the application in Chapter 4.

1.4. SIMD Systems

The SIMD systems discussed in this work fall into the following two categories. **Bit-serial** systems are composed of PEs that can process only a single bit at a time. **Bit-parallel** systems are composed of PEs that process multiple bits at once. Such PEs are said to process words. CLIP4 [Duf82], DAP [Red79], MPP [Bat80, Bat82], and STARAN [Bat74, Bat76, Bat77b, Pot82] are all bit-serial systems. ILLIAC IV [Bar68, Bou72], MuRSS [SmS82], and PASM [SiS81] are all bit-parallel or word organized system. All of the systems, except PASM, are purely SIMD machines. PASM, however, can be either SIMD or MIMD as needed.

Section 1.4.1 will discuss DAP, CLIP4, and STARAN. The strengths and weaknesses of DAP, CLIP4, and STARAN are presented in Section 1.4.2. ILLIAC IV and its applications have been extensively discussed in [Bar68, Bou72, HoS82, Sto80, Thu76]. PASM is described in Chapter 2. Both MuRSS and MPP are presented detail in Chapter 3. For brevity, a discussion of ILLIAC IV, PASM, MuRSS, and MPP is omitted here.

1.4.1. Three Bit-serial SIMD Systems

The Cellular Logic Image Processor (CLIP) series of processors was first completed in 1971. Since that time, five variations on the original machine have been built. Most recently, CLIP4, a 96-by-96 processor array, designed to process video input from a TV camera, was completed. The organization of the CLIP4 system is shown in Fig. 1.4.1.1 [Duf82]. Each PE has 32-bits of memory associated with it. The incoming video image is digitized into 6-bit quantities which are then processed bit-serially (as six bit-planes) by the 96-by-96 array of PEs. To control the array, extract instructions, and, coordinate the peripherals associated with the array, a controller is provided. A PDP-11/10 acts as host for the system.

A PE in CLIP4 can communicate with either its eight nearest neighbors or its six nearest neighbors depending on which communication mode is selected. These two modes are shown in Fig. 1.4.1.2 [Duf82]. The internal organization of a PE is shown in Fig. 1.4.1.3 [Duf82]. The boolean processor can perform all boolean operations on single-bit inputs. Addition (subtraction) can be done by performing the logical operations to generate the sum (difference) and then generating the carry (borrow). Carries (borrows) are then routed through the

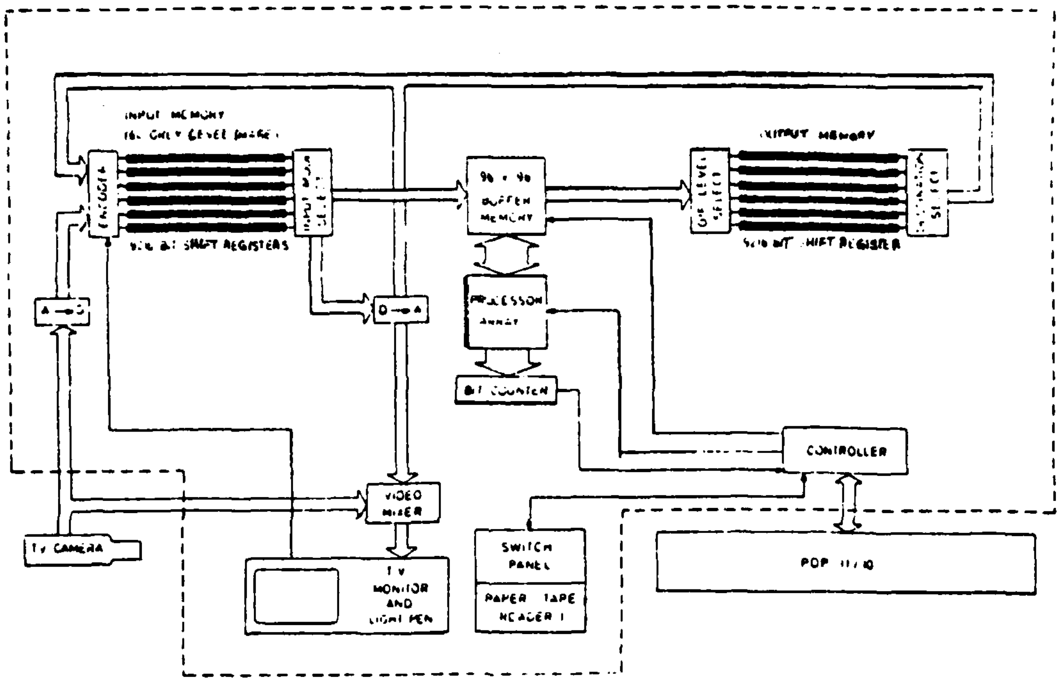


Fig. 1.4.1.1 The CLIP4 system configuration [Duf82]

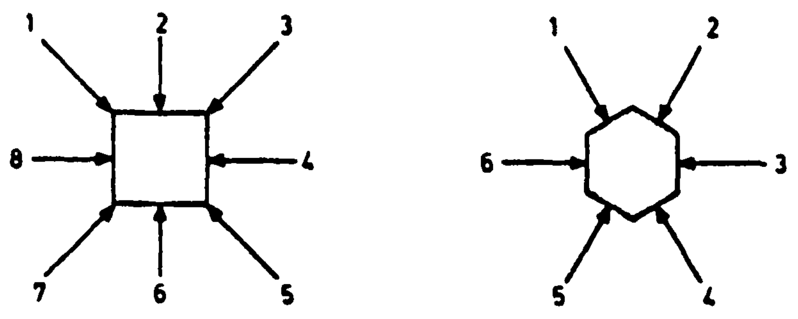
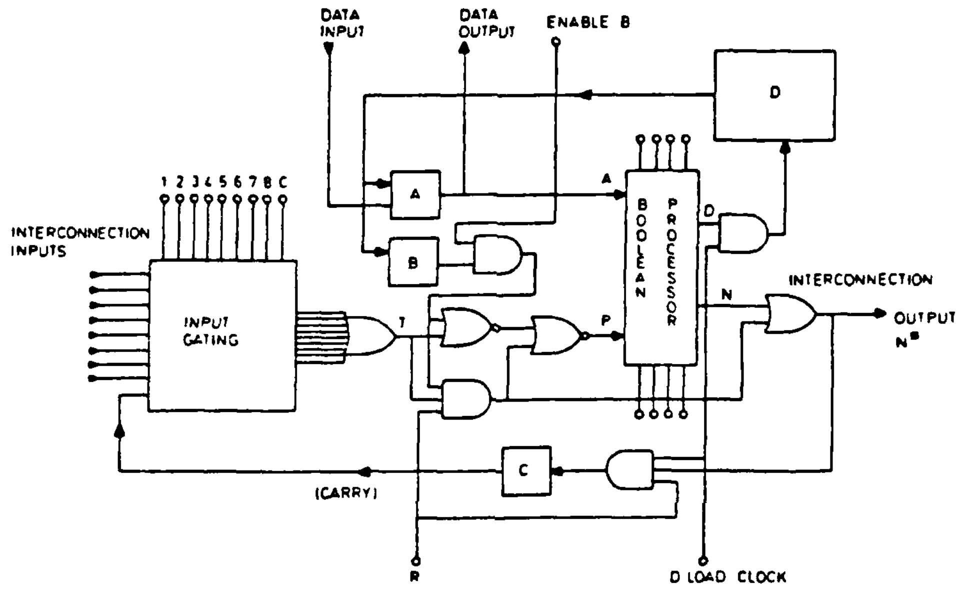


Fig. 1.4.1.2 Interconnection in CLIP arrays [Duf82]



1.4.1.3 The complete logic circuit for CLIP4 [Duf82] gating array for use in calculating the next bit.

In conclusion, CLIP4 can perform **picture element (pixel)** independent operations, i.e., operations where each pixel is treated independently of its surrounding pixels, as well as many nearest neighbor operations. CLIP4 is capable of performing a variety of image processing tasks in real-time.

To process computationally intensive tasks, the **Distributed Array Processor** (**DAP**) project was started at ICL in 1972. The result of this project was a 64-by-64 array of PEs called the ICL DAP. Unlike CLIP4, DAP is 4-connected. This corresponds to a subset of the eight nearest-neighbor interconnection function presented in Fig. 1.4.1.2 consisting of connections 2, 4, 6, and 8. The architecture of the PE is shown in Fig. 1.4.1.4 [Red79]. The ALU in a DAP PE is very simple. Many logical functions must be broken down into sequences of AND and NOT operations.

Instead of having 32-bits of memory associated with each PE, like CLIP4, the DAP PEs have 4k-bits of RAM associated with each PE. All input and output to DAP is done through the hosts memory, i.e., the DAP memories are a portion of the hosts memory. This has the advantage that it eliminates idle transfer time, but it requires the DAP to be used in conjunction with an ICL 2900 series mainframe, which is expensive (cost: $ 1,000,000 and up) [Ger83]. A detailed comparison and contrast of DAP, CLIP4, and MPP appears in [Ger83].

STARAN [Bat74, Bat76] is a bit-serial system that differs greatly from CLIP4 and DAP. The original STARAN is composed of 256 PEs, a 256-by-256 bit **Multi-Dimensional Access** (**MDA**) memory, and an interconnection network. The MDA memory can be accessed by bit-slices, byte-slices, words, or by other portions. In STARAN-E [Bat77b], the MDA memory is composed

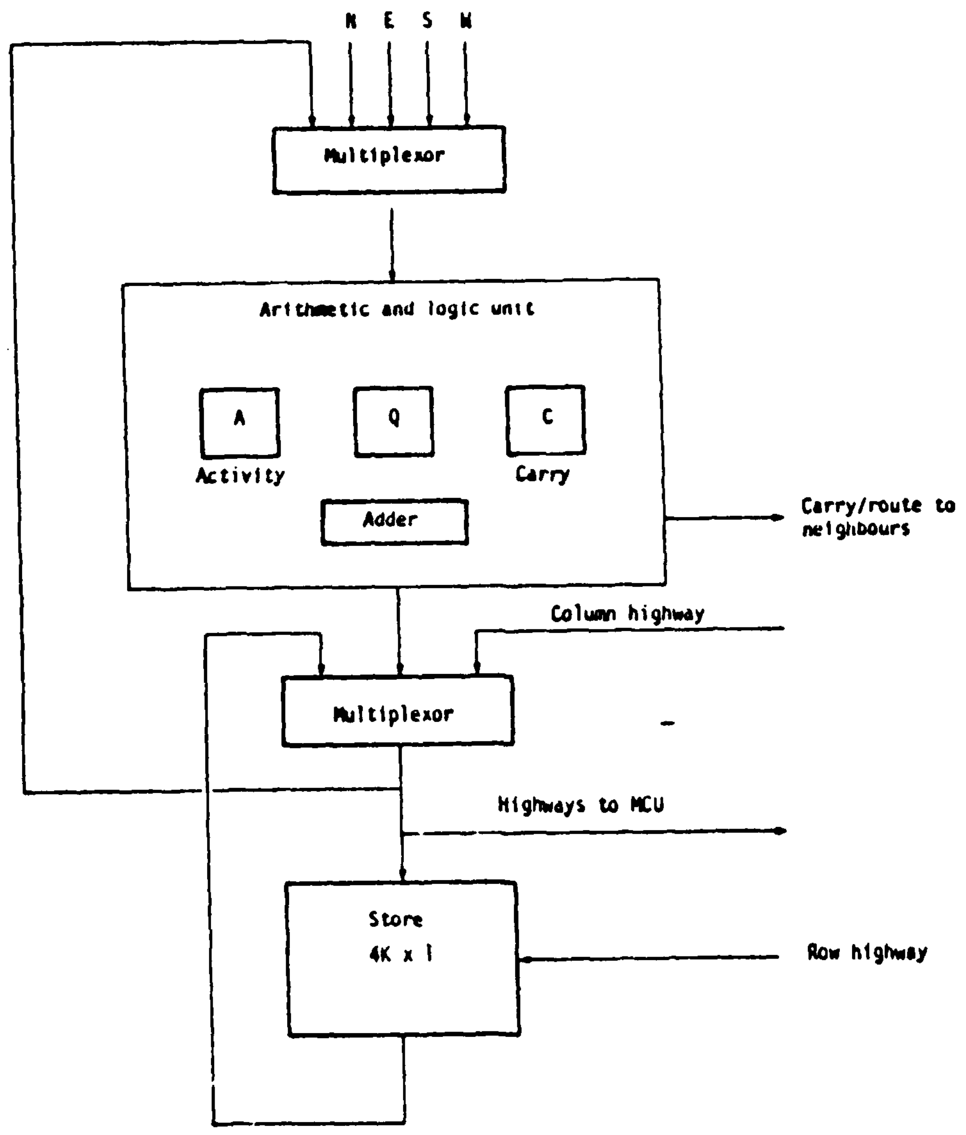
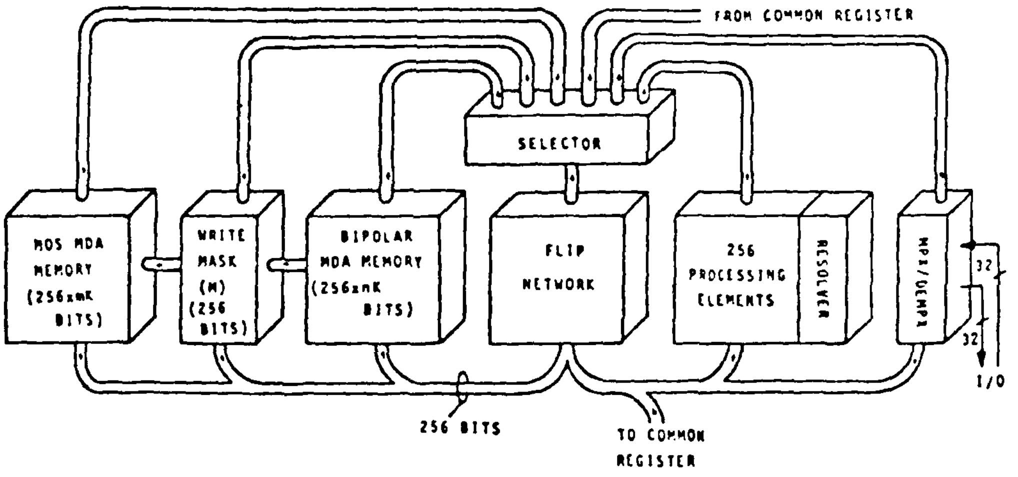


Fig. 1.4.1.4 DAP processing element [Red79]

of up to 256 256-by-256 bit planes of memory. STARAN-E is shown in Fig. 1.4.1.5 [Bat77b]. Instead of having the nearest-neighbor interconnections, like CLIP4 and DAP, STARAN is equipped with a multistage permutation network called the flip network. This is a multistage cube type of network [Sie85]. Its capabilities are discussed in [Bat76].

Fig. '.4.1.6 [Thu76] shows the layout of the STARAN memory array. Two registers, (X and Y) represent 256 1-bit PEs. The logic associated with the X- and Y- register can perform any of the sixteen Boolean functions of two variables. Inputs for the two variable Boolean functions are the present state of the register and the input from the permutation network, which can either be memory or the output of another PE. In addition for PE i, either X_i or Y_i may be used as a mask for an operation on the other register, Fig. 1.4.1.6 e.g., $X_i \leftarrow f(X_i,\text{network}_i)$ if $Y_i = 1$ (i=0, 1, ..., 255). The status of M_i determines which memory locations are modified for a masked write operation. Addition on STARAN is demonstrated in [Bat74].

STARAN was designed to be connected to a variety of host computers as a special purpose peripheral. Three systems cited in [Bat74] are: a DEC-PDP/11, a Honeywell HIS-645, and an XDS Σ 5. The application of STARAN to fast Fourier transformation, sonar post-processing, and air traffic control are all presented in [Bat74]. The application of STARAN to pattern processing is discussed in [Pot82].



1.4.1.5 Series E array module [Bat76]

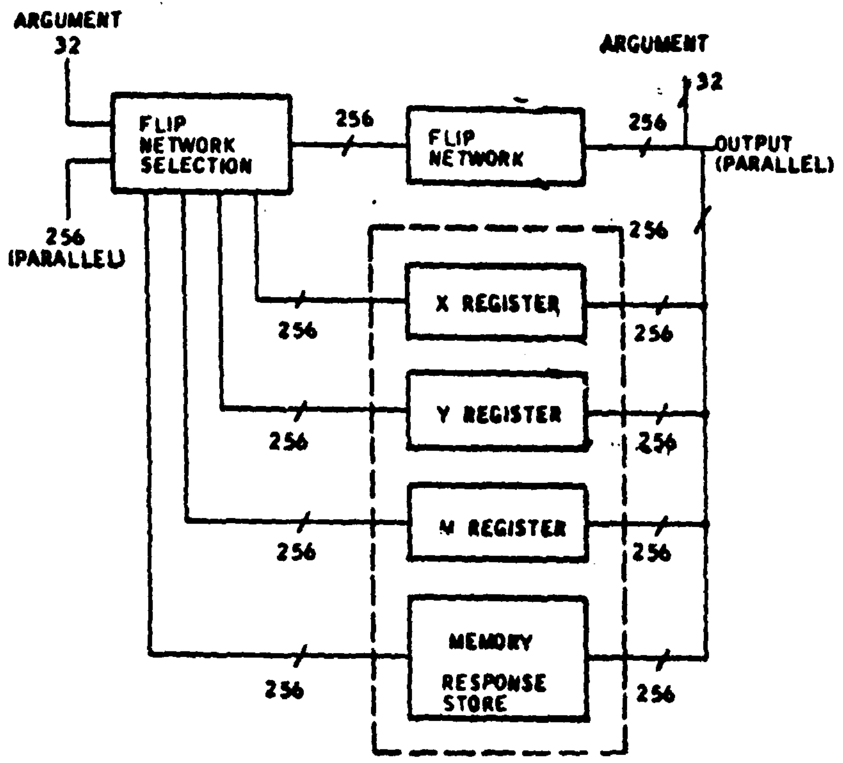


Fig. 1.4.1.6 Internal block diagram of a memory array [Thu76]

1.4.2. STARAN, DAP, and CLIP4 -- Comparisons and Contrasts

The design of STARAN is vastly different from those of DAP and CLIP4. DAP and CLIP4 have simple nearest-neighbor inter-processor connections. STARAN's permutation network, is more costly. For simple operations on binary arrays, such as erosion and dilations, the DAP and CLIP4 interconnection patterns are simple to use. However, on operations such as FFTs, STARAN can use the permutation network for performing the butterfly operations; this is not feasible using DAP and CLIP4.

CLIP4 processors can address a small amount of memory (32-bits each), DAP processors can each address 4K-bits of memory, and STARAN processors share one common memory store (some number of 256-by-256 bit planes). Thus, DAP and CLIP4 spend no time fetching and storing operands and temporary results from a global memory, except for initial loading and final unloading. Both STARAN and STARAN-E with bipolar memory have circumvented the problem of a global memory becoming a system bottleneck by using memory that is faster than the registers on either DAP or CLIP4 and that is as fast as the PE registers on STARAN. In addition, memory is accessed in such a way that there is no network contention [Bat77a]. Thus, there is no penalty for having the remote memory. The advantage of the scheme used for STARAN is that permuting data through the network data does not involve PE operations. For example, to transmit data in PE i's memory to PE i+1's memory requires a reconfiguration of the network. For both CLIP4 and DAP, this same operation would require a read from memory, a store in the network register, a read from the network register, and a store in local memory. Clearly, the scheme used for STARAN is less cumbersome and less time consuming.

The bit-serial nature of the PEs allow a great deal of flexibility of precision and representation of data. The PEs composing DAP are limited to the Boolean AND and NOT operations, making operations such as addition complex entities. The CLIP4 processor is capable of performing the Boolean AND, OR, and EXCLUSIVE OR operations; however, the architecture of the PEs facilitates addition. STARAN PEs are capable of performing Boolean AND, OR, NOT, TRUE, FALSE, and EXCLUSIVE OR. In addition, STARAN PEs can perform these operations with up to three arguments, (the X-register, the Y-register, and input from the MDA), making a wide variety of operations possible.

CLIP4 PEs have a small amount of associated memory, increasing control unit overhead for tasks that require more than 32-bits of associated memory for parameters and constants. DAP PEs have a larger available memory (4K-bits). STARAN-E avoids this problem with the 256 256-by-256 bit planes of memory.

Because of the organization of all three arrays, the method of calculating a function of a few variables and using the result to index into a table of entries is extremely difficult, as the result of the calculation must be globally transmitted by the Control Unit to each PE. According to [Ger83], this process may be faster in a sequential machine. This is, however, a fault with bit-serial processing, not these architectures.

In conclusion, three bit-serial SIMD architectures have been introduced and discussed. The bit-serial architecture lends itself well to a wide variety of processing tasks and data precisions. Bit-serial processing makes operations on words (such as floating point addition) more difficult because the operands are processed one bit at a time.

1.5. MIMD Systems

SIMD systems provide an environment where every PE performs the same operations at the same time. Conditional operations, such as: if (condition) then { A } else { B } require all PEs not satisfying the condition to be idled while the remaining PEs execute the block of code corresponding to "A." Upon their completion, the active PEs are idled while the remaining PEs execute the block of code corresponding to "B." The idling of PEs reduces the potential gains in the throughput that the system can give. For some tasks, SIMD systems may not give desirable performance. MIMD systems may, for these tasks, give an increased throughput over SIMD systems. The added flexibility of MIMD systems comes with an increased cost of overhead to perform synchronization when it is necessary. There are certain problems tha are not appropriate to the single instruction stream limitations of SIMD machines, justifying the extra cost of MIMD processing.

The architecture of a bit-serial MIMD system, Cytocomputer will be discussed in Section 1.5.1. A word-oriented system, PICAP II, will be discussed in Section 1.5.2. Two more word-oriented systems are discussed later. The CDC FP ırray and the proposed system PASM are is presented in detail in Chapter 2.

1.5.1. Cytocomputer -- A Bit-serial MIMD System

Cytocomputer was developed at the Environmental Institute of Michigan (ERIM) to perform window or cell based image processing operations. Its name comes from the Greek word "cyto," meaning cell [Ste80, Lom80]. The concept of a cell accurately describes the architecture of the Cytocomputer. With DAP and CLIP4, there is one PE per pixel. An interconnection network is required for window based operations. Cytocomputer uses internal storage in the PEs to achieve the nearest-neighbor connectivity. One PE performs a given operation for the entirety of an image, greatly reducing the number of PEs required. This significantly reduces the complexity, cost, and speed of Cytocomputer relative to CLIP4 and DAP.

The architecture of Cytocomputer is simple and is shown in Fig. 1.5.1.1 [Ste80, LoM80]. Cytocomputer consists of K (presently 80) identical stages in a pipeline. Each of the stages is a fully table-driven cellular logic machine capable of performing operations involving either four, six, or eight nearest-neighbors. In addition, each stage has a point-by-point logic function, which is capable of performing non-neighborhood operations, such as thresholding.

The nearest neighbor connectivity is achieved by loading data from the input stream (or previous stage) into a shift-register, as shown in Fig. 1.5.1.1. Only nine elements, arranged in a three-by-three square, in the shift register are accessible at one time. This defines the neighborhood function. To be consistent with [Ste80], let N be the number of elements in a row of an image. Thus, to store the necessary amount of information to process a three-by-three window, 2N+3 pixels must be stored by each stage or PE. Windows are achieved as shown in Fig. 1.5.1.2 [Ste80]. Results of calculations are passed on to the next stage for further processing. After their last use, the input data to each stage are discarded.

Each of the PEs is driven by a common clock and is capable of performing independent cell (window) operations. For each of the 80 stages, the time for a pixel operation is 640 ns. Further increases in throughput are possible by adding additional stages to the pipeline. The present speed of Cytocomputer

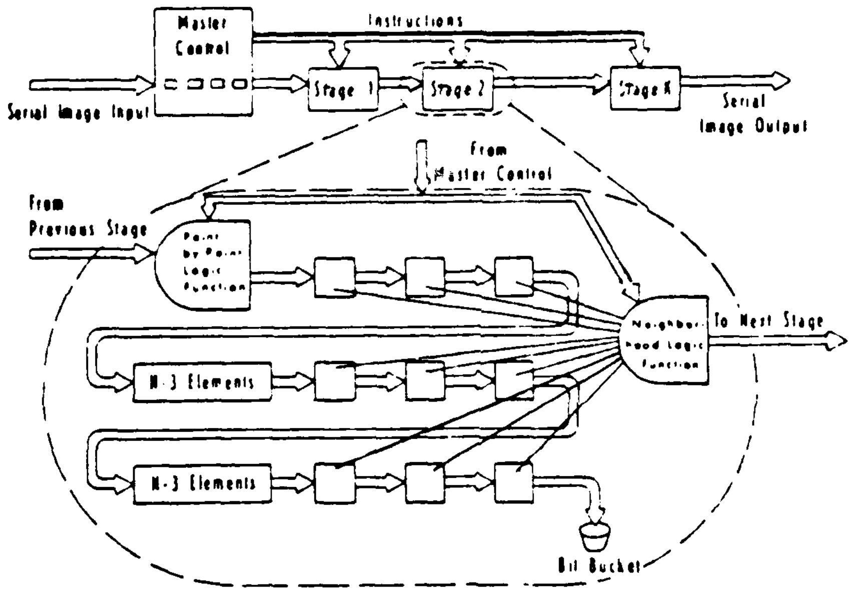


Fig. 1.5.1.1 Pipeline image processor [Ste80]

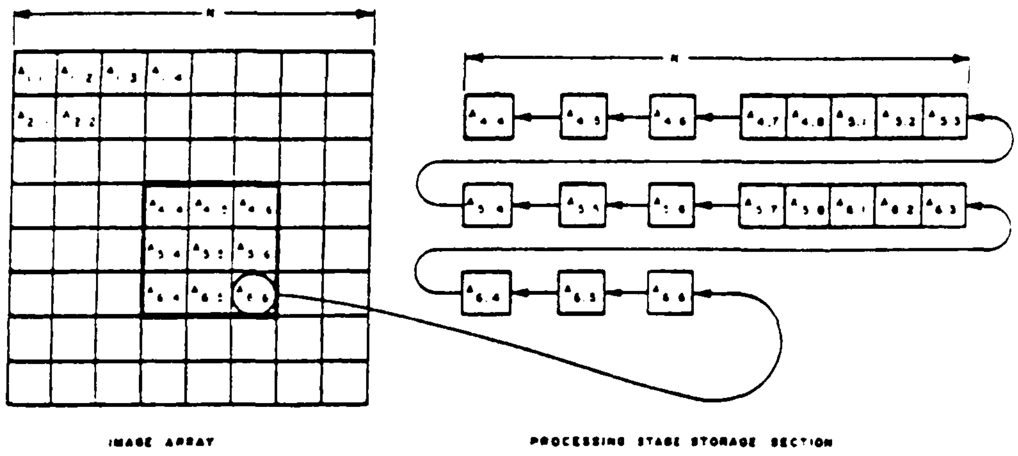


Fig. 1.5.1.2 Moving window implemented with shift register storage [Ste80]

allows it to perform many applications at a real-time rate. Applications of Cytocomputer to image processing tasks are discussed in [Ste80, PrD79, LoM80].

1.5.2. PICAP II -- A Word Oriented MIMD Machine

The **PIC**ture Array **P**rocessor (PICAP) was developed at Linkoping University in 1972. It is an MIMD system with up to sixteen word oriented processors connected to a shared picture memory through a time-shared high speed bus. The "word-size" each processor operates on is a 64-by-64 window of 4-bit integers. The architecture of PICAP II is shown in Fig. 1.5.2.1 [KrD82]. PICAP's picture memory consists of 4 Mbytes of interleaved RAM, which is sequentially addressable. With this architecture, PICAP is capable of processing multiple images simultaneously with little overhead.

Tasks that are too large for a single PICAP processor can be subdivided and placed on different processors. This offers a great deal of flexibility when applying PICAP to large image processing tasks.

For PICAP II, the shared bus is capable of transmitting 4×10^7 pixels per second, 40 times greater than that of its SIMD predecessor, PICAP I. The host computer is a PDP-11 series computer that is also used to oversee the operation of the system. PICAP has a real-time video input and monitor, which allows interactive image processing of image data. Pictures are interactively processed on PICAP through a structured high-level language called **P**icture **P**rocessing **L**anguage (**PPL**) [KrD82], which allows interactive processing, loading, and display of images. A FORTRAN interface is also available.

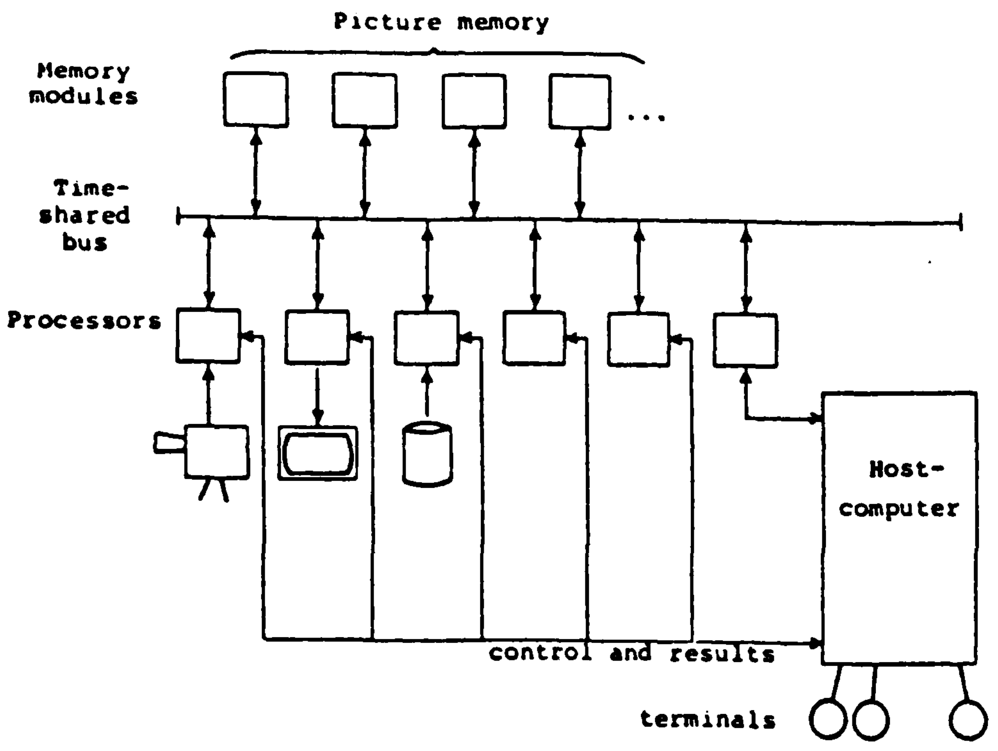


Fig. 1.5.2.1 PICAP II system architecture [Krd82]

A discussion of both PICAP I and PICAP II can be found in [KrD82]. Applications of PICAP II to image processing tasks can be found in [KrG82].

1.6. Conclusions

Several SIMD and MIMD systems for image processing to were discussed. Both word-oriented and bit-serial architectures were presented. General descriptions and applications of a wide variety of processors for image processing may be found in the following books: [Duf83], [DuL81], [FuI82], and [PrU82].

CHAPTER 2
PARALLEL PROCESSING IMPLEMENTATIONS OF A CONTEXTUAL CLASSIFIER

2.1 Introduction

Multispectral image data collected by remote sensing devices aboard aircraft and spacecraft are relatively complex data entities. Both the spatial attributes and spectral attributes of these data are known to be information bearing [SwD78], but to reduce the computation involved, most analysis efforts have focused on one or the other. Characteristic spatial features include, for example, shape, texture, and structural relationships. Useful research has been accomplished in the direction of incorporating spatial information into the data analysis process (e.g., [HaS73], [KeL76], [WeD76]).

The "class" associated with a given pixel is not independent of the classes of adjacent pixels. Stated in terms of a statistical classification framework, there may be a better chance of correctly classifying a given pixel, if in addition to the spectral measurements associated with the pixel itself, the measurements and/or classifications of its "neighbors" are considered as well. The image can be considered to be a two-dimensional random process incorporated into the classification strategy. This is the objective of "contextual classifiers" [WeS71], in which a form of compound decision theory is employed through the use of a statistical characterization of context. Recent investigations have demonstrated the effectiveness of a contextual classifier that combines spatial and spectral information by exploiting the tendency of certain ground-cover classes to occur more frequently in some spatial contexts than in others [SwS80],[SwV81],[TiS81],[WeS71].

The practical utilization of this contextual classifier in remote sensing has awaited the solution of two key problems: (1) lack of an effective method for characterizing and extracting contextual information in multispectral remote sensing imagery, and (2) the need to reduce the execution time of the very computation-intensive contextual classification algorithm. The first of these problems has been solved by development of an unbiased estimation procedure which provides a good characterization of the contextual information without requiring exorbitant amounts of classifier training data ("ground truth") [TiS81]. Although the resulting improvement in classification accuracy is significant compared to conventional no-context statistical classification methods, the practicality of the contextual classifier depends on the solution of the second problem, the subject of this chapter.

A reduction in the execution time of classification algorithms such as the contextual classifier (and even much simpler algorithms used for remote sensing data analysis) can be achieved through the use of parallelism. There are several types of parallel processing systems. An **SIMD** (**S**ingle **I**nstruction stream -- **M**ultiple **D**ata stream) machine [Fly66] typically consists of a control unit, N processors, N memory modules, and an interconnection network. The control unit broadcasts instructions to all of the processors, and all active (enabled) processors execute the same instruction at the same time. Each active processor executes the instruction on data in its own associated memory module. The interconnection network provides a communications facility for the processors and memory modules. An **MIMD** (*Multiple Instruction stream -- Multiple* **D**ata stream) machine [Fly66] typically consists of N processors and N memory modules, where each processor can follow an independent instruction stream. As with SIMD architecture, there is a multiple data stream and an interconnection network. CDC Flexible Processor (FP) systems are MIMD architectures that have been built [CDC77a], [CDC77b]. PASM is a proposed partitionable SIMD/MIMD multimicroprocessor system for image processing and pattern recognition [SiS81]. For this application, the use of PASM in the SIMD mode of operation will be considered.

Maximum likelihood classification [SwD78], often used in remote sensing, classifies each pixel independently of all others. Using either the SIMD or MIMD mode of parallelism, the image can be subdivided among the processors, each processor classifying its own subimage. Thus, N processors would be able to execute maximum likelihood classification approximately N times faster than one processor of the same type. However, parallel implementations of contextual classifiers are, in general, not so straightforward, due to the use of neighborhood information. The way in which parallel machines such as the CDC FP system and PASM perform contextual classifications is examined in the following sections.

Section 2.2 briefly describes contextual classification and gives a uniprocessor algorithm for performing it. The implementation of a contextual classification algorithm on an FP system and a comparison of the timings obtained on an FP system simulator to those obtained on a PDP-11/70 are discussed in Section 2.3. In Section 2.4, the way in which PASM can be applied to contextual classification is considered.

2.2. Contextual Classification

2.2.1. Definitions

The image data to be classified are assumed to be a two-dimensional I-by-J array of multivariate pixels. Associated with the pixel at "row i" and "column j" is the multivariate measurement n-vector $X_{ij} \epsilon R^n$ and the true class of the pixel $\Theta_{ij} \epsilon \Omega = \{\omega_1, \ldots, \omega_C\}$ The measurement vectors have **class-conditional densities** $f(X|\omega_k)$, $k = 1,2,\ldots,C$, and are assumed to be class-conditionally independent. The objective is to classify the pixels in the array.

In order to incorporate contextual information into the classification process, when each pixel is to be classified, p−1 of its neighbors are also examined. This neighborhood, including the pixel to be classified, will be referred to as the **p-array.** To classify each pixel, the contextual classifier computes the probability of the given observed pixel being in class k by also considering the measurement vectors (values) observed for the neighbor pixels in the p-array. Specifically, for each pixel, for each class in Ω, a discriminant function g is calculated. The pixel is assigned to the class for which g is the greatest. Each value of g is computed as a weighted sum of the product of probabilities based on the pixels in the neighborhood. This is described below mathematically for pixel (i,j) being in class ω_k. (The description is followed by an example to clarify the notation used. Further details may be found in [SwS80],[SwV81].)

$$g_k(\underline{X}_{ij}) = \sum_{\substack{\underline{\Theta}_{ij} \in \Omega^p, \\ \Theta_{ij} = \omega_k}} \left[\left[\prod_{\gamma=1}^{p} f(X_\gamma | \Theta_\gamma) \right] G^p(\underline{\Theta}_{ij}) \right]$$

where

$X_\gamma \in \underline{X}_{ij}$ is the measurement vector from the γ^{th} pixel in the p-array (for pixel (i,j))

$\Theta_\gamma \in \underline{\Theta}_{ij}$ is the class of the γth pixel in the p-array (for pixel (i,j))

$f(X_\gamma | \Theta_\gamma)$ is the class-conditional density of X_γ given that the γth pixel is from class Θ_γ

$G^p(\underline{\Theta}_{ij}) = G(\Theta_1, \Theta_2, \ldots, \Theta_p)$ is the a priori probability of observing the p-array $\Theta_1, \Theta_2, \ldots, \Theta_p$

Within the p-array, the pixel locations may be numbered in any convenient, but fixed order. The joint probability distribution G^p is referred to as the **context distribution.** The class-conditional density of pixel measurement vector X given that the pixel is from class k is:

$$f(X | \omega_k) = e^{\frac{-[\log|\Sigma_k| + (X-m_k)^T \Sigma_k^{-1} (X-m_k)]}{2}}$$

where the measurement vector for each pixel is of size four, Σ_k^{-1} is the inverse of the covariance matrix for class k (four-by-four matrix), m_k is the mean vector for class k (size four vector), "T" indicates the transpose, "log" is the natural logarithm, and $|\Sigma_k|$ is the determinant of the covariance matrix. This is the same function as used for the maximum likelihood classification [SwD78].

Consider, as an example, the horizontally linear neighborhood shown in Fig. 2.2.1.1(a), where pixel (i,j) is the middle pixel, and assume there are two possible classes: $\Omega = \{a,b\}$. Then the discriminant function for class b is explicitly:

$$\begin{aligned} g_b(X_{ij}) = {} & f(X_1|a)f(X_2|b)f(X_3|a)G(a,b,a) \\ & + f(X_1|a)f(X_2|b)f(X_3|b)G(a,b,b) \\ & + f(X_1|b)f(X_2|b)f(X_3|a)G(b,b,a) \\ & + f(X_1|b)f(X_2|b)f(X_3|b)G(b,b,b) \end{aligned}$$

After computing the discriminant functions of g_a and g_b for pixel (i,j), pixel (i.j) is assigned to the class which has the larger discriminant value. (Edge pixels of the image not having the appropriate p−1 neighbors are not classified.)

Consider the case where there is a non-linear three-by-three context array (neighborhood), as shown in Fig. 2.2.1.1(b). Here, for each g, with C classes, there are C^8 product terms with nine factors in each term. In general, for each g, there are C^{p-1} product terms, each term having p+1 factors. In the LANDSAT data used in the testing described in [TiS81], the percentage of non-zero G^9's was about 1% (based on a size nine neighborhood and 14 classes), so to conserve space and to increase throughput, only non-zero $G^{p\prime}$s are stored. This technique will be discussed in later sections. All of the calculations are done using floating point data.

X_1	X_2	X_3

(i,j-1)	(i,j)	(i,j+1)

(a) Linear one-by-three neighborhood (p=2)

X_1	X_2	X_3
X_4	X_5	X_6
X_7	X_8	X_9

(i-1,j-1)	(i-1,j)	(i-1,j+1)
(i,j-1)	(i,j)	(i,j+1)
(i+1,j-1)	(i+1,j)	(i+1,j+1)

(b) Non-linear three-by-three neighborhood (p=8)

Fig. 2.2.1.1 Linear neighborhoods

2.2.2. Uniprocessor Algorithm

The algorithm shown in Fig. 2.2.2.1 is a uniprocessor implementation of the size three contextual classifier. $f(X|\Theta_\gamma)$ is independent of the position within a window, and thus does not change when a window is moved. This algorithm is consistent with the theory presented above; however, to minimize execution time, an array (called "hold" is used to store "compf" values. Since $f(X|\Theta_\gamma)$ is required for all windows that contain pixel X, redundant calculations may be eliminated by storing $f(X|\Theta_\gamma)$ in a temporary array. The stored $f(X|\Theta_\gamma)$ is discarded when pixel X will no longer appear in any windows. For the uniprocessor implementation, the temporary array is called "hold."

Let "hold(m,k)" be a two-dimensional array of size three-by-C, i.e., $0 \leq m \leq 2$ and $1 \leq k \leq C$. "hold(cr,k)" (statement S5) is a vector of length C containing the class-conditional density values ("compf" values, statement S3) for the pixel (i,j) ("cr" is an abbreviation for center). "hold(lt,k)" (statement S4) and "hold(rt,k)" (statement S6) are the analogous vectors for the pixel (i,j−1) (the left neighbor) and pixel (i,j+1) (the right neighbor), respectively. By using this array to save the class-conditional densities, each density (for a given pixel and class) is calculated only once.

The algorithm calculates the class-conditional densities for the first three columns each time a new row is to be classified and stores them in "hold." (statement S3). Each time a new pixel in a given row is to be classified (statement S7), the pointers to these values in "hold" are updated (statement S17). In particular, the data in "lt" is disposed of, "lt" is updated to point to the data previously pointed to by "cr", "cr" points to the data previously pointed to by "rt", and "rt" points to the newly calculated data (statement S17) for the incoming pixel.

```
Main Loop
for i = 0 to I-1 do /* row index */
    for k = 1 to C do /* for each class */
        for m = 0 to 2 do hold(m,k) = compf(i,m,k) /* cols.0-2 */
    lt = 0 /* hold(lt,k,) is left neighbor */
    cr = 1 /* hold(cr,k) is pixel being classified */
    rt = 2 /* hold(rt,k) is right neighbor */
    for j = 1 to J-2 do /* column index */
        value = -1; class = -1 /* max "g" and class */
        for k = 1 to C do /* for each class */
            current = g(lt,cr,rt,k)
            if current > value /* compare with max */
                then value = current; class = k
        print pixel(i,j) is classified as "class"
        if j ≠ J-2 then /* update hold pointers */
            tp = lt; lt = cr; cr = rt; rt = tp
            for k = 1 to C do /* compf's for next col */
                hold(rt,k) = compf(i,j+2,k)

Discriminant Function Calculation
fuction g(lt,cr,rt,k) /* for pixel cr, class k */
sum = 0 /* initialize sum, used to accumulate g */
for r = 1 to C do /* all classes for pixel (i,j-1) */
    for q = 1 to C do /* all classes for pixel (i,j+1) */
        if G(r,k,q) ≠ 0 /* do not multiply if G = 0 */
            then sum = hold(lt,r) * hold(cr,k)
                          * hold(rt,q) * G(r,k,q) + sum
return (sum) /* sum contains value of g(lt,cr,rt,k) */

Class-Conditional Density Calculation
function compf(a,b,k) /* for pixel (a,b), class k *
x = A(a,b) /* x is the pixel (a,b) measurement vector */
expo = $-[\log|\Sigma_k| + (x-m_k)^T\Sigma_k^{-1}(x-m_k)]/2$
return ($e^{expo}$) /* return value of f(A(a,b)|j) */
```

Fig. 2.2.2.1 Uniprocessor implementation of size three contextual classifier algorithm (p=2)

The complexity of the algorithm is proportional to $I*J*C^3$ assignments, multiplications, and additions, and $I*J*C$ "compf" calculations. Typically, $10 \leq C \leq 60$ for the analysis of LANDSAT data.

The algorithm can be extended for a non-linear contextual classifier with a neighborhood of size nine (as shown in Fig. 2.2.1.1(b)). The complexity of the algorithm would have growth proportional to $I*J*C^9$ assignments, multiplications, and additions. The number of "compf" calculations would still be $I*J*C$. In this case, "hold" would be a $(2*J+3)$-by-C array (assuming the neighborhood window moves along rows). Fig. 2.2.2.2 shows the pixels whose "compf" values are stored in the "hold" array. The $2*J+3$ pixels whose "compf" values are stored in "hold" are chosen to make it unnecessary to perform redundant "compf" calculations. In general, when classifying pixel (i,j), "hold" has the "compf" values for pixels $j-1$ to $J-1$ of row $i-1$, pixels 0 to $J-1$ (all) of row i, and pixels 0 to $j+1$ of row $i+1$. After the classification of pixel (i,j), the values for $(i+1,j+2)$ are added and the values for $(i-1,j-1)$ are removed. When the pixels on a new row are to be classified, call it i', then the values for pixels $(i'-2,J-3)$, $(i'-2,J-2)$, and $(i'-2,J-1)$ are removed and the values for $(i'+1,0)$, $(i'+1,1)$, and $(i'+1,2)$ are added. (This assumes row i' is classified after $i'-1$.) Given this, the rest of transforming the algorithm for the size nine square neighborhood case is straightforward.

In summary, the uniprocessor one-by-three algorithm was presented. The extension to the three-by-three case was discussed. Extension to other size and shape neighborhoods is similar. The next two sections discuss parallel implementations using FPs and PASM respectively.

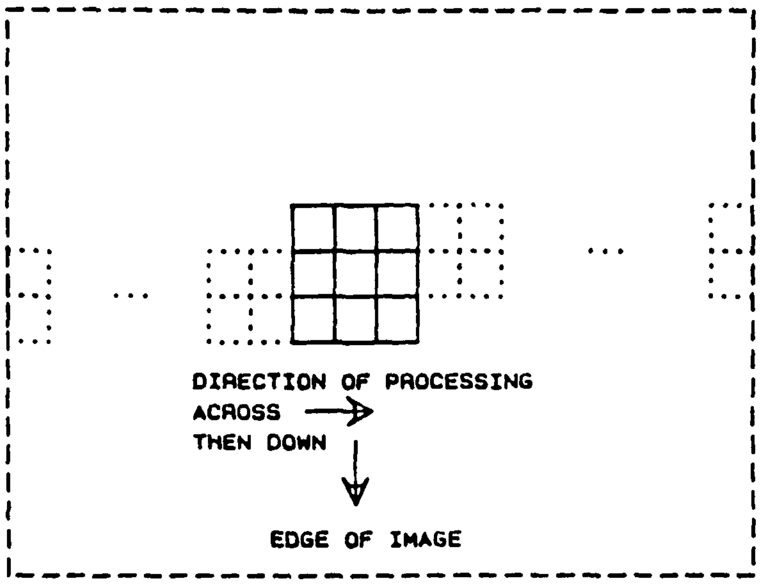


Fig. 2.2.2.2 Pixels whose "compf" values are stored in "hold" array

2.3. MIMD Implementation on the CDC Flexible Processor System

2.3.1. Flexible Processor System

The Control Data Corporation Flexible Processor (FP) system is a multiprocessor system which has been recommended for use in remote sensing. The basic components of an FP are shown in Fig. 2.3.1.1. There can be up to 16 FPs linked together, providing much parallelism at the processor level. The FPs can communicate among themselves through a high-speed ring or shared bulk memory. A possible FP system configuration is presented in Fig. 2.3.1.2.

The instruction cycle time of each FP is 125 nsecs. An FP is programmed in micro-assembly language, allowing parallelism at the instruction level. For example, it is possible to conditionally increment an index register, execute a program jump, multiply two 8-bit integers, and add two 32-bit integers -- all simultaneously. This type of operational overlap, in conjunction with the capability to use up to 16 FPs in parallel, greatly increases the speed of the FP system.

The following list summarizes the important architectural features of an FP:

User microprogrammable.

Dual 16-bit internal bus system.

Able to operate with either 16- or 32-bit words.

125 nsec. instruction cycle time.

125 nsec. time to add two 32-bit integers.

250 nsec. time to multiply two 8-bit integers.

Register files of over 8000 16-bit words.

60 nsec. read/write time for register files.

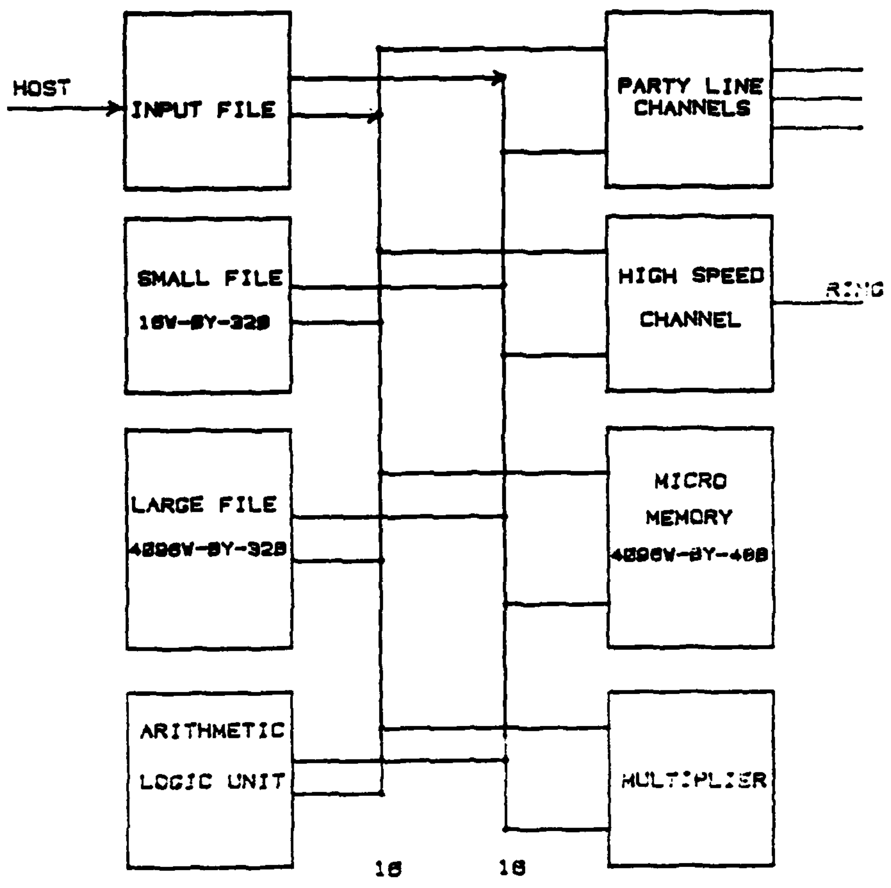


Fig. 2.3.1.1 Components of an FP ([CDC77a],[CDC77b])

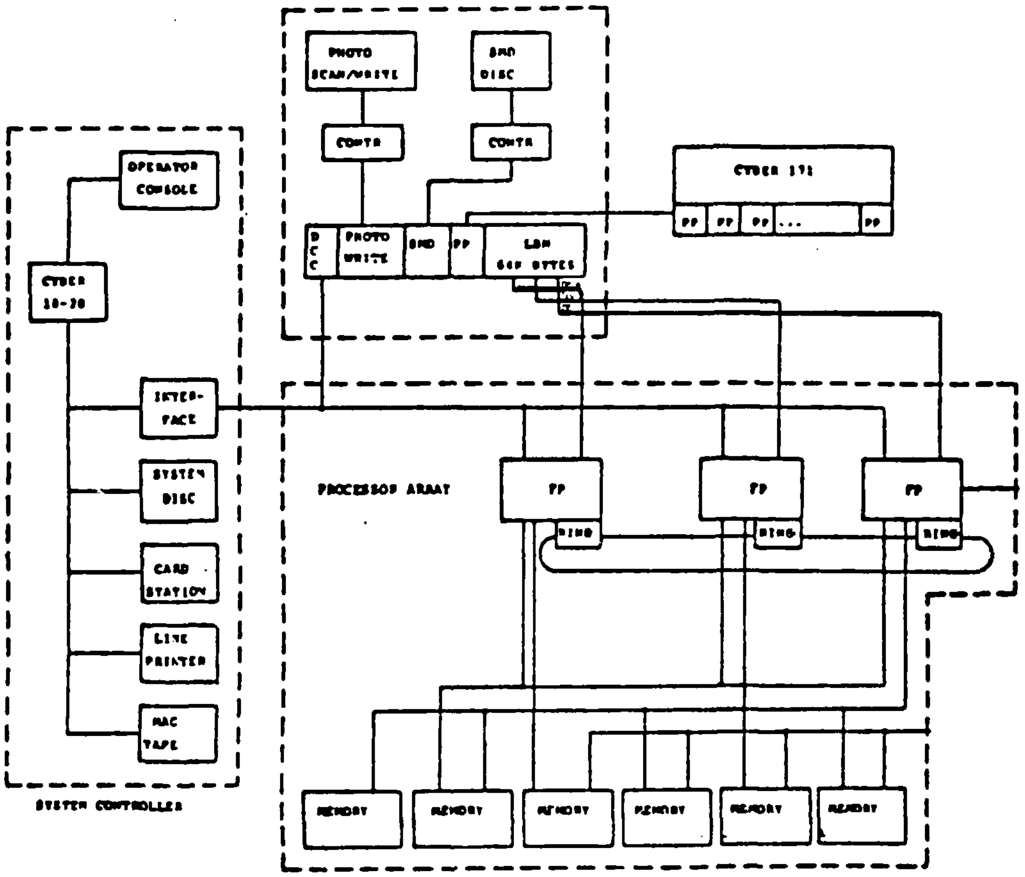


Fig. 2.3.1.2 A potential FP system configuration ([CDC77a],[CDC77b])

Up to 16 banks of 250 nsec. bulk memory (each bank holds 64K words).

In order to debug, verify and time FP algorithms, a simulator and an assembler were developed for a system of up to 16 FPs. The experience gained through the use of the simulator has made evident the following advantages and disadvantages of the FP system.

Advantages:

Multiple processors (up to 16)

User microprogrammable -- parallelism at the instruction level

Connection ring for inter-FP communications

Shared bulk memory units

Separate arithmetic logic unit and hardware multiply

Disadvantages:

No floating point hardware

Micro-assembly language -- difficult to program

Program memory limited to 4K microinstructions

Both the simulator and the assembler are designed to operate under the UNIX operating system. They are described in [SmS80]. More details about the FP system can be found in [SmS80],[SwS80],[CDC77a],[CDC77b].

2.3.2. Linear Contextual Classifiers

Consider using an N ($\leq$16) FP system to implement the contextual classifier based on a horizontally linear neighborhood of size three (Fig. 2.2.1.1(a)). Divide the I-by-J image into subimages of I/N rows J pixels long, as shown in Fig. 2.3.2.1. This method of dividing the image is called **striping.** Assign each subimage to a different FP. The entire neighborhood of each pixel is included in its subimage. No interaction between FPs is needed, i.e., each FP can process its subimage independently. A perfect factor of N improvement speedup over a single FP occurs if I is a multiple of N. The degradation in performance that arises when I is not a multiple of N is less than 1% for large images [SwS80].

An FP micro-assembly language version of the algorithm stated in Fig. 2.2.2.1 was written. *Because each FP is microprogrammable, determining* program correctness and analyzing the execution time are done through the use of the micro-assembler and simulator. All floating point operations are done in software. Mantissa normalization of all floating point operands gives rise to a variation in the overall execution time per pixel. This variation can be as much as 10:1 [SmS80].

Each pixel measurement vector consisted of four 32-bit floating point representations of 8-bit integers; the input data were converted to floating point notation prior to the execution of the classifier. This conversion is not included in either the FP or comparative PDP-11 timings. Covariance matrices consisted of ten 32-bit floating point numbers. Further, 32-bit floating point numbers were used to represent the logarithms of the determinants of the covariance matrices and the a priori probabilities. The pixel measurement vectors, covariance matrices, logarithms of the determinants of the covariance

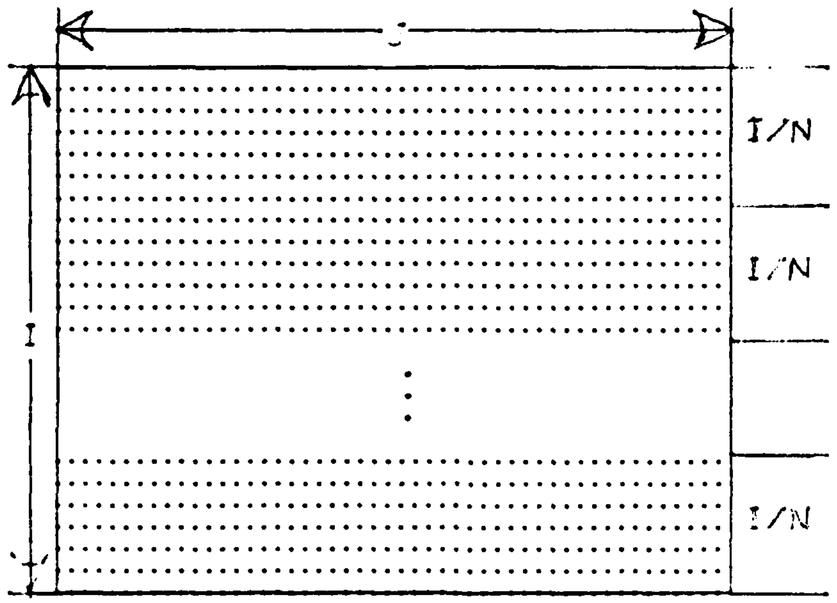


Fig. 2.3.2.1 Striping method of dividing an I-by-J image among N FPs

matrices, a priori probabilities, and a temporary variable array are all stored in the "large file" (see Fig. 2.3.1.1). Thus, in this case, each FP has all the information it needs for performing the classification on its subimage stored in its register file and no "bulk memory" accesses are required.

If the number of non-zero a priori probabilities is small (less than 50%), and the contextual information (configuration of classes) associated with each G^p can be stored in the space of one floating point number (32 bits), then any algorithm that stores all a priori probabilities will waste memory space. This is the case in the LANDSAT data used for this experiment. Each G^p is stored as two 32-bit quantities. The first 32-bit quantity contains information about the class of each pixel within the p-array. For example, if G(3,3,2) is non-zero, the word preceding it is a representation (catenation) of 3,3, and 2. This allows $\lfloor 32/p \rfloor$ bits per class, i.e., up to $2^{\lfloor 32/p \rfloor}$ classes. (Thus, for the size three neighborhood being considered, C can be as large as 1024.) The second 32 bits is the value of the G^p itself. Only the non-zero G^ps are stored, so only the non-zero G^ps affect the computation time.

For larger windows (larger p), it is possible that $2^{\lfloor 32/p \rfloor}$ will not be large enough to include all possible classes. If this occurs, one or two additional 32-bit words can be used to store the class information about the p-array. In such cases, the non-zero G^ps would have to be less than 30% or 25% respectively in order for this scheme not to require additional space. As stated previously, based on an analysis performed, the percentage of non-zero G^ps is much smaller than this.

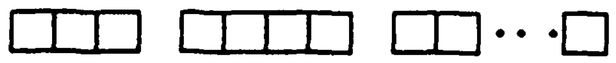
When this memory arrangement is employed, the needed class information is obtained by masking off the desired bits and shifting the result right (producing a number between 0 and $2^{\lfloor 32k/p \rfloor}-1$, where k is a number between 1 and 3 depending on the number of words used to store the class information.) If the desired information does not cross a word boundary, this operation will require 3p steps per non-zero G^p (load, logical and, shift), otherwise it will require 7p steps per non-zero G^p (load, logical and, shift, load, logical and, add, shift.) Consider, instead, using the straight forward approach of storing all G^ps, both zero and non-zero. For a window of size p, a p-element vector (containing elements between 0 and C−1) is required in order to create the p^C possible window configurations. Incrementing an index value requires four operations consisting of: storing the address of the index in the large file address register, reading the index from the large file, incrementing the index, and the storing the new value in the large file. This is done each time an index is incremented. In addition, each time an index is incremented, it must be compared to the C. If it equals C, it should be set to 0 and the next index incremented. 2p operations are required (store address of index in large file address register and store initial value of index) to initialize the indices. Thus, the time required to handle the indices for this scheme is $2p+5\sum_{i=1}^{p}(C^i)$ steps per G^p (zero or non-zero.) Thus, the proposed algorithm will not only be more space efficient, but it will run faster.

For the purposes of testing the FP implementation of the one-by-three linear contextual classifier program, measurement vectors from 30 rows of 16 pixels were classified. The data set consisted of a four-class subset of the LANDSAT data used in [SwV81]. To provide a basis for comparison, a similar contextual classifier was run on a PDP-11/70 over the same test data. It was found that lack of exponent range in the 11/70 floating point hardware required extra handling. FP floating point algorithms are implemented in the software, so a 14-bit exponent was used to overcome this problem. A description of the floating point software is available in [SmS80]. The FP "e" calculations are based on those in [Har68]. Twenty non-zero G^ps were chosen for the benchmark tests. Running under the above constraints, the single FP classifier took .035 secs./pixel, while the PDP-11/70 required .050 secs./pixel, a 30% improvement.

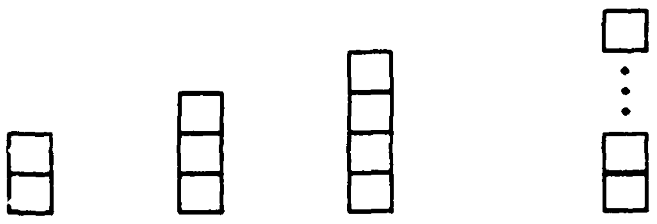
Using .05 secs. per pixel as the PDP processing time and .035 secs. per pixel as the single FP processing time, a 16 FP configuration would perform contextual classifications at a rate of 457 pixels per sec., as opposed to 20 pixels per sec. for a single PDP-11/70. There are, of course, cost differences between these two systems; however, the purpose here is to show the gains made possible by a multiprocessor FP system. In general, different size horizontally linear (Fig. 2.3.2.2(a)), vertically linear (Fig. 2.3.2.2(b)), and diagonally linear neighborhoods (Fig. 2.3.2.2(c)) of various sizes can be processed in a manner similar to that for the horizontally linear neighborhood of size three [SwS80].

2.3.3. Non-linear Contextual Classifiers

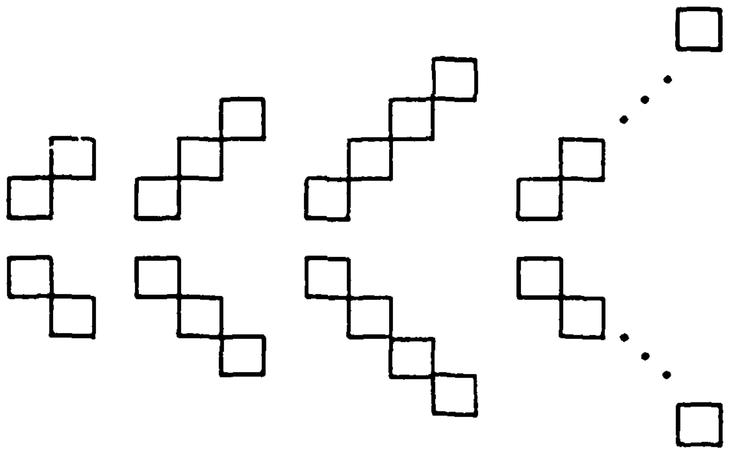
Consider non-linear neighborhoods, that is, neighborhoods which do not fit into one of the linear classes. For example, all of the neighborhoods in Fig. 2.3.3.1 are non-linear. It can be shown that there is no way to partition an image into N (not necessarily equal) sections such that a contextual classifier using a non-linear neighborhood can be performed without data transfers among FPs [SwS80]. The specific non-linear case under consideration is the three-by-three non-linear neighborhood, shown in Fig. 2.2.1.1(b). First, the



(a) Horizontally linear neighborhoods



(b) Vertically linear neighborhoods



(c) Diagonally linear neighborhoods

Fig. 2.3.2.2 Linear neighborhoods

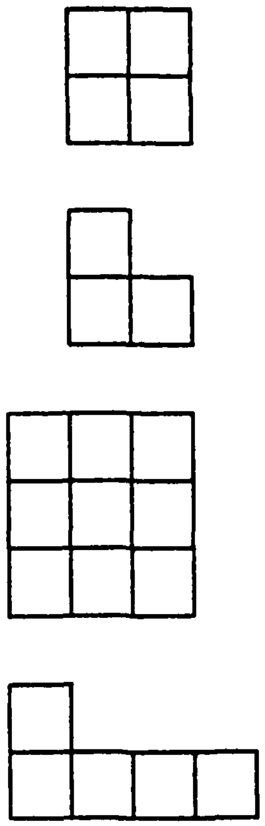


Fig. 2.3.3.1 Non-linear neighborhoods

single FP timings are considered, then the timings for an system of N FPs are considered.

The eight-nearest neighbor contextual classifier is similar to the previously described linear case. Differences arise in the calculation of the discriminant function (discussed in Section 2.2.1), the method of updating the "hold" data for a given window (discussed in Section 2.2.2), and the method of data storage (discussed below).

Timings run from LANDSAT data from [SwV81] show that, on the average, the FP implementation of the four-class, size nine square neighborhood contextual classifier with all data entries and a priori information stored in the large file requires .137 secs./pixel. A PDP-11/70 implementation of the same algorithm requires .154 secs./pixel. Thus, there is an 11% improvement. The improvement is not as much for this case as in the size three horizontally linear case because the FP performs floating point operations in the software. The more terms in the product term, the more time the FP will spend normalizing intermediate results. Tests for the 11/70 were run with 50 non-zero G^ps and four spectral classes on 52 lines of 16 pixels. A 30-line-by-16-pixel subset of the above image was used to derive the FP timings for a 52-line image. Pixels on the top and bottom line of an image are not classified, and thus do not appear in the number of classified pixels. As a result, for the first and last rows of an image, the classifier must calculate the class conditional probabilities for these pixels without ever classifying them. Therefore, the results are slightly biased in favor of the 11/70 implementation. Once again, only the non-zero G^ps are stored, so only the non-zero G^ps affect computation time.

Using .154 secs. per pixel as the PDP processing time and .137 secs. per pixel as the single FP processing time, a 16 FP system would perform contextual classifications at a rate of approximately 116 pixels per sec., as opposed to the 6 pixels per sec. rate of a single PDP-11/70. This assumes, however, that all needed data are stored in the large file, a somewhat unrealistic assumption. The use of the bulk memories for storing and sharing data is discussed in the next three sections.

2.3.4. Processing of Images with Large Numbers of G^ps

If the a priori probabilities are too large to fit in the register files, bulk memory can be used to store the overflow G^ps. The width of the bulk memory is 16 bits. Each G^p is composed of either two, three, or four 32-bit quantities. One contains the G^p itself, while the rest is the contextual information associated with a given pixel (see 2.3.2). A 64-bit G^p can be accessed with four reads from bulk memory, while a 96-bit read can be accessed with six reads, and a 128-bit G^p can be accessed with eight reads. One of the special features associated with an FP is that every time a read from bulk memory is performed, the pointer to bulk memory is automatically incremented [CDC77a]. A read from bulk memory is accomplished in two steps [CDC77a], [CDC77b]. First the read must be initialized and second (after .250μ-secs.) the data must be read from the bulk memory [CDC77a],[CDC77b]. On the surface, it would appear that a 16-bit read requires four clock cycles; however, this is not the case. The read can be initialized in parallel with other operations; thus no time is lost due to the initialization. An FP can wait for the data or it can execute other instructions in the meantime. Thus, the total cost of a read from bulk memory is one instruction cycle per 16 bits. The cost, then, of accessing a G^P and its corresponding context configuration from the bulk memory is $2 + 2k$ instruction cycles, or 250μ–secs + $k \times 250\mu$–secs, where k is the number of words used to store the context information. To perform the corresponding operation from the large file requires $.250\mu$–secs., or two instruction cycles.

As an example, use the benchmark eight-nearest neighbor non-linear context array, where k=1. Allow all 50 of the G^Ps to be stored in bulk memory. The total time spent accessing the G^Ps is:

$$.500\frac{\mu\text{–secs}}{G^P} \times \frac{\#\text{ of non–zero } G^P(=50)}{\text{pixel}} = 25\frac{\mu\text{–secs.}}{\text{pixel}}$$

Only half of this time, however, represents additional processing time over fetching the G^P and its corresponding context array from the large file. Thus, the additional processing time required to process a G^P stored in bulk memory is 12.5μ–secs per pixel. When this is compared to the 137,000 μ–secs./pixel required for classification, this time represents a negligible cost. In the cases where there are more classes, this ratio will become more negligible.

2.3.5. Processing of Images in Bulk Memory

If an image is small, data vectors may be stored in the large file. This was the method used to acquire the timings presented. For actual images, however, the large file is too small to hold the image data. Pixel measurement vectors can be stored in bulk memory. There is, however, an additional cost associated with reading pixel measurement vectors from bulk memory. Pixel data is represented as a one-by-four vector of 32-bit floating point numbers. It was earlier stated that a 16-bit read from bulk memory requires the same amount of time as a 32-bit read from the large file. Thus, reading a 32-bit number from bulk memory will require twice as much time as a corresponding read from the large file. Reading a data vector from the large file will require four instruction cycles, or .5μ-secs./pixel. Reading the same data from bulk memory will require an additional processing time of four instruction cycles, or .5μ-secs./pixel. This is minimal when compared with the 137,000μ-secs./pixel processing time associated with the eight nearest-neighbor contextual classifier.

2.3.6. A 16 FP System

Consider the problem of using N ($\leq$16) FPs together to do contextual classification with a square size nine neighborhood. Assume the image data is stored in the bulk memories. The approach taken is to divide the image among the FPs using the "striping" method (Fig. 2.3.2.1). Each FP classifies the pixels in its own subimage. Because the p-array is non-linear, FPs will have to communicate to share subimage edge data [SwS80]. For example, to classify the bottom row of FP 0's subimage, information about the pixels in the top row of FP 1's subimage is needed (i.e., the neighborhood window crosses subimages boundaries). Thus, some way to achieve this sharing is necessary.

The speed at which the contextual classifier runs depends on the floating point algorithms which are implemented in the software. This can cause a bottleneck in the processing if one FP is required to wait for another. Synchronization can require large amounts of time if the full 16 processor array is used, since at each step, the slowest FP will determine the execution time. Thus, asynchronous processing at the instruction level is necessary.

An FP is capable of addressing up to three channels of 16-by-128K bytes of bulk memory each [CDC77a],[CDC77b]. The sharing of bulk memory is a scheme that can be used for transferring data among FPs. One possible implementation is shown in Fig. 2.3.6.1. Bus 0 of FP i will be shared with FP i−1, while bus 1 will be local to FP i, and bus 2 will be shared with FP i+1. An FP will be allowed to address only half of its L memory banks at one time. This is done to facilitate double buffering. The other L/2 memory banks will be accessible by the host. This allows the FP to classify one image while the host unloads and stores the results of the previous classification and then loads the next image to be processed.

Assume each FP will classify the pixels in I/N rows (Fig. 2.3.2.1). If border areas are stored in the shared memory banks, a processor will begin processing in banks of bus 1. Processing will continue through half of the L/2 banks in bus 1 to bank 0 on bus 2. After all the data in the banks on data bus 2 have been processed, processing will continue to the banks on bus 3.

Allowing 25% of FP i's data to be stored in the shared banks on bus 1, 50% of the data to be stored in the local banks on bus 2, and 25% of the data to be stored in the shared banks on bus 3, no contention will occur. Consider that for processor i to "catch up" with processor i+1, processor i will have to process more than 75% of its data in the time that it takes processor i+1 to process 25% of its data. Thus, contention is not a problem.

When an image is divided by the striping scheme, all non-linear windows will require FPs to share data. In particular, for the case of an A-by-A window, (A−1) rows of "compf"/pixel values must be commonly accessible by adjacent FPs. This is shown in Fig. 2.3.6.2. Assuming that an FP classifies all pixels in its subimage, that the pixel to be classified is in the middle of the window,

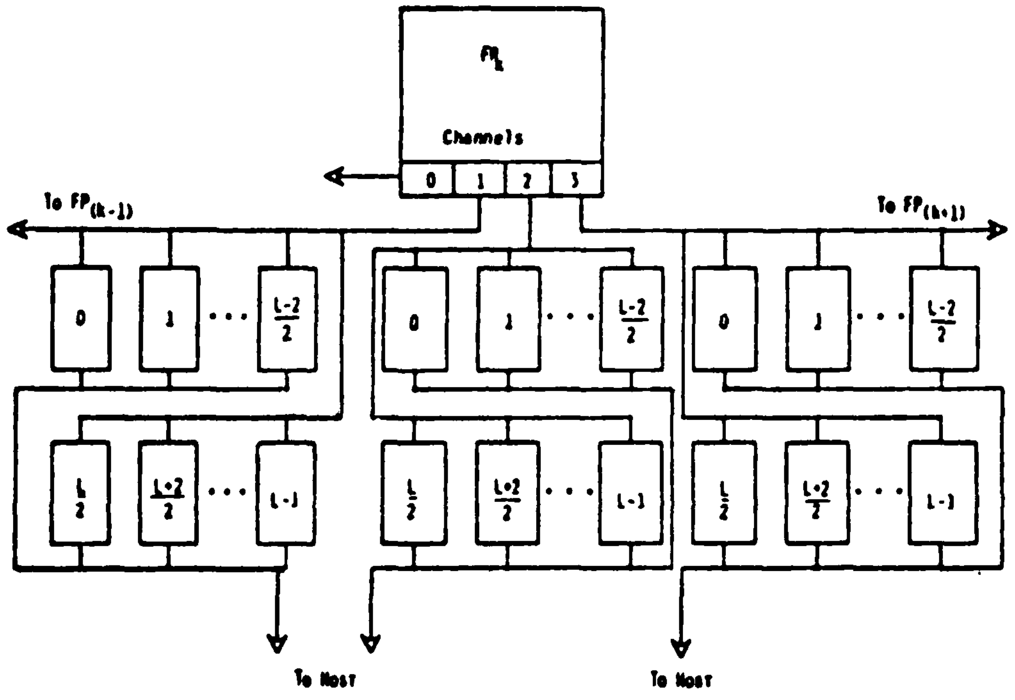


Fig. 2.3.6.1 A potential FP architecture for image processing

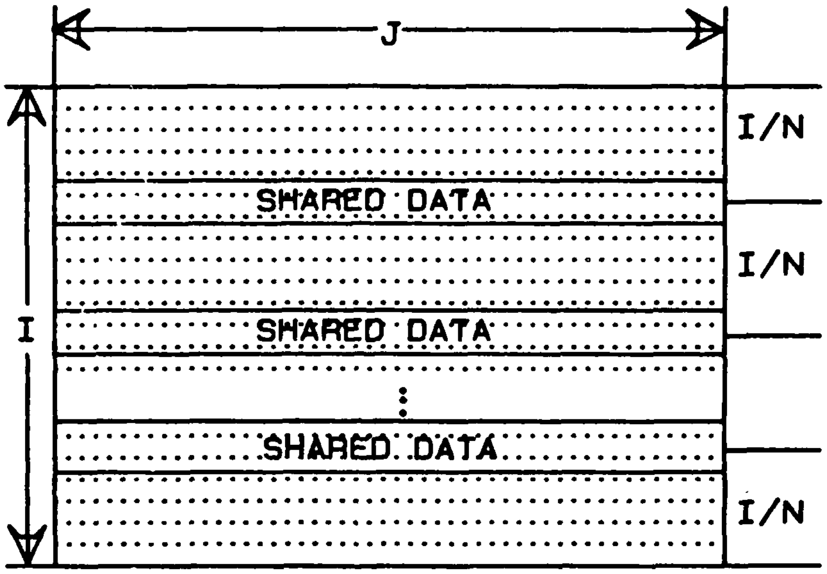


Fig. 2.3.6.2 Data required for classification of non-linear windows

and that A is odd, FP i (i>0) will require the (A−1)/2 bottom rows of data from the subimage of FP i−1 to classify the top row of its subimage (in addition to the (A−1)/2 rows of data from its own subimage). In addition, FP i will require the (A−1)/2 top rows of data from the subimage of FP i+1 to classify the last row in its subimage. Once the "compf" values for a given pixel are calculated, they do not change. Thus, if FP i calculates the "compf" values for the (A−1)/2 bottom rows of pixels from the subimage that "belongs" to FP i−1 and stores those "compf" values and the "compf" values for the top (A−1)/2 rows of its subimage in shared bulk memory, FP i−1 will not need to recalculate the "compf" values for those pixels. While FP i is calculating the compf values for the bottom (A−1)/2 rows of data from the subimage of FP i−1, FP i+1 is calculating the "compf" values for the (A−1)/2 bottom rows of data from the subimage of FP i. When FP i classifies the bottom (A−1)/2 rows of its subimage, the needed "compf" values will have already been calculated by FP i+1. Thus, to classify the bottom (A−1)/2 rows of data from a given subimage, FPs will not need to calculate any "compf" values, as they are already stored in either the hold array or in the shared bulk memory. There is little possibility that one processor will require data before it is ready. For a processor to require such data, it would have to process (I/N)−((A−1)/2) rows of its data in the same time that another processor would have had to classify less than (A−1)/2 rows of its data.

2.3.7. Processing of Large Images

Assume that an FP system is configured as previously described. If the image to be processed will fit into bulk memory, the image can be processed according to the "striping scheme" discussed earlier. There is, however, another problem that can arise. An image may be too large to fit in the bulk memory.

Assume that there are L′ bulk memory banks per FP for data, separate from the bulk memory banks for the GPs, there are N FPs and that a three-by-three neighborhood is being classified. If an image will not fit into the N*L′/2 bulk memory banks, the host will transmit only the leftmost unprocessed columns of the image that will fit into N*L′/2 bulk memory banks at a time, L′/2 banks per FP. While the FP is processing one subimage in one half of its memory, the host can be loading the next subimage into the other half of the bulk memory. This will overlap the FP operation with the host's operation. If an image and its associated data can fit in N*L′ memory banks, it is still beneficial to use the striping scheme, as this will facilitate the preloading of the next image to be processed. Fig. 2.3.7.1 is an example of how an image is divided and processed. The FPs process subimages from left to right. Each subimage will be processed as described in Section 2.3.6. The stored class-conditional densities ("compf" values) for the rightmost two columns of data must be saved, as they are needed to process the next subimage. These columns of data will be stored in one of the L′ memory banks. This memory bank will not be accessed by the host, as it will contain the "compf" values necessary for the FP to process the next subimage. The exception to this rule is the last subimage. Since the FP will have no further processing, it is not necessary to save these values. Neither the first nor the last column an FP processes will be classified, as there is insufficient context information.

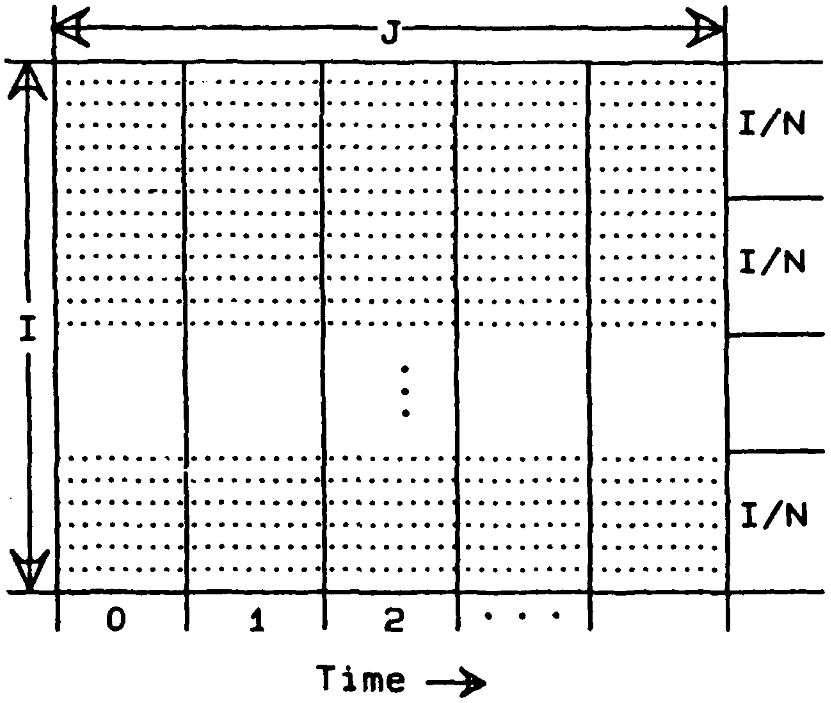


Fig. 2.3.7.1 Processing an image that is too large for bulk memories

Since the floating point operations require variable amounts of time, an FP processing its portion of the image may finish before the rest of the processors. With the FPs running asynchronously, it is theoretically possible for a given FP eventually to get two subimages ahead of its neighboring FPs. Subimage edge data would be destroyed for the neighboring FPs if the host were to load new data into the shared memory banks before the neighboring two FPs had finished with the old data. To prevent this from happening, after an FP processes two subimages, it must wait for the other FPs to finish.

When an FP finishes writing results into a bank of bulk memory, it signals the host to read all necessary data from that memory bank, even though an adjacent FP will need to read data corresponding to the subimage edge pixels from that bulk memory bank to process the next subimage. Since a read is non-destructive, the host reading from bulk memory will not hamper an FP reading from the same bulk memory bank. All FPs accessing a given bulk memory bank must set flags in bulk memory before the host can write to this bank. This will prevent the host from overwriting data that is still in use. As was stated in Section 2.3.2, with 20 non-zero G^ps, a single FP classifier took .035 secs. to classify a single pixel. Reading a pixel measurement vector from bulk memory will require 4.0 μ–secs.. Most of the execution time is spent in mathematical calculations, not fetching data, so any possible contention will have a negligible effect on pixel processing time.

2.3.8. Summary

In summary, the organization of the FP system given above will allow contention-free sharing of data. This means that N FPs will be able to operate approximately N times faster than one FP. Furthermore, the double-buffering of the bulk memories will allow the loading of images to be processed and storage of results to be overlapped with the classification operation of the FPs.

2.4. SIMD Implementations on PASM

2.4.1. Introduction

PASM is a dynamically reconfigurable multimicrocomputer system whose design will support as many as 1024 processors [SiS81]. SIMD implementations of contextual classifiers based on PASM are discussed in the next section. First, a brief overview of PASM is presented, limited to those aspects of PASM that are needed to understand the SIMD algorithms that follow.

2.4.2. Overview of PASM

Fig. 2.4.2.1 is a block diagram of PASM. The heart of the system is the **Parallel Computation Unit (PCU),** which contains N processors, N memory modules, and the interconnection network. The **PCU processors** are microprocessors that perform the actual computations. The **PCU memory modules** are used by the PCU processors for data storage in SIMD mode. When a PCU processor is combined with a PCU memory unit, it is referred to as a **Processing Element (PE).** The **interconnection network** provides a

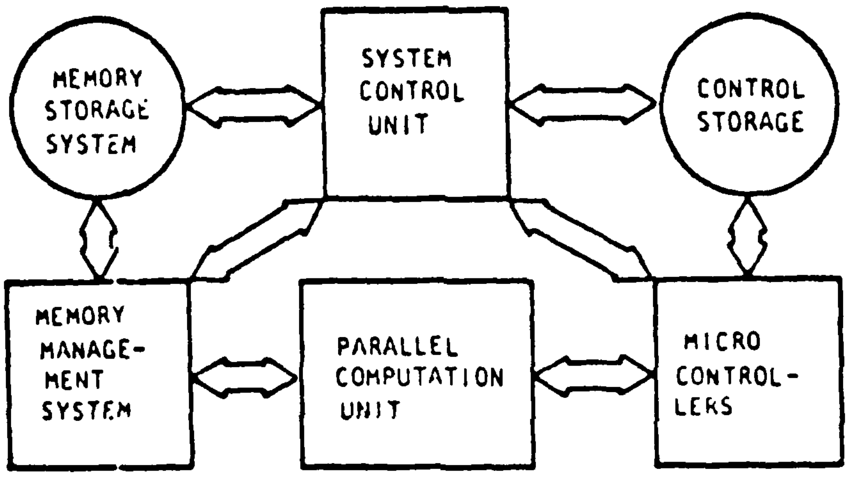


Fig. 2.4.2. Block diagram of PASM [SiS81]

means of communication among the PCU processors and memory modules. PASM uses data conditional and PE address masks to activate and deactivate PCU processors in SIMD mode.

The processors, memory modules, and interconnection network of the PCU are organized as shown in Fig. 2.4.2.2. A pair of memory units is used for each PCU memory module so that data can be moved between one memory unit and the secondary storage, while the PCU processor operates on data in the other memory unit. Each PCU memory unit may be as large as 64K 16-bit words. Two choices being considered for the network are the Generalized Cube [SiM81b] and Augmented Data Manipulator [SiM81a]. Their relative merits are currently under study [McS82].

The **Micro Controllers (MCs)** are a set of microprocessors which act as the **control unit** for the PCU processors in SIMD mode. **Control Storage** contains the programs for the MCs. Each MC memory module consists of a pair of memory units. This allows programs and/or common data to be moved between Control Storage and one MC memory unit, while the MC is using the other memory unit.

The **Memory Management System** controls the loading and unloading of the PCU memory modules. It employs a set of cooperating dedicated microprocessors. The **Memory Storage System** is the secondary storage for these files. Multiple devices are used to allow parallel data transfers. The **System Control Unit** is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM.

The approach taken to contextual classification using PASM in SIMD mode is different from that for the FP system, since the processors are synchronized and there is no directly-wired shared memory. There are three main differences between the FP and SIMD implementations. First, it is technologically feasible to construct a multimicroprocessor SIMD machine with many more than 16 processors. Second, there are differences in computational capabilities, i.e., 16 FPs may be faster than 32 microprocessors. Third, in SIMD mode, the program is stored in the control unit (MCs), which broadcasts it to the PCU microprocessors. The control unit also stores the G^p array, decoding and broadcasting each element as needed. In the FP system, each FP stores a copy of the program and G^p array.

2.4.3. *Linear Contextual Classification on PASM*

Consider using PASM to implement the contextual classifier based on a horizontally linear neighborhood of size three. If the image to be classified is a typical LANDSAT [NAS72] frame (I=3250,J=2340), 776 PEs will be assigned 7427 pixels and 248 PEs will be assigned 7426 pixels. Classification is accomplished by having each of the PE's execute the serial algorithm of Section 2.2.2 simultaneously. For example, all PEs first calculate the "compf" values for their pixels. This is done simultaneously in all PEs, where the 248 PEs assigned 7426 pixels will be disabled for the last PE operations. All PEs will then send their neighbor the "compf" values that need to be shared. By extending the previously discussed striping scheme to include a non-integer number of rows assigned to each PE, this task division is realizable. The modified striping scheme, shown in Fig. 2.4.3.1, requires 2C additional network transfers over the original striping scheme for sharing "compf" values between adjacent PEs. This cost is negligible when compared to the classification time of 7426 pixels. Each of the interconnection networks under consideration for PASM can perform each of the 2C required data transfers in one pass through the network, where each transfer involves N PEs i.e., when PE i is transferring data to PE i−1, PE i−1 is transferring data to PE i−2, etc. On PASM, a PE will get an instruction to send another PE the shared data. This differs from the FP system, where an FP gets the data it needs on its own. The asynchronous nature of the FP system makes this modification to the striping algorithm less efficient on the CDC system.

An image may be so large that not all of the data will fit into the PCU memory space allocated. The double-buffered memory modules can be used so that as soon as the data in one memory unit are processed, the processor can switch to the other unit and continue executing the same program. When the processor is ready to switch memory units, it signals the Memory Management System that it has finished using the data in the memory unit to which it is currently connected. The processor switches memory units, assuming that the data is present, and then checks a data identification tag to ensure that the new data are available. The Memory Management System can then unload the "processed" memory unit and load it with the next subimage. For both the one-by-three linear window and the three-by-three nonlinear window, this scheme will require some mechanism to allow the "compf" values for the last two columns of a subimage in a given memory bank to be available when the associated processor switches to the next memory unit.

One method of doing this maintains a copy of local data in both memory units associated with a given processor, so that switching memory units does

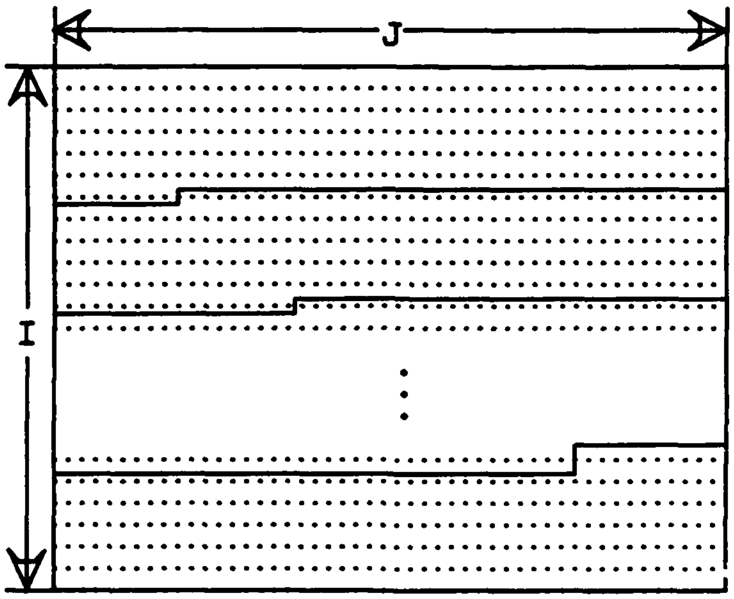


Fig. 2.4.3.1 Modified striping scheme

not alter the local variable storage associated with the processor [SiS81]. In essence, this technique makes use of the conventional store through techniques, as described in [Hay78]. This scheme would be used only when multiple subimages are to be processed.

The time required to classify a LANDSAT frame is the same as the time required for each PE to classify 7427 pixels. If each PE were to classify 7427 pixels, 7,605,248 pixels would be classified, representing a speedup of 1024. For a 3250-by-2340 image, PASM will classify 7,605,000 pixels in the same time. This is 99.997% of the theoretical improvement of 1024.

2.4.4. Non-Linear Contextual Classification on PASM

Consider implementing a three-by-three non-linear contextual classifier on PASM. The I-by-J image is divided into N subimages. Each PE will be assigned an $(I/\sqrt{N})$-by-$(J/\sqrt{N})$ array as shown in Fig. 2.4.4.1. If I is non-divisible by $\sqrt{N}$, some PEs will have to process $(I/\sqrt{N})+1$ rows of data, while others will have to process $I/\sqrt{N}$. Similarly, if J is non-divisible by $\sqrt{N}$, some PEs will have to process $(J/\sqrt{N})+1$ columns of data instead of $J/\sqrt{N}$. In all cases, the PEs processing the smaller amount of data will be disabled while the remaining PEs continue processing. All of the PEs will execute the algorithm discussed in Section 2.2. Each PE can classify all the pixels in its subimage which are not on the subimage edges. All PEs can do this simultaneously. To classify subimage edge pixels, the PEs must share data by passing information through the interconnection network. For example, in order for PE 0 to classify pixel $(0,(J/\sqrt{N})-1)$ it needs to get the "compf" values for pixel $(0,J/\sqrt{N})$ from PE 1. Both networks under consideration can perform each of the nearest neighbor

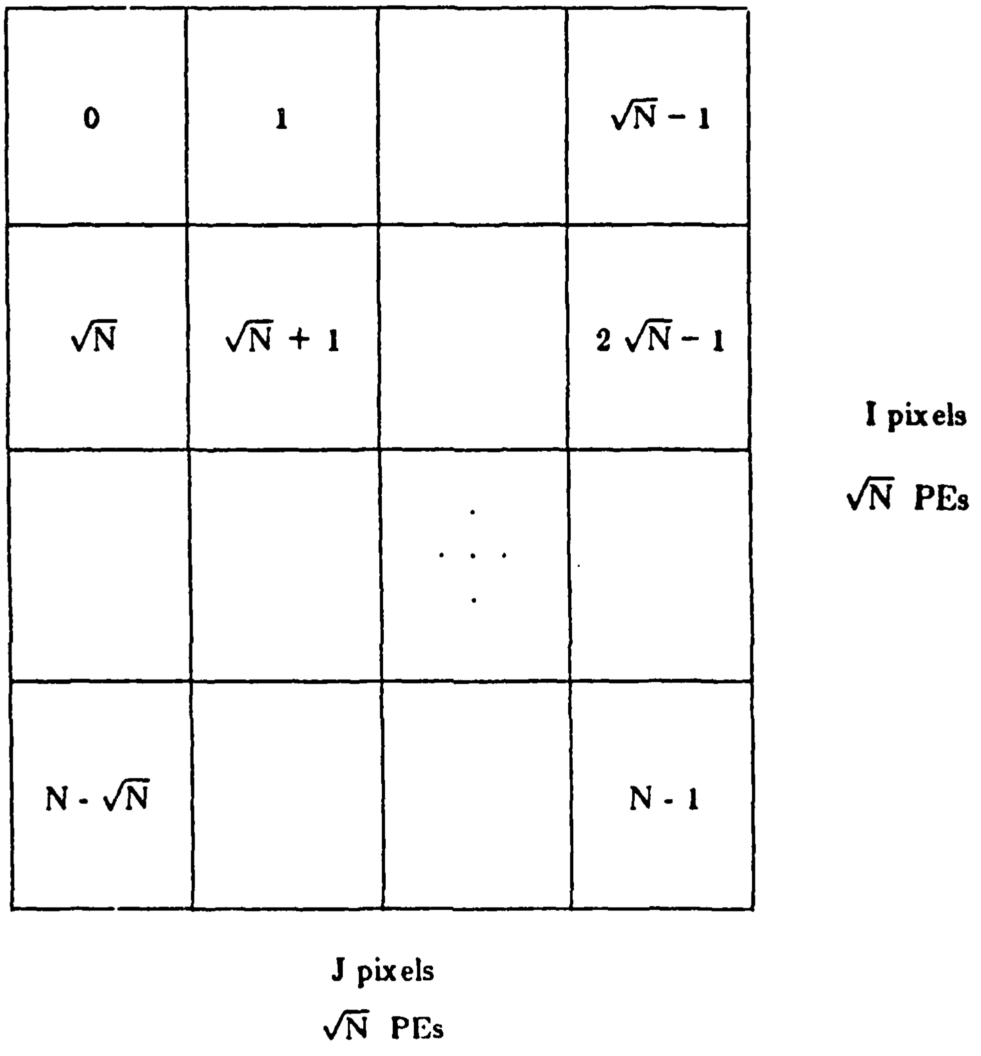


Fig. 2.4.4.1. Dividing an image using a "checkerboard" pattern. Each square represents one PE with a $(I/\sqrt{N})$–by–$(J/\sqrt{N})$ subimage. The PE number is in the square.

inter-PE transfer operations in one pass through the network.

One way to share "compf" values among PEs is to have each PE first compute and store the "compf" values for its edge pixels in a vector called EDGE. (Later, when a PE needs the "compf" values for these pixels in order to classify pixels in its own subimage, they are read from EDGE, not recomputed.) Each PE sends copies of these values to the appropriate "adjacent" PE. A PE saves the value it receives in a vector OUTEREDGE. Each PE accesses its own OUTEREDGE vector when it is ready to classify its edge pixels. This method requires only $((2(I+J)/\sqrt{N})+4)C$ parallel data transfers. For each of the required transfers, the networks being considered for PASM will allow all PEs to perform the transfer simultaneously. A checkerboard division of the image was used since, in general, it requires fewer inter-PE transfers than dividing the image by rows or columns. For arithmetic operations and "compf" calculations, a perfect factor of N speedup is attained. This is done at the "cost" of $((2(I+J)/\sqrt{N})+4)C$ inter-PE transfers. These data transfers are negligible when compared with the $I*J*C/N$ "compf" computations.

2.5. Conclusions

Based on simulated results, timings for contextual classification on an FP system have been presented and discussed. A potential system configuration for the FP system has been presented, and its use discussed. For comparison, timings have been presented for contextual classification on a PDP-11/70. It was found that a PDP 11/70 runs at a slightly slower speed than a single FP on the contextual classification algorithms examined. Further, it was shown that N FPs could execute contextual classification almost N times as fast as one FP. Thus, the multiprocessor parallelism of an FP system can be successfully exploited.

It was shown that N processors in the SIMD mode of operation could accomplish contextual classification almost N times faster than one processor of the same type. In particular, an SIMD algorithm for PASM to perform the computationally intensive task of contextual classification was presented.

The FP and PASM approaches could be combined [SmS82]. A multimicroprocessor SIMD machine with shared memories (as in the FP approach) and no interconnection network would be an efficient special-purpose system for performing contextual classification with various size and shape neighborhoods.

Thus, through the use of parallel computer systems, such as PASM and CDC FPs, the types of computations required for contextual classifiers and other computationally demanding remote sensing processes can be implemented efficiently. This will not only reduce the computation time required to do contextual classification, but will also allow the investigation of techniques which may otherwise be considered infeasible.

CHAPTER 3
PARALLEL PROCESSING CONCEPTS FOR REMOTE SENSING APPLICATIONS

3.1. Introduction

Multispectral image data collected by remote sensing devices aboard aircraft and spacecraft are relatively complex data entities. Because of the multispectral nature of remote sensing image data, vectors are used to represent the data. The execution of even the simplest classification algorithms may require large amounts of computation time. Thus, in order to allow complex classification algorithms to become more feasible, special hardware (such as the previously discussed parallel architectures) to increase the execution speed is of interest.

For many remote sensing tasks, all pixels in a given image are treated in a similar fashion. This implies that the same numerical operations are done on all pixels. Thus, the same instructions are performed on multiple data sets. It would appear that SIMD machines, such as those discussed in Chapter 1, are particularly well-suited to these tasks. Further, since images as large as 3250-by-2340 pixels [NAS72] are common, a system that has as many as 1024 processors would be well-suited for image processing tasks. Large scale integration makes just such parallel systems possible.

The applications of such a machine to image processing tasks is the topic under consideration here. Section 3.2 introduces a potential machine architecture. Sections 3.3, 3.4, 3.5, and 3.6 discuss how such a system can be applied to smoothing, maximum likelihood classification, contextual classification, and image correlation, respectively. The fault tolerance of MuRSS is discussed in Section 3.7. Enhancements to the MuRSS architecture to increase fault tolerance are presented in Section 3.8, where the fault tolerance of both the original and enhanced systems are compared. An overview of MPP, the **M**assively **P**arallel **P**rocessor (an already existing architecture) is presented in Section 3.9, along with a discussion comparing MPP to the enhanced MuRSS system in the areas of performance, capabilities, and fault tolerance.

3.2. Machine Architecture

The proposed SIMD architecture, **Mu**ltimicroprocessor **R**emote **S**ensing **S**ystem **(MuRSS),** is shown in Fig. 3.2.1. The system consists of N+1 processing units **(PUs)** numbered from 0 to N and 2N+2 memory modules numbered from 0 to 2N+1 (Fig. 3.2.2). During normal operation, N PUs (numbered 0 to N-1) and 2N memory modules (numbered 0 to 2N-1) will be used (Fig. 3.2.3). PU number N, memory module number 2N, and the wrap-around connection are for fault tolerance.

Each PU will be a commonly available microprocessor, such as a 68000 [Mot80] equipped with a floating point unit and will be connected to four busses in addition to its own private bus. The private bus will be connected to the PU's private memory which will contain such things as local variables and

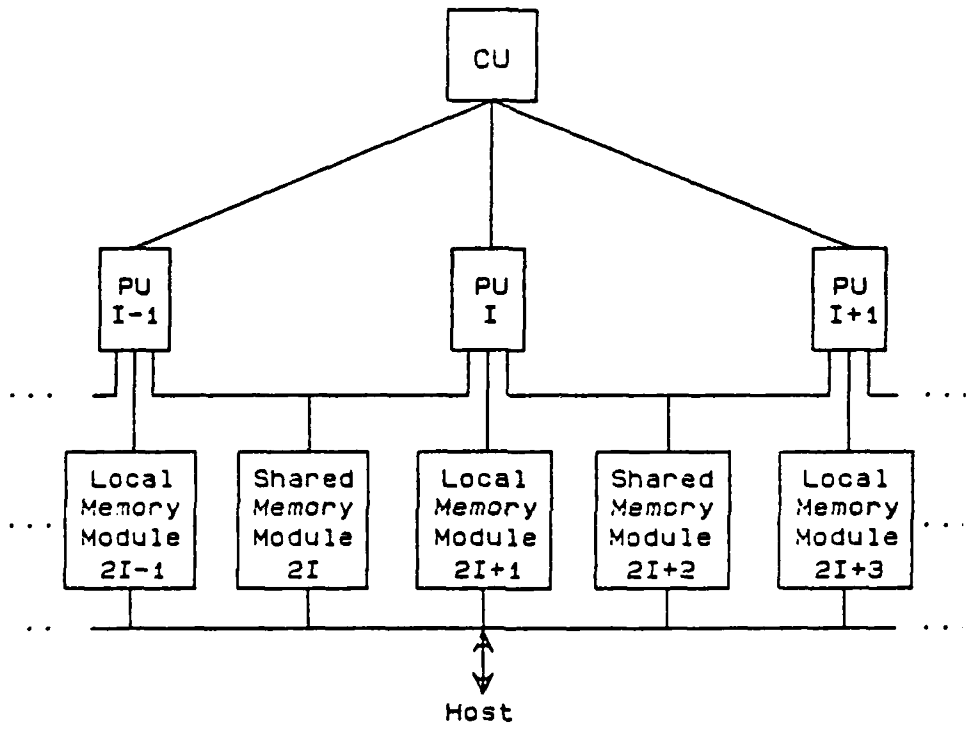
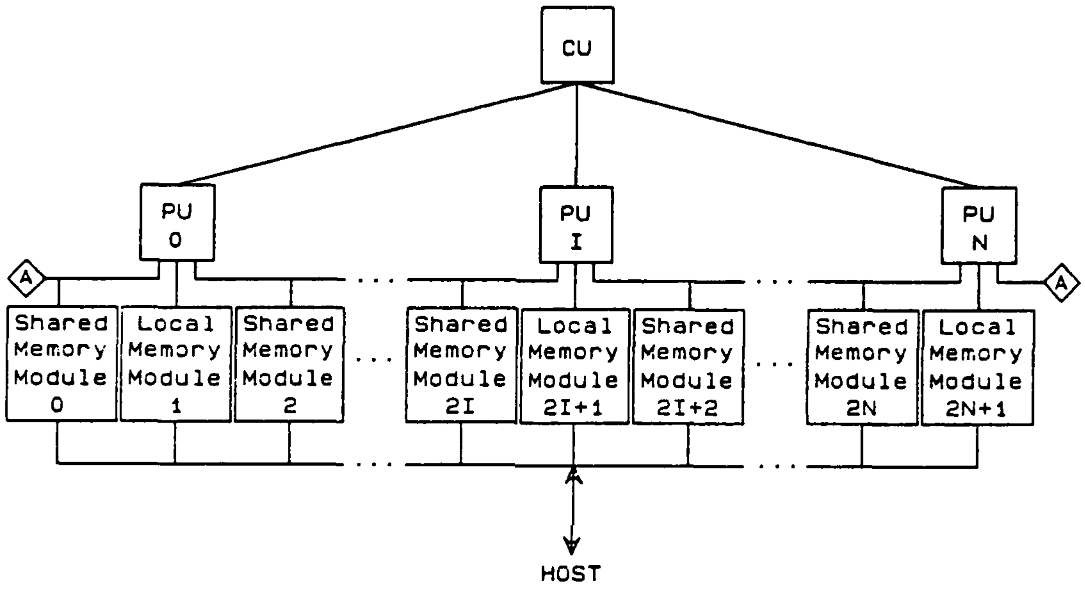


Fig. 3.2.1 MuRSS system architecture



Note: ⟨A⟩ denotes wrap-around connection

Fig. 3.2.2 N+1 PU MuRSS system overview

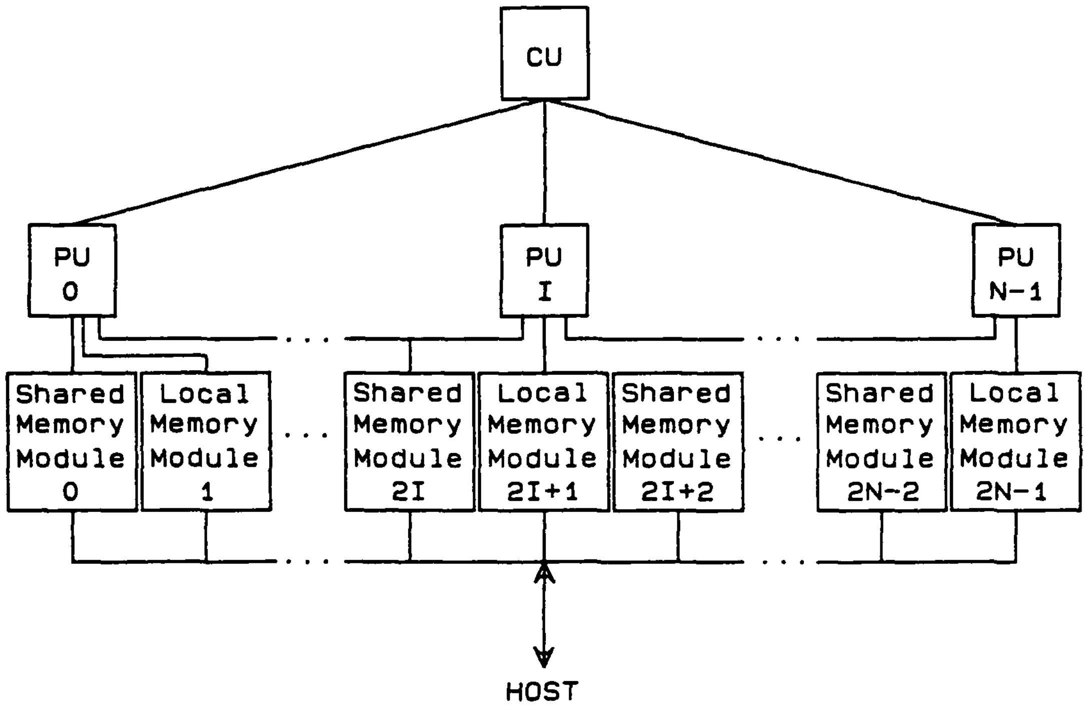


Fig. 3.2.3 MuRSS system architecture during normal operation

monitor routines. One of the remaining four busses will be used to communicate with the control unit, while the other three busses, numbered 0,1, and 2 (Fig. 3.2.4), will be connected to banks of memory. Two of these busses will be connected to "shared" memory banks. Thus, these busses, and consequently the associated memory banks, will also be connected to adjacent processors. This will allow data to be shared among adjacent PUs for window based operations, like the contextual classifier discussed in Section 2.2. (Note that the 2 bus of PU N will share its memory with the 0 bus of PU 0 for reasons discussed later). The third bus will be connected to a "local" memory bank. Each of the three busses of a PU can address up to 2^8 64K-byte banks of memory.

It would appear that direct PU-to-PU intercommunication could occur through the shared busses. This is not possible because MuRSS is an SIMD architecture with no special latching hardware on the shared busses. Since all the PUs must either read or write simultaneously, data cannot be shipped from PU-to-PU without some form of latch (like the shared memory). Thus, PU-to-PU intercommunication must be done through the shared memory. (Such latches could be added to the design, but for the applications investigated thus far, the use of the shared memory for communication appears to be sufficient.) Therefore, the memory banks that are "shared" can be used to store common data for a PU and its linearly adjacent neighbor, eliminating the need for a more complex interconnection structure when performing window-based processing operations.

Memory contention is not a problem, as the only way contention can occur is if two processors try to access the same shared memory banks. This cannot happen with this SIMD system, since whenever processor I is using its 0 bus,

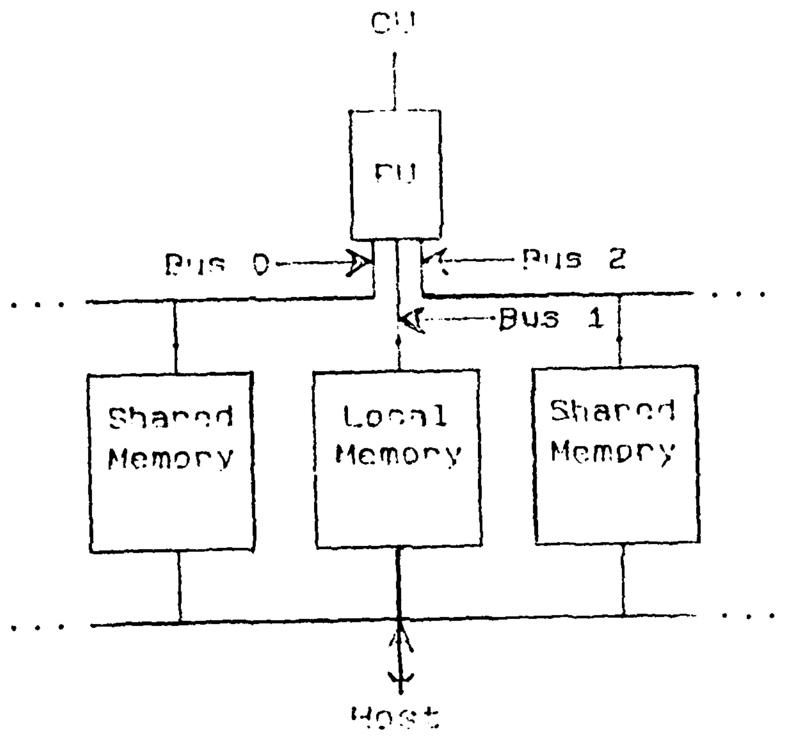


Fig. 3.2.4 Bus structure of MuRSS PU

processor I−1 must also be using its 0 bus (it cannot, for example, be using its 2 bus) (Fig. 3.2.4). For the purposes of this discussion, the memories (either directly or indirectly) associated with busses 0 and 1 of PU I will be said to be **associated with** PU I. In general, memory modules 2I and 2I+1 will be **associated with** PU I, shared memory module 2I with bus 0 and local memory module 2I+1 with bus 1.

It is possible that the shared memories may be needed to store local data, e.g., when there is too much local data for the local memories to handle. In this case, only the memory addressable by the busses associated with each processor (i.e., bus 0 and bus 1) should be used to store local data. Thus, for PU I, memory module 2I should store data to be shared with PU I-1 and any local data that will not fit into memory module 2I+1. Memory module 2I+1 should be used to store the majority of local data for PU I. Memory module 2I+2 should not be used for data local to PU I.

This requirement is not a rigid requirement, i.e., when all 2N+1 memory banks are working, PU I could use memory modules 2I, 2I+1, and 2I+2 for local data; however, if even one memory bank fails, algorithms not satisfying this requirement cannot be executed by MuRSS.

The organization of the memory is shown in Fig. 3.2.5. This figure assumes that there are **L** memory banks associated with each bus. The memory associated with MuRSS will be dual ported, allowing a given memory bank to be connected to two busses simultaneously. One bus will be connected to a MuRSS PU, while the other bus will be connected to the host. This will allow the host to address the memories separately from the processors, enabling the host to load/unload data into/from half the banks, while the processor operates on data from the other half, maximizing overlap. This type of overlap

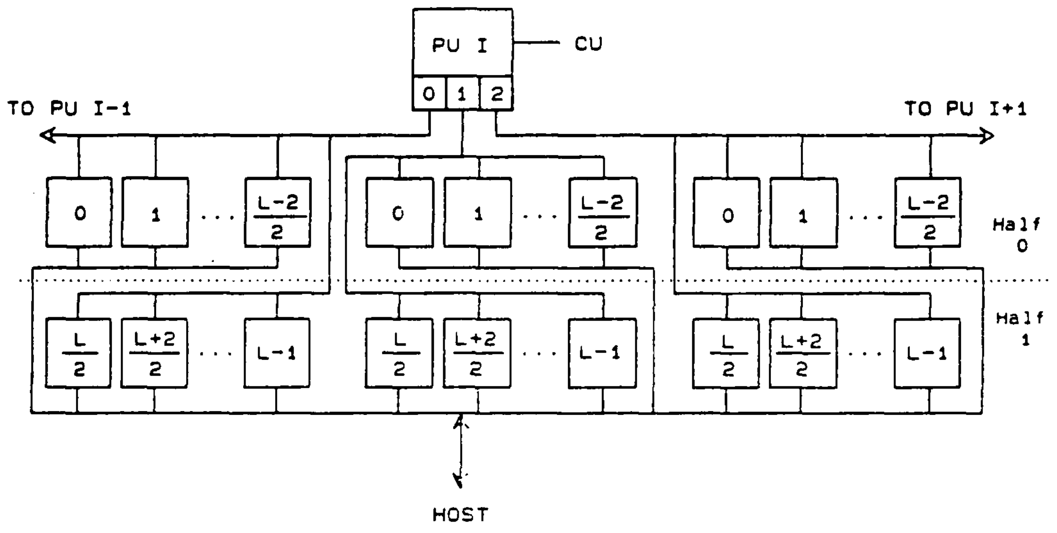
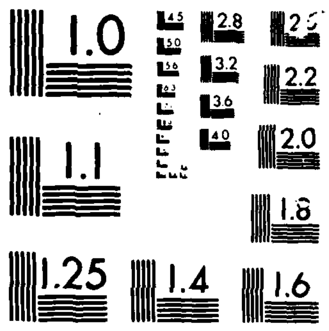


Fig. 3.2.5 Organization of MuRSS memory

is called **double buffering** and is similar to the approaches taken with the CDC FP system in Section 2.3.6 and with the PASM system in Section 2.4.3 [SiS81]. Double buffering can be implemented in hardware, allowing the memory to be addressed contiguously, simplifying the loading and unloading of data. If the addresses associated with the memory banks (as viewed by the host) are:

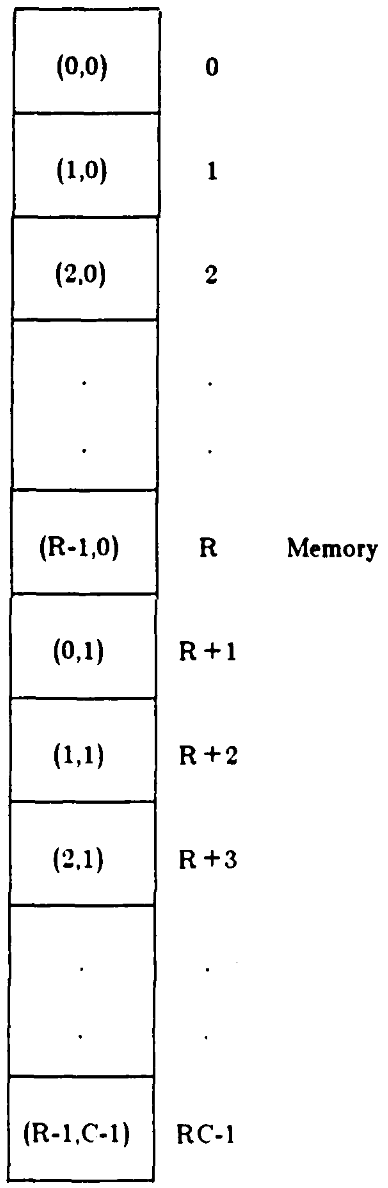
Use:	Half	PU number	Bus	Bank	Address
Bit Positions:	35	34 - 25	24	23 - 16	15 - 0

where the **Half** indicates which half of the double buffer is to be addressed, the PU number is the number of the associated PU, and the **Bus** bit is the bus to be addressed (0=left, 1=center). When a fault occurs, the CU can re-program the PU numbers, so the remaining memory can be treated as contiguous by the host (this is discussed further below). If all memory banks are attached to a bus that is accessible by the host, the host can view the memories as contiguous. each PU is associated with 2^9 64K-byte memory banks, many processors will not be able to directly this much memory ($> 2^{32}$ memory locations), so the host may need to use some form of memory controller. Some memory controllers may allow a special micro-program to be installed to facilitate handling the memory organization.



MICROCOP' CHART

Consider the procedures that the host must perform to address the pixel (i,j) in an R row by C column image consisting of b-byte elements. Assume the data is stored in column major format, i.e.,



If each PU has the same number of columns of data, then pixel (i,j) is in PU P:

$$P = \left\lfloor j \times \frac{N}{C} \right\rfloor \qquad 0 \leq j \leq C-1$$

where there are N PUs in use. The host can calculate the bus B to be:

$$B = \begin{cases} 0 & \left(j - \frac{P \times C}{N} \right) < C' \\ 1 & \text{else} \end{cases}$$

where there are C' columns of data stored in each shared memory unit. Let B' be the base address of the array within the given memory unit. The address within the bus would be:

$$\text{address} = B' + \left\{ \left[\left(j - \frac{P \times C}{N} \right) - (C' \times B) \right] \times R + i \right\} \times b$$

This looks very complex, but these calculations must be done only once per column. Further, if many columns of data are to be loaded/unload into/from the memory units, the following algorithm can be applied:

```
int P;     /* PU counter */
int B';    /* Base address of array in shared memory */
int B";    /* Base address of array in local  memory */
int C';    /* Columns of data stored in shared memory */
int C";    /* Columns of data stored in local  memory */
int N;     /* PUs in use */

for ( P = 0 ; P < N ; P = P + 1 ) { /* for each processor */

        /* completely unload bus 0 of Processor P */
        read (b*R bytes from address B' of bus 0 of PU P);

        /* completely unload bus 1 of Processor P */
        read (b*R bytes from address B" of bus 1 of PU P);

}
```

This type of scheme is particularly convenient if a memory controller is used and the memory controller can perform **Direct Memory Access (DMA)** to and from the host's memory. If DMA is used, the above algorithm for unloading data from an N=1024 MuRSS would require:

2048 block reads
1024 compares and
1024 additions.

Further, if the entire "half 0" or "half 1" of the memory banks are to be read/written, only one read/write (of size 2^{35} bytes) would be needed. These transfers could occur between MuRSS and the host's secondary memory or the host's primary memory if it is large enough.

Loading and unloading of data by rows is very complex because the image data is stored in columns. The following algorithm demonstrates how the host must unload row data from MuRSS when an image is stored in column major format:

```
int P;              /* PU counter */
int B';      /* Base address of array in shared memory */
int B";      /* Base address of array in local  memory */
int C';      /* Columns stored in shared memory */
int C";      /* Columns stored in local  memory */
int bR;      /* Bytes of data per column */
int N;              /* PUs in use */
int i;              /* Row counter */
int j;              /* Column counter */

for ( i=0 ; i < R ; i=i+1) { /* each row */
        for ( P=0 ; P < N ; P=P+1 ) { /* each processor */
                for ( j=B' ; j < bRC' ; j=j+bR ) { /* shared columns */
                        /* unload one data item from bus 0 of PU P */
                        read (b bytes from address j of bus 0 of PU P);
                }
                for ( j=B" ; j < bRC ; j=j+R" ) { /* local columns */
                        /* unload one data item from bus 1 of PU P */
                        read (b bytes from address j of bus 1 of PU P);
                }
        }
}
```

This algorithm represents a significant number of calculations on the part of the host. With the large number of individual reads, each which takes time to create a system buffer, it is less cumbersome for the host to unload the image in column format and transpose the image in its own memory.

If the image is loaded in row major format, the algorithms are similar, but rows and columns are reversed. Similarly, for such a scheme, it is simple for the host to deal with row data and complex for the host to deal with column data. Given that an image is treated consistently (i.e., not transposed during loading or unloading), MuRSS can handle data in either row major or column major format without excessive processing. For example, consider the image in Fig. 2.3.2.1. Here, each PU would hold an entire stripe I/N-by-J pixels large, effectively processing the image in row major format. The shared data in Fig. 2.3.6.2, as required for classification of non-linear windows, would be stored in the shared memories.

Consider an image stored in column major format. Define the relative index of the pixel (i,j) to be the row and column of the pixel relative to the uppermost left pixel in the PU's address space. In an image stored in column major format, the absolute pixel (i,j) would have relative address (i,j'), where j' is the number of columns to the right of the leftmost column addressable by the PU. Thus, if each PU could address ten columns of data, the relative address (1,0) would correspond to the N pixels whose absolute addresses were $\left\{(1,10 \times k) \;\middle|\; k=0,1,2,\ldots,N-1\right\}$. Typically, if C' columns of data were stored in the shared memory associated with bus 0 of PU I, then C'/2 pixels would be processed by PU I−1 and C'/2 pixels would be processed by PU I, as was done for the FP system discussed in Section 2.3.6. This means that PUs will typically start their processing for the pixels with relative address (0,C'/2). For pixels with relative address (i,j'), if there are C' columns of data associated with busses 0 and 2 and C" columns of data associated with 1, the bus can be determined as follows:

$$\text{bus} = \begin{cases} 0 & j' < C' \\ 1 & C' \le j' < C' + C'' \\ 2 & \text{else} \end{cases}$$

The address of the pixel (within the bus) is:

$$\text{address} = \begin{cases} b \times (j' \times R + i) & \text{bus } 0 \\ b \times ((j' - C') \times R + i) & \text{bus } 1 \\ b \times ((j' - C' - C'') \times R + i) & \text{bus } 2 \end{cases}$$

Addressing within a given column requires setting a pointer to the base address of the column and incrementing or decrementing it by a fixed amount. If $(N > 2^8)$ and

The CU will be a special purpose processor. It will be equipped with memory, in which it will store its program, global data, the program to be broadcast to the PUs, and its local variables. The amount of memory is variable and is a function of cost and the processor chosen for the CU.

The host will be assumed to be a computer such as an IBM-370 or a PDP-11 series machine. All support operations, such as formatting input and formatting output, will be performed by the host.

Each PU is based on the Motorola 68000 microprocessor. From [Mot81], a 12.5 MHz 68000 can perform a 16-bit integer addition in 400 nsec. The 1024 68000's in MuRSS can perform 2560 million integer additions per second. In addition, MuRSS equipped with Motorola's high speed floating point software can perform 73 million 32-bit floating point additions per second or 36 million 32-bit floating point multiplications per second. When the PUs are equipped with the planned 16.666 MHz MC68881 floating point processor, MuRSS is capable of 367 million 32-bit floating point additions, 330 million 32-bit floating point multiplications, or 270 million 32-bit floating point divisions per second. All floating point operations are in accordance with the IEEE floating-point specification P754.

3.3. Smoothing on a Parallel SIMD Machine

Smoothing is a method of noise reduction for image data. The measurement vector for each pixel is replaced by the average of the measurement vector for that pixel and the measurement vectors of the eight surrounding pixels. Consider the following example, as shown in Fig. 2.2.1.1(b). $x_{i,j}$, the measurement vector for pixel (i,j) is replaced by:

$$x'_{i,j} = \frac{(x_{i-1,j-1}+x_{i,j-1}+x_{i+1,j-1}+x_{i-1,j}+x_{i,j}+x_{i+1,j}+x_{i-1,j+1}+x_{i,j+1}+x_{i+1,j+1})}{9}$$

Thus, for each pixel, eight vector additions and one division of a vector by a constant is required. Consider the case where each measurement vector is 4-dimensional and the image is I-by-J pixels. Smoothing the image on a serial machine will require 8*I*J vector additions and I*J divisions, translating to 32*I*J additions and 4*I*J divisions.

If I is sufficiently large ($> 2N+1$) and a multiple of N, the image can be divided into N rows I/N pixels high as shown in Fig. 2.3.2.1. This scheme is called **striping** and has been discussed in Section 2.3.2. Each processor will process one stripe. In order to process all pixels in a given stripe, a processor will need to access one row of pixels from each bordering stripe. This means that at least two rows of data will have to be stored in shared memory. For example, with a 512-by-512 image and 32 processors, processor 0 will process rows 0 to 15, while processor 1 will process rows 16 to 31, etc. Memory 0 will store rows 0, memory 1 will store rows 1 through 14, memory 2 will store rows 15 and 16, etc. Note that memories 0 and 2 could contain more rows of data. In general, up to two rows of data must be stored in each shared memory. The rest of the image can be stored in the local memory banks. The total processing time associated with an image is: 32*I*J/N additions and 4*I*J/N divisions. Thus, the theoretical maximum speedup by a factor of N is achieved.

If I is not a multiple of N, all processors will process $\lfloor I/N \rfloor$ rows, then I mod N processors will have to process one extra row of data. For simplicity, assume that rows cannot be subdivided. Thus, some processors will have to process a stripe $\lfloor I/N \rfloor$ rows wide, while other processors will have to process a stripe $\lceil I/N \rceil$ rows wide. If each row is J pixels wide, the total processing time associated with a given image will be:

$$32*J*(\lceil I/N \rceil) \quad \text{additions}$$

$$4*J*(\lceil I/N \rceil) \quad \text{divisions}$$

This represents an increase of at most 32*J additions and 4*J divisions over the ideal case. The efficiency of the above implementation can be represented by the ratio of the time required for an ideal speedup to the actual processing time [SiS82b]. This translates to:

$$\frac{I/N}{\lceil I/N \rceil}$$

The worst case efficiency is achieved when one processor is running while the remaining processors are idled. Mathematically, this is when the difference between I/N and $\lceil I/N \rceil$ is a maximum. For example, with N=1024 and an image with 4097 rows, this represents an efficiency of 80%, while for I=65537, this represents an efficiency of 98.4%. The larger the image, the closer the efficiency is to 100%.

Note that the efficiency is a function of the number of rows. Processing columns instead of rows will make the efficiency a function of the number of columns and may allow N processors to operate more efficiently. An alternative to the above method is to use the "modified striping" scheme discussed in Section 2.4.3.

The time required to smooth an image using modified striping is:

$$32 * \lceil I * J/N \rceil \text{ additions}$$

$$4 * \lceil I * J/N \rceil \text{ divisions}$$

For the ideal speedup of N, the ceiling function would be absent, thus the ratio of the ideal speedup to the actual speedup becomes:

$$\frac{I * J/N}{\lceil I * J/N \rceil}$$

For N=1024, and an image of size 1025-by-4097, the efficiency is 99.99+%.

This method, thus, leads to a higher overall utilization of the processors. Further, for images greater than 2N-by-2N, the utilization is independent of the orientation of the image, i.e, whether the image is striped based on rows or columns.

If edge data is to be handled differently than data internal to the image, when one or more processors reach an edge, all other processors must be disabled. The remaining processors then process their edge data. This is not required in the simple striping scheme, as all the processors reach an edge at the same time. In a modified striping scheme (with horizontal stripes), the probability that a given processor is processing an edge pixel is:

$$p_{edge} = \frac{2*\lceil I/N \rceil}{\lceil I*J/N \rceil}$$

In addition, each PE must decide (for each pixel it processes), whether that pixel is an edge or non-edge pixel. The modified striping scheme requires $2*(\lceil I*J/N \rceil)$ more comparisons and a maximum of $p_{edge}*\lceil I*J/N \rceil$ more edge pixel computations than the simple striping scheme in the ideal case where I or J divides N. Simple striping requires at most 2 more edge pixel computations and I−2 more internal pixel computations than simple striping in the ideal case. The striping scheme to be used should minimize the number of computations above the ideal case.

Images smaller than 2N rows have not been considered, as they do not have enough rows to utilize the full machine. Each processor will have to store at least one row of data in each of its shared memory banks. This implies that there are at least two rows of data per processor. Multiplication of the two row minimum by the N processors yields 2N rows. If striping is done by columns, then the argument is similar. To process small images (using rows),

$\lceil I/2 \rceil$ processors would have to be enabled, while the rest of the processors were disabled for the entire task.

3.4. Maximum Likelihood Classification

Maximum likelihood classification (MLC) [SwD78] classifies each pixel independently of all others. Assume that the input data can be described by a Gaussian distribution function [SwD78]. Thus, the probability that pixel (i,j) is in a given class $\omega_k \in \Omega = \{\omega_1, \omega_2, \cdots \omega_n\}$ is:

$$p(X_{ij} | \omega_k) = \frac{1}{\sqrt{2\Pi^n | \Sigma_k |}} e^{-\frac{1}{2}(X_{ij}-M_k)^T \Sigma_k^{-1} (X_{ij}-M_k)}$$

where X_{ij} is the measurement vector for pixel (i,j), M_k is the mean vector for class k, Σ_k is the covariance matrix for class k. A pixel is assigned to a given class such that $p(X_{ij} | \omega_k)$ is maximized. It is possible to use a discriminant function [SwD78]:

$$d(X_{ij} | \omega_k) = -\left[\ln \left| \Sigma_k \right| + (X_{ij}-m_k)^T \Sigma_k^{-1} (X_{ij}-m_k) \right]$$

Maximizing this last discriminant function for X_{ij} over Ω will yield the same result as maximizing $p(X_{ij} | \omega_k)$ over the same Ω. The discriminant function is considerably less complex to calculate than the probability, so discussion is based on the discriminant function.

The calculation of $-\ln\left|\Sigma_k\right|$ and Σ_k^{-1} is done once for each information class and is negligible when compared to the calculation of the discriminant function for each class for each pixel in a given image. Again assuming X_{ij} is 4-dimensional, $X_{ij}-m_k$ can be done in four additions per class per pixel. By utilizing the symmetry of Σ_k^{-1}, $(X_{ij}-m_k)\Sigma_k^{-1}(X_{ij}-m_k)$ can be performed in 20 multiplies and 9 additions for the four spectral band case. Thus, the calculation of the discriminant function will require 20 multiplies, 15 additions, and one sign change per pixel per class. Finally, for C class data, C-1 compares per pixel will be needed in addition to the calculation of the discriminant function. On an I-by-J image, classification of all I*J pixels will require 20*I*J*C multiplications, 15*I*J*C additions, and I*J*(C-1) compares for a standard serial processor.

Consider implementing the MLC on MuRSS. The CU will broadcast class dependent constants, such as Σ_k^{-1} and m_k as part of the SIMD program. Each pixel is classified independently, thus there is no need for any inter-processor communication. Using the modified striping scheme to divide the I-by-J image, N PUs will be able to perform an MLC

$$\frac{I*J/N}{\lceil I*J/N \rceil} \times N$$

times faster than a single PU. Further, since this operation requires no inter-processor data transfers, images as small as N pixels can be processed without disabling PUs for the entire operation.

3.5. Contextual Classification

The "class" associated with a given pixel is not independent of the classes of adjacent pixels. Stated in terms of a statistical classification framework, there may be a better chance of correctly classifying a given pixel if, in addition to the spectral measurements associated with the pixel itself, the measurements and/or classifications of its "neighbors" are considered as well. The image can be considered to be a two-dimensional random process incorporated into the classification strategy. This is the objective of "contextual classifiers" ([WeS71] and [SwV81]), in which a form of compound decision theory is employed through the use of a statistical characterization of context. Recent investigations have demonstrated the effectiveness of a contextual classifier that combines spatial and spectral information by exploiting the tendency of certain ground-cover classes to occur more frequently in some spatial contexts than in others [SwS80], [WeS71], [SwV81], and [TiS81]. For a more complete description of contextual classifiers, please refer to Section 2.2.1.

The application of MuRSS to contextual classification is a straightforward extension of the method applied in Sections 2.3.2 and 2.3.3. For the three-by-three window, data allocation and timing analysis is analogous to that for smoothing. The main difference is that for smoothing, only the raw pixel data is shared. For the contextual classifier, the "compf" values of the subimage edge pixels are shared instead. The parallel processor version of the one-by-three horizontally linear window is similar. Other sizes and shapes of windows can be handled analogously.

3.6. Image Correlation on a Parallel Machine

Image correlation, as described in [SiS82a], is used to measure the degree of similarity between a match image and an equal sized area of an input image. Typical images can be at least 4096-by-4096 pixels, with match areas on the order of 64-by-64 pixels. For the purposes of this paper, images on the order of 65536-by-65536 pixels will be considered.

Let the symbols x and y denote single elements of arrays X and Y, where X is the match image and Y is the area of the input image under consideration (same dimensions as X). Let M be the total number of elements in the match area X. Define:

$$S_{XX} = (1/M)(\sum x^2-(\sum x)^2)$$

$$S_{XY} = (1/M)(\sum xy-\sum x\sum y)$$

$$S_{YY} = (1/M)(\sum y^2-(\sum y)^2)$$

$$R_{XY} = S_{XY}/\sqrt{S_{XX}S_{YY}}$$

S_{XY} is the covariance of the match area with a portion of the input area. Large positive values for S_{XY} indicate similarity between the match image and the input image, while large negative values for S_{XY} indicate similarity between the negative of the match image and the input image. Values near zero indicate little similarity between the two images. R_{XY} is the linear correlation coefficient of the statistics. Simplistically R_{XY} is a normalized version of S_{XY} in which $R_{XY} = 1$ indicates an identical match, $R_{XY} = -1$ indicates an identical match with the negative of the input area, and $R_{XY} = 0$ indicates no correlation between the match area and the input image. A correlation value will be computed for each position in which the match image can fit into the R row by C column input image.

The calculation of R_{XY} is dominated by the time to compute $\sum xy$, $\sum y$, and $\sum y^2$. $\sum x$ and $\sum x^2$ do not change from input window to input window, and can thus be pre-computed. For a match template with r rows and c columns, each $\sum xy$ and $\sum y^2$ requires r*c multiplications and rc−1 additions. $\sum y$ requires rc−1 additions. These operations have to be done for each position of the match template in the input image. Special methods of computing $\sum y^2$ and $\sum y$ can decrease the time requirements of this algorithm. Consider the following algorithm for computing the sum of the pixel values ($\sum y$'s) in each match template.

Assume that for input image Y the position of the match area is defined by the coordinates of the upper left hand corner of the match area. Define a vector "colsum" [SiS82a] of length C as:

$$\text{colsum}(j) = \sum_{i=k}^{k+r-1} Y(i,j)$$

where k is the row coordinate of the current portion of the match area and $0 \leq j < C$. Let "SUM" be an R−r+1-by-C−c+1 array, where SUM_{ij} is the sum of the pixels of the input image for the match area position (i,j), $0 \leq i \leq R-r+1$, $0 \leq j \leq C-c+1$.

Initially, colsum is calculated for all C columns of row 0. SUM(0,0) is formed by summing colsum(j) ($0 \leq j \leq c-1$). This requires r*c multiplications and (r*c)-1 additions. SUM(0,1) is formed by subtracting colsum(0) from SUM(0,0) and adding colsum(c) to the result. In general:

$$SUM(0,j) = SUM(O,j-1) - colsum(j-1) + colsum(j+c-1)$$

After the processing of a given row is complete, colsum(j) is updated for the next row by subtracting Y(i,j) from the old colsum(j) and adding Y(i+r−1,j) to the result. This changes the complexity for the calculation of the $\sum y$'s to: 3c−1 additions/subtractions per template position for the column 0 entries of all other rows, and 4 additions/subtractions per template position for all other template positions.

For a typical 64-by-64 match image, straight forward computation of $\sum y$ requires 4095 additions per match template position on the input image. This is the same number of operations required per match template position in row 0 of the input image. For template positions in column 0 of the other rows, 191 additions are required. Computation of $\sum y^2$'s is similar to the computation of the $\sum y'$s.

Consider the application of MuRSS to this task. Each PU will apply the serial algorithm to its assigned pixels. Pixels will be assigned to PUs based on the vertical striping scheme. If a column of pixels lies in memory associated with bus 0 or bus 1 of PU I, then PU I is responsible for the computation of the colsum and the analogous y^2 entries associated with that column. If the pixel in the upper left hand corner of a window lies in memory associated with bus 0 or bus 1 of PU I, then PU I is responsible for the computation of that window. When PU I is performing computations on its rightmost c-1 columns, it uses the colsum values stored in its bus 2 memory by the previous computations of PU I+1 (recall that PU I+1's bus 0 memory is PU I's bus 2 memory). Thus, at least c-1 colsum values and the corresponding y values must be stored in memory associated with each bus 0.

For an R-by-C image and N PUs, a simple vertical striping scheme will assign each PU a subimage either R$-$by$-\lceil C/N \rceil$ or R$-$by$-\lceil C/N \rceil$. Thus, the total time required for the calculation of the $\sum$xy's is $(R-r+1)*(\lceil C/N \rceil-c+1)*((r*c)-1)$ additions, and $(R-r+1)*(\lceil C/N \rceil-c+1)*r*c$ multiplications. The total time associated with the calculation of the $\sum$y's is $[(R-r)*((3*c)-1)]$ + $[(\lceil C/N \rceil-c)*((r*c)-1)]$ + $[(R-r)*(\lceil C/N \rceil-c)*4]$ additions. The time required to calculate the $\sum y^2$'s is similar to the time associated with the calculation of the $\sum$y's. Extension to the modified striping scheme is similar to the smoothing case.

If $C < N*(c-1)$, then $c-1$ columns of data cannot be associated with each bus 0, thus the PUs cannot all be enabled. If $R \geq N*(r-1)$, the stripes can be horizontal instead of vertical. In this case, r and c are swapped, as well as R and C.

3.7. The Fault Tolerance of MuRSS

The throughput of a MuRSS is limited to the largest number of adjacent working (usable) PUs. Consider a simple example with an N=8 MuRSS system (a PU fault is represented by **BOLD** print in a box).

Physical: 0 1 2 3 4 5 6 $\boxed{\mathbf{7}}$ 8

A single failure leaves eight usable PUs. (If there were no wrap-around connection, the number of usable PUs would be seven.) The CU can alter the PU numbers and subsequently the numbers associated with the memory modules. Thus, for the above fault the CU would renumber the PUs to (an * indicates an unused PU):

Physical:	0	1	2	3	4	5	6	$\boxed{7}$	8
Locigal:	1	2	3	4	5	6	7	$\boxed{*}$	0

Fault detection procedures are beyond the scope of this work. In both this section and in Section 3.8., the concern is with fault recovery once the existence and location of a fault is known.

When a MuRSS processor the renumbered MuRSS PUs start with logical PU 0 to the right of the failed processor. The numbers continue incrementing, through the wrap-around connection, ending up with the virtual PU N-1 on the left of the failed processor. When a local memory module fails, e.g., $2I+1$, it is treated like a fault with PU I. A fault in a shared memory, e.g., $2I$ is treated the same way.

It is possible for the faulty processor or memory module to fail in such a way that that adjacent PUs cannot access the busses shared with the faulty unit. In such a case, not only the faulty PU but the PU associated with the inaccessible shared memory module would be unusable because of the inability to access shared memory. Thus, this would be handled as if two adjacent PUs failed. This is a special case of the multiple failure situation discussed later.

A multiple failure, such as:

$$\text{Physical:} \quad 0 \quad 1 \quad 2 \quad 3 \; \boxed{4} \; 5 \quad 6 \; \boxed{7} \; 8$$

reduces the number of usable PUs to five. (PUs 5 and 6 cannot share data with adjacent PUs, and subsequently could not be used for any algorithm requiring data to be shared among PUs.) If either PU 5 or PU 6 or both were also faulty, the same number of usable PUs would exist, as demonstrated below:

$$0 \quad 1 \quad 2 \quad 3 \; \boxed{4}\boxed{5} \; 6 \; \boxed{7} \; 8$$

$$0 \quad 1 \quad 2 \quad 3 \; \boxed{4} \; 5 \; \boxed{6}\boxed{7} \; 8$$

- and -

$$0 \quad 1 \quad 2 \quad 3 \; \boxed{4}\boxed{5}\boxed{6}\boxed{7} \; 8$$

In such an event, the PUs would be renumbered to:

Physical:	0	1	2	3	4	5	6	7	8
Logical:	1	2	3	4	*	*	*	*	0

Again, an * indicates an idled PU.

A fault in a shared memory, e.g., 2I is treated the same way. Multiple memory faults associated with the same PU I, only idle PU I. Multiple memory faults associated with different PUs idle their associated PUs and subsequently are treated like multiple PU faults.

It was previously stated that if any local data for PU I is to be stored in a shared memory module, that it should be in memory module 2I. This is required if an algorithm is to be run on a system with a single fault in one of the shared memory modules. If this rule is not followed, a fault in a shared memory bank would require the two PUs attached to a faulty shared memory module to be disabled instead of one, decreasing the throughput of the system.

The minimum number of usable PUs in an N PU MuRSS with F PU faults (or disabled PUs) can be expressed by the equation:

$$\text{Usable PUs (min)} = \begin{cases} N & F = 0 \\ \left\lfloor \frac{N}{F} \right\rfloor & 1 \leq F \leq N+1 \end{cases}$$

This minimum occurs when faulty PUs are evenly distributed throughout the system. A few faults can seriously cripple MuRSS, as is shown in Fig. 3.7.1. It is worthy of note, that this is a worst case possibility. If the failures are close together, the number of usable PUs will be greatly increased. For example, if the faulty PUs are adjacent, the number of usable PUs is N−F +1.

3.8. An Enhanced MuRSS

To minimize the degradation of MuRSS in a multiple fault environment, consider the modifications shown in Fig. 3.8.1. The wrap-around connection between PU N and PU 0 is the same as before (see Fig. 3.2.2). In this figure describing the Enhanced **MuRSS (EMuRSS)**, there is a **bypass box** associated with each PU's shared busses. The operation of the bypass boxes is controlled by the CU.

In addition to the bypass boxes, there is deselection circuitry, such as the SN74S244 [Uni78], between each shared memory module and its corresponding bus. This circuitry will be used isolate faults in the shared memory modules so that the shared busses are still usable. It is assumed that there is some form of isolation hardware, such as the SN74S244 [Uni78], between each of the memory modules (both local and shared) and the host bus to prevent a memory module from failing in such a way as to make the host to memory module bus unusable. The deselection and isolation hardware is controlled by the CU.

The effect of the bypass boxes is to allow the system to reconfigure "around" a faulty unit. Consider, an N=8 EMuRSS system where PU 7 is faulty.

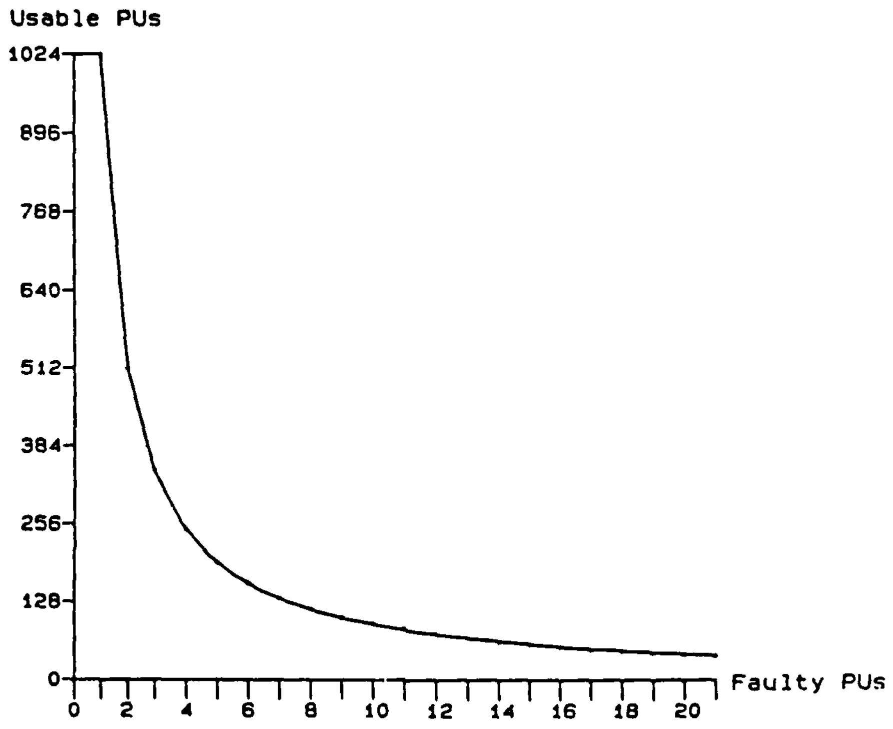


Fig. 3.7.1 Minimum number of usable PUs in a 1024 PU MuRSS versus number of faulty PUs

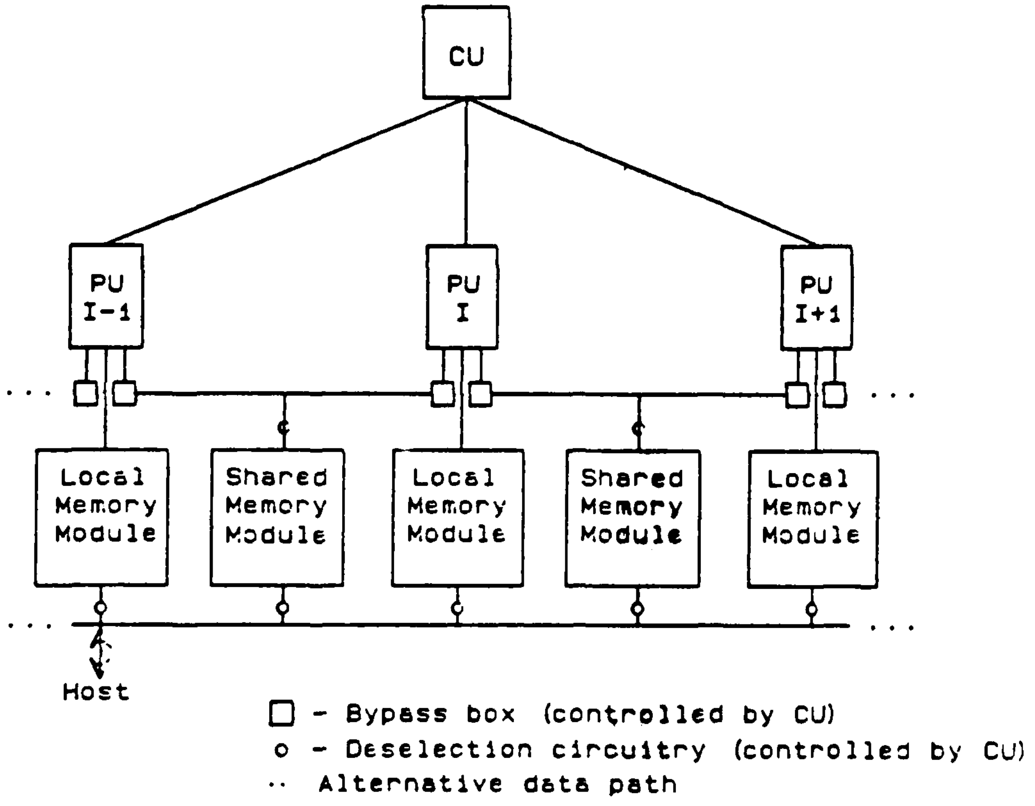


Fig. 3.8.1 Fault tolerant MuRSS system architecture

Physical:	0	1	2	3	4	5	6	$\boxed{7}$	8

In the single fault case, with the use of bypass boxes there are still eight usable PUs. When a double fault occurs, such as any of those shown in Fig. 3.8.2., the number of usable PUs is seven, because the use of bypass boxes allows the connectivity to be maintained. In a normal MuRSS, the number of usable PUs would be 6, 5, 4, 4, 5, 6, 7, and 7 respectively. Multiple (more than two) faults are handled similarly.

The two modes and corresponding effects of bypass boxes are shown in Fig. 3.8.3. These modes allow MuRSS to completely bypass a faulty PU. When there is a failure in a PU I the PU is bypassed and its associated shared memory is deselected. It is assumed that the bypass box/deselection circuitry can isolate any faulty hardware from the shared busses, allowing normal communications to take place between the two processors adjacent to the faulty PU.

The CU can re-assign the PU numbers, allowing the PUs and their associated memories to be treated like they were contiguous. As was used before, the PUs have a physical number and a logical number. The logical PU number will not only simplify the addressing by the host, but will, when combined with the "wrap-around" connection, allow the system to handle one complete shared bus or bypass box failure with no degradation.

If a single PU fails, the bypass boxes associated with its shared busses are set to bypass mode. The shared memory associated with its bus 0 is deselected. Disabling the faulty PU has the effect of disabling its local

$$\boxed{0}\quad 1\quad 2\quad 3\quad 4\quad 5\quad 6\quad \boxed{7}\quad 8$$

$$0\quad \boxed{1}\quad 2\quad 3\quad 4\quad 5\quad 6\quad \boxed{7}\quad 8$$

$$0\quad 1\quad \boxed{2}\quad 3\quad 4\quad 5\quad 6\quad \boxed{7}\quad 8$$

$$0\quad 1\quad 2\quad \boxed{3}\quad 4\quad 5\quad 6\quad \boxed{7}\quad 8$$

$$0\quad 1\quad 2\quad 3\quad \boxed{4}\quad 5\quad 6\quad \boxed{7}\quad 8$$

$$0\quad 1\quad 2\quad 3\quad 4\quad \boxed{5}\quad 6\quad \boxed{7}\quad 8$$

$$0\quad 1\quad 2\quad 3\quad 4\quad 5\quad \boxed{6}\boxed{7}\quad 8$$

$$0\quad 1\quad 2\quad 3\quad 4\quad 5\quad 6\quad \boxed{7}\boxed{8}$$

Fig. 3.8.2. Double faults in EMuRSS leaving 7 usable PUs.

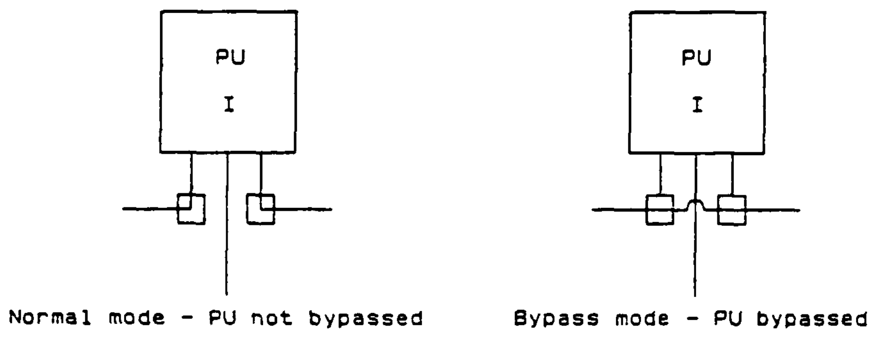


Fig. 3.8.3 Two modes of a bypass box

memory, thus contention on the host bus is not a problem. The logical PU number of all PUs whose physical PU numbers are greater than the faulty PU is decremented by one, as is shown in the following example:

Physical:	0	1	$\boxed{2}$	3	4	5	6	7	8
Logical:	0	1	$\boxed{*}$	2	3	4	5	6	7

Physical PU N (previously disabled) becomes logical PU N-1. When a memory module (either local or shared) fails, it is handled exactly like a fault with the associated PU. The wrap-around connection is not used when there is a fault with a single PU or memory module. Multiple faulty PUs are handled similarly, only in the multiple fault case, the performance is degraded as there are no more working PUs to replace the faulty units. Multiple faulty shared and local memory modules are handled like multiple faulty PUs.

A single faulty bypass box is handled using the wrap-around connection. If there is a fault with one of the bypass boxes associated with PU I, PU I is disabled. PUs with physical numbers I+1 to N are given logical numbers 0 to N−I−1 and PUs with physical numbers 0 to I-1 are given logical numbers N−I to N−1. Using the wrap-around connection places the faulty bypass box on the logical end of the array, where it and its associated PU (PU I) are unused. If, in addition to a single faulty bypass box, there are any faulty PUs or memory modules, these additional faults can be handled as described in the last paragraph.

In general, multiple faulty bypass boxes break the connectivity of EMuRSS. It is assumed that a bypass box failure does not pull down a shared bus. If it does, it is treated the same as a shared bus failure. Multiple faulty bypass boxes have the same result as multiple PU failures in MuRSS. Thus, the number of usable PUs is less than N. The set of adjacent usable PUs may or may not use the wrap around connection.

If the multiple faulty bypass boxes share the same bus, EMuRSS can handle two faults with no degradation. This is shown in Fig. 3.8.4. This is the same situation for a single faulty shared bus, i.e., the bus shared by PUs I-1 and I in Fig. 3.8.5. If the two faulty bypass boxes are connected to the same PU, i.e., bus 0 and bus 2 of PU I, EMuRSS can handle two faults with no degradation. This is shown in Fig. 3.8.6. If the faults are on contiguous busses, e.g., PU I's 0 bus, PU I's 2 bus, and PU I−1's 2 bus, up to three faults can be tolerated with no degradation in performance. This is shown in Fig. 3.8.7.

Multiple faulty busses break the connectivity of EMuRSS. This situation is the same as the case for multiple faulty bypass boxes previously discussed.

Since mechanical connections, such as those between a chip and a bus, are significantly more prone to failure than those within a chip, the number of mechanical connections can give a fair indication of the probability of failure of a unit. In MuRSS, both shared and local memory busses are connected to 2^8 64-Kbyte chips and each chip has 28 pins, so (including the 64 pins on the 68000) there are a minimum of 14400 mechanical chip connections that can cause a fault within each PU and its associated memories (only those busses associated with a PU are considered). This figure is clearly conservative

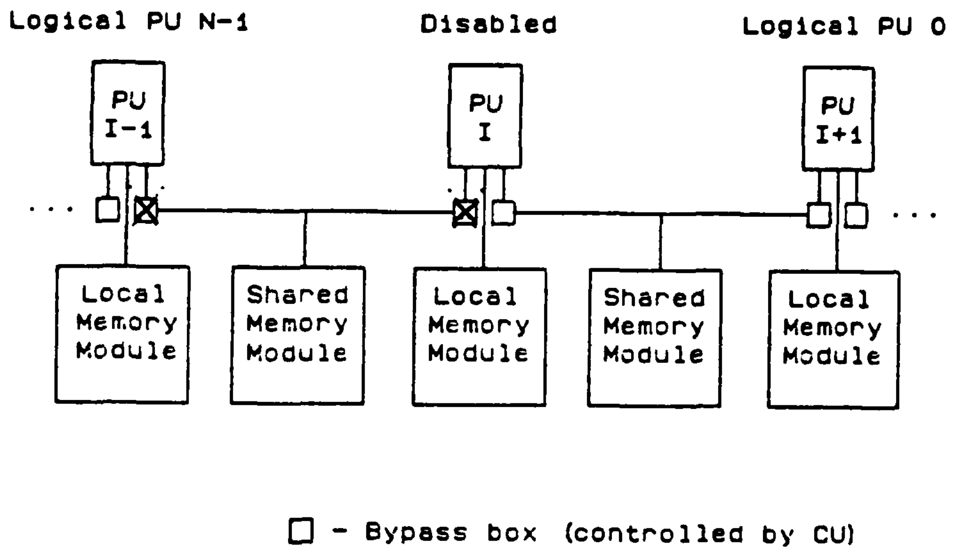


Fig. 3.8.4 EMuRSS reconfiguration around two box faults on same shared bus

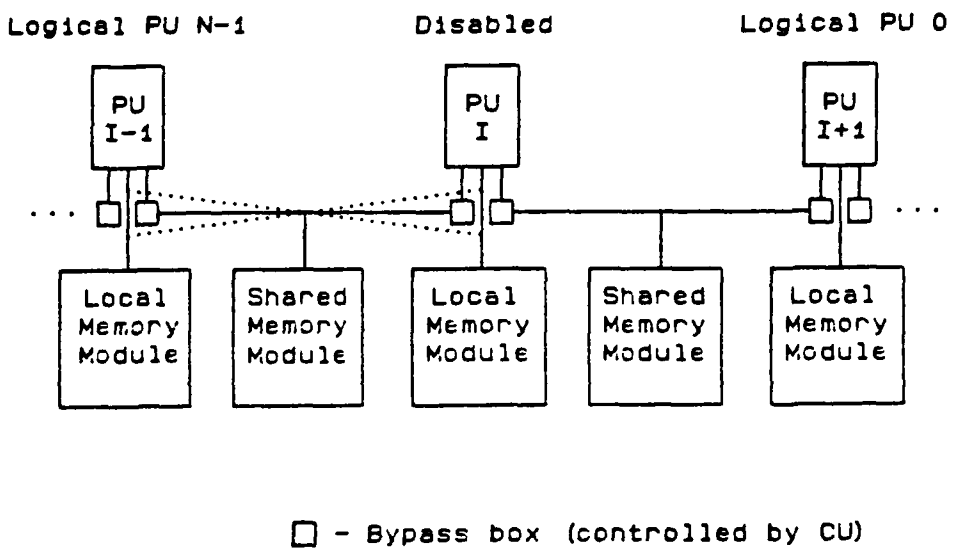


Fig. 3.8.5 EMuRSS reconfiguration around faulty shared memory bus

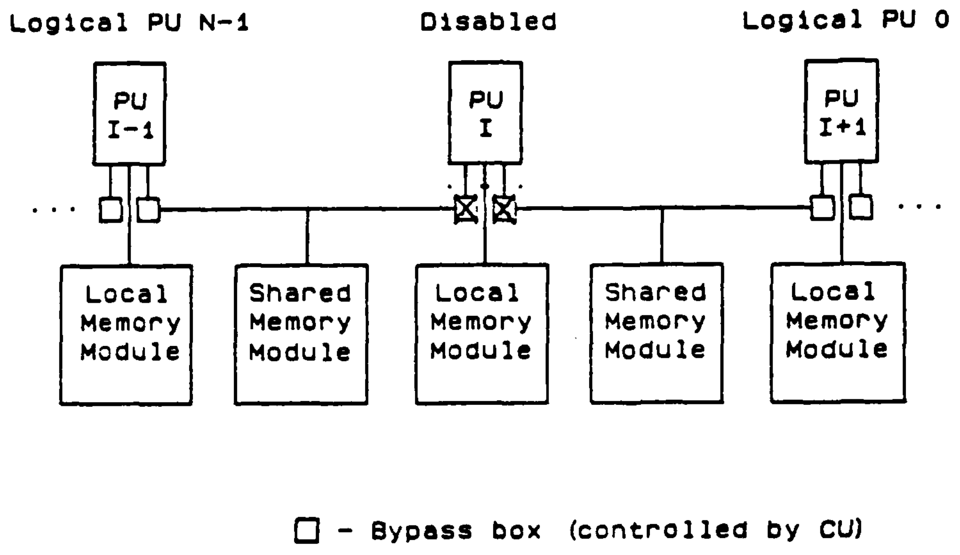


Fig. 3.8.6 EMuRSS reconfiguration around two bypass box faults associated with PU I

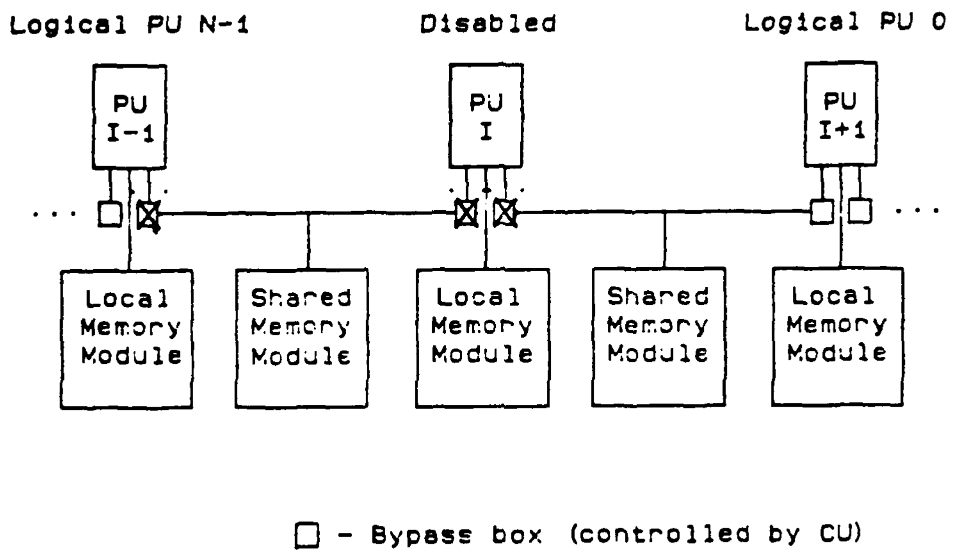


Fig. 3.8.7 EMuRSS reconfiguration around three adjacent bypass box faults

because a failure in any support hardware (e.g., the CU) will also cause a fault. For simplicity, only chip connections (pins), as opposed to chip connections and bus connections (e.g., connections from busses to boards), will be used for this discussion. MuRSS (N=1024) has 14,745,600 mechanical chip connections.

The fault bypass circuitry in EMuRSS consists of the bypass box, the bus performing the bypass, and shared memory unit deselection hardware. Thus, there are chip connections to the CU, **PU, shared bus, shared memory, bypass bus,** and **deselection circuitry** that can fail. The connections in bold print are to busses with 26 connections for address, 8 connections for data, 4 connections for signals, and 2 connections for power and ground. This comprises 200 connections. The CU must have one line to control each bypass box and one line to control the memory deselection circuitry, making 202 mechanical chip connections that can cause a fault. The processor/memory hardware is 88 times more likely to fail due to a mechanical connection than the bypass circuitry. EMuRSS has 14,952,488 mechanical connections. This represents an increase in hardware complexity of 1.4 percent over the non-fault tolerant MuRSS, which is a trivial change in the complexity of the system when it is compared to the additional fault handling capability of the system.

The 1.4 percent figure does not accurately represent the fault tolerance of the EMuRSS. A fault in any two of the 14,680,014 connections will yield up to half the system unusable. Thus, these connections can be labeled as critical to the system's operation. EMuRSS has 204,800 connections. This represents a significant decrease in the number of critical connections.

To compare the performance of MuRSS to EMuRSS, consider the following example. If **UPWBB** is the number of **U**sable **P**U's in an N+1 PU MuRSS **W**ith **B**ypass **B**oxes, UPWBB = N − F + 1, where F is the number of faults in the system ($1 \leq F \leq N+1$). (If $F=0$, UPWBB=N.)

For $F < N+1$ faults, the number of **U**sable **P**Us in a system with **N**o **B**ypass **B**oxes **(UPNBB)** would be no less than $\left\lfloor \frac{N}{F} \right\rfloor$. The benefit of the bypass units is demonstrated in Fig. 3.8.8, where UPWBB/UPNBB is graphed with respect to F. The "sawtooth" nature of this graph stems from the floor function in the definition of UPNBB. At no time is UPWBB less than UPNBB, but for an N=1024 PU system, UPWBB can be up to 512 times greater than UPNBB.

Thus for a small increase in hardware complexity, the degradation in the system performance due to multiple faults can be significantly reduced (by up to a factor of 512 on a 1024 processor system).

3.9. MPP -- A Massively Parallel Processor

For the basis of comparison, consider the **M**assively **P**arallel **P**rocessor **(MPP)** as described in [Bat82] and [Bat80]. MPP is an SIMD machine which was designed to work efficiently on a variety of image processing tasks, such as correlation and multispectral classification. Fig. 3.9.1 is a block diagram of MPP, which illustrates the four major sub-units. The **AR**ray **U**nit **(ARU)** is the unit that actually contains the **P**rocessing **E**lements **(PEs)**, and is capable of processing arrays of data at high speed. Each of the PEs in the ARU performs instructions broadcast by the **A**rray **C**ontrol **U**nit **(ACU)** on data that are stored in local memory.

Logically, the ARU consists of a 128-by-128 array of PEs. Physically, the ARU contains an extra 128-by-4 array of PEs for fault tolerance. The size of

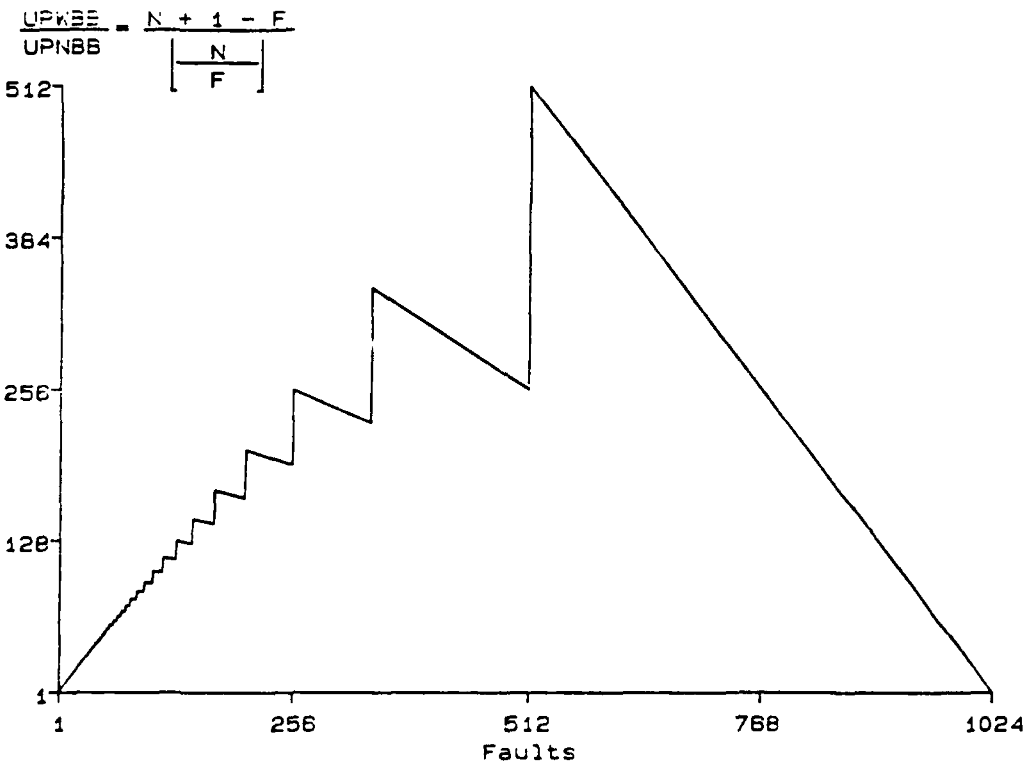


Fig. 3.8.8 Ratio of usable PUs in EMuRSS to usable PUs in MuRSS

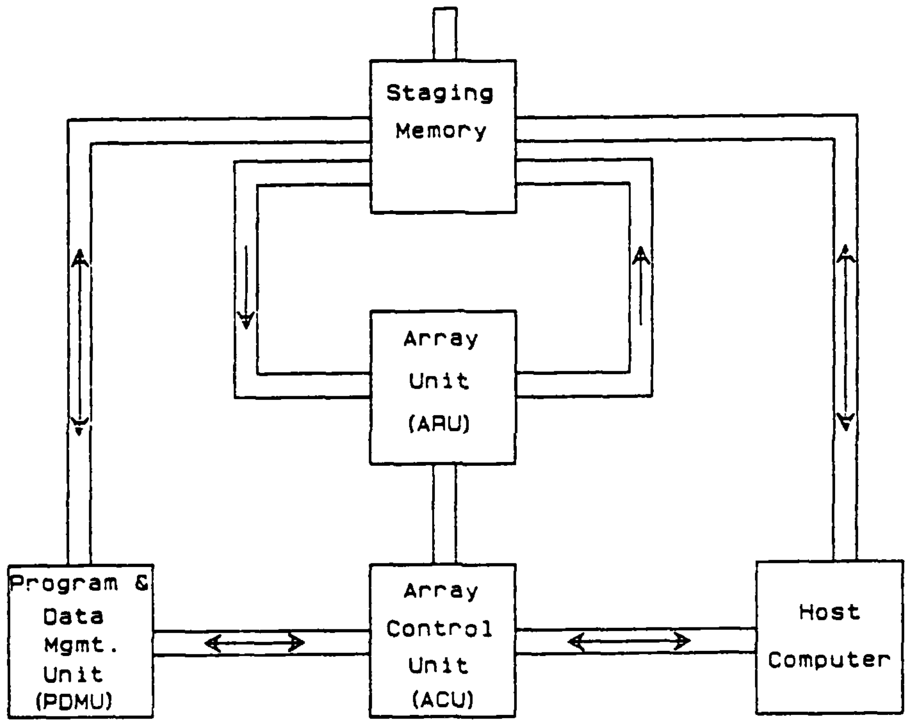


Fig. 3.9.1 Block diagram of MPP ([Bat80],[Bat82])

the extra 128-by-4 array was determined by packaging constraints. The bit-serial nature of the PEs allows MPP to perform efficiently on operands of all lengths. The 16,384 PEs operate instructions on 16,384 bits at a time, which allows for a very high processing speed.

Each PE in the 128-by-128 array communicates with its four nearest neighbors in a fashion similar to ILLIAC IV ([Bar68] and [Bou72]). A topology register in the ACU allows the user to software select what happens to edge data in the ARU. Top-bottom connections in the ARU are handled independently from the left-right connections, allowing the user greater flexibility. There are four possible connections that a PE, call it PE_I on the right edge of the array can make in addition to connecting to $PE_{I ciminus 1}$, $PE_{I ciminus 128}$, and $PE_{I \oplus 128}$. where $\oplus$ and ciminus are modulo 16384 addition and subtraction respectively. They are:

1) open (no connection)

2) connect to PE_{I+1}(PE_{16383} has no connection)

3) connect to left edge PE of same row

4) same as 2), but connect PE_{16383} to PE_0

The connections for left edge PEs correspond to these connections. Top-bottom connections are less complex than the left-right connections, in that the top and bottom PEs of a given column may be either connected or left open.

Each PE in the ARU contains a full adder, a shift register, six 1-bit registers, a programmable length shift register, 1K-bits of RAM, a data bus, combinatorial logic, and a mask register. Fig. 3.9.2 shows the layout of the PE. 100ns is the basic cycle time for the PE; however, routing operations are

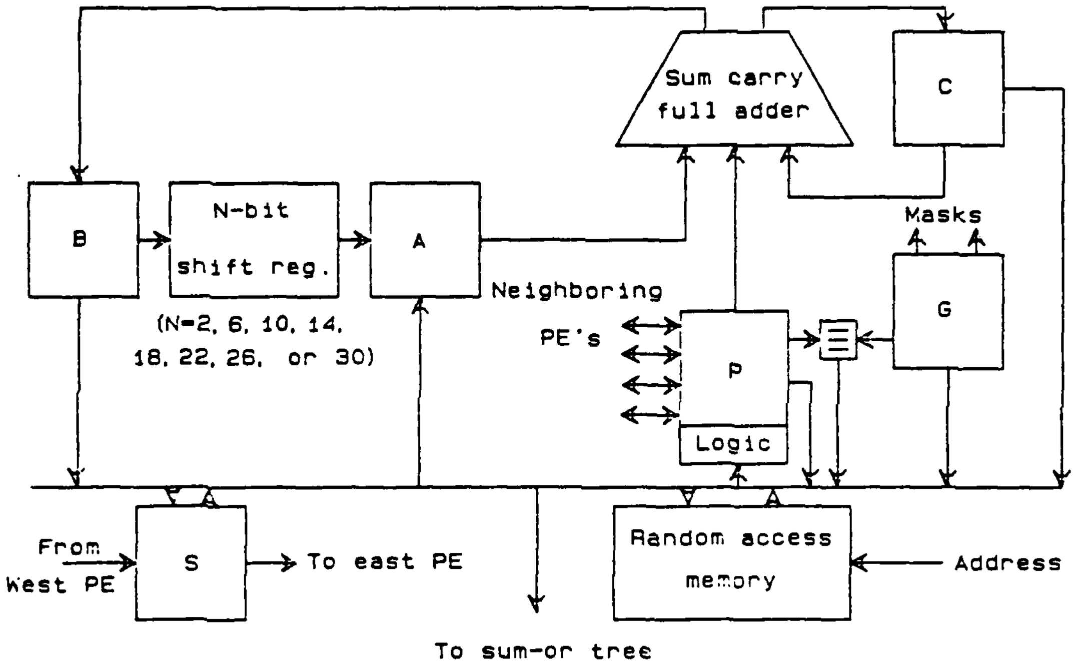


Fig. 3.9.2 MPP PE architecture ([Bat80],[Bat82])

masked independently of arithmetic operations, so masked routing operations may be combined with unmasked arithmetic operations. PEs perform the instruction generated by the ACU on the data stored in their local array. Fig. 3.9.3 is a block diagram of the ACU. The three units comprising the ACU are the I/O control unit (which manages the flow of data), the PE control unit (which performs array arithmetic for the applications program), and the main control unit (which performs scalar arithmetic for the applications program). Operations of each of the units are overlapped to minimize execution time.

The **P**rogram and **D**ata **M**anagement **U**nit **(PDMU)** controls the overall flow of programs and data in the system (Fig. 3.9.1), and is comprised of a DEC PDP-11. The staging memory are used for format conversion between the incoming data and the data to be processed. Once the data has been processed, it is returned to the staging memory, where additional formatting can be performed.

Through its massive parallelism, high clock rate, and its functional overlap, MPP is capable of performing 400 million 32-bit floating point additions per second, 200 million 32-bit floating point multiplications per second, or 3277 million 16-bit integer additions per second.

It is difficult to compare the cost of EMuRSS and MPP, since MPP is constructed of specially designed VLSI chips and EMuRSS would not be. The complexity of this comparison is compounded by the fact that hardware costs change so rapidly. Therefore, the comparison will be limited to the area of processing speeds, fault tolerance, and capabilities of MPP and a 1024 processor EMuRSS, both of which process 16,384 bits at a time. The purpose of this comparison is to highlight the differences in the two architectural approaches.

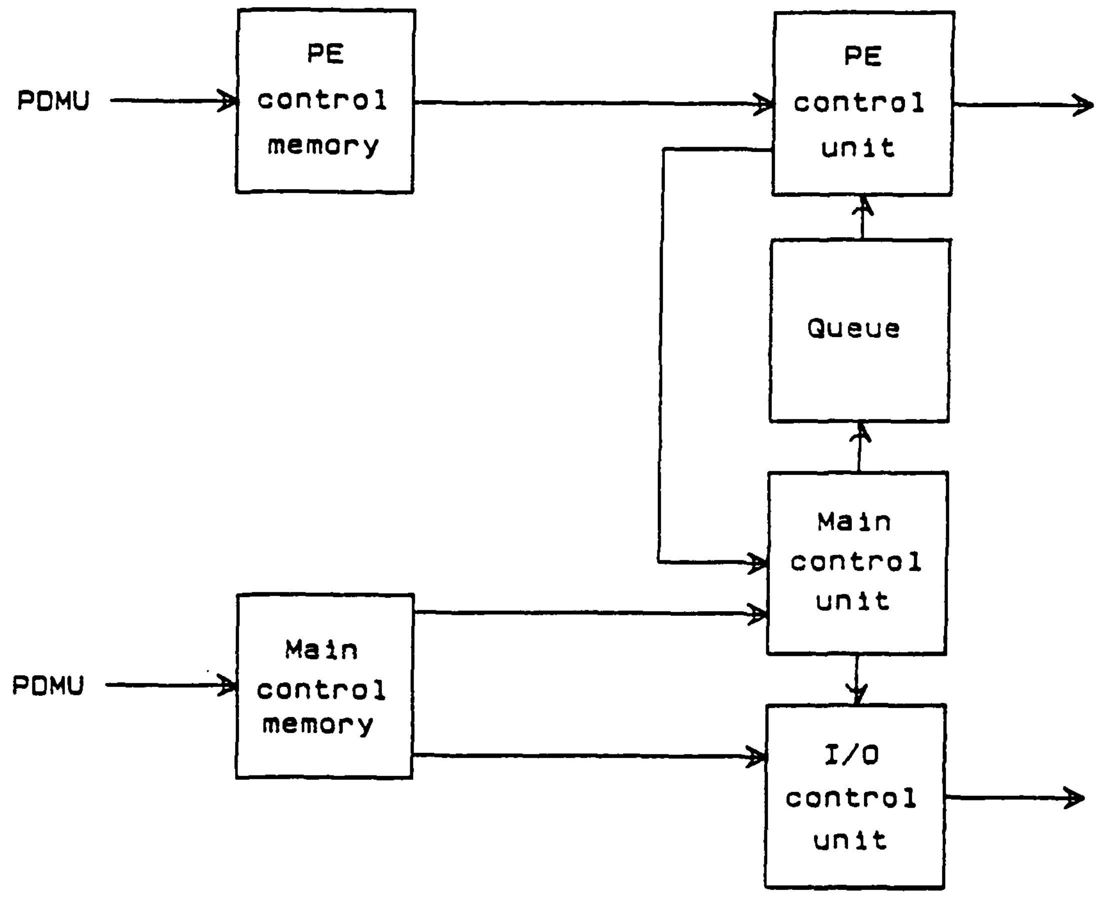


Fig. 3.9.3 Block diagram of the ACU ([Bat80],[Bat82])

EMuRSS can perform 3072 million 16-bit integer additions per second and MPP can perform 3277 million 16-bit integer additions per second. MPP is 6 percent faster than EMuRSS. EMuRSS can perform 307.2 million 16-bit integer multiplications (yielding a 32-bit result) per second. MPP can perform 1861 million 8-bit integer multiplications (yielding a 16-bit result) per second and 902 million 12-bit multiplications per second (yielding a 24-bit result). 32-bit data was not available.

In terms of floating point operations per second, MPP outperforms the EMuRSS without the floating point processor (both MuRSS and EMuRSS have the same processing speeds). The cycle time for MPP is 100 nsec, while the cycle time for EMuRSS is less than 80 nsec. (see Section 3.2), so it would be intuitively pleasing if EMuRSS outperformed MPP. Both processors operate on 16,384 bits of information at a time; however, in all cases MPP will require the minimum number of cycles for a given operation for a given number of bits because of its bit serial nature. For example, a typical 32-bit floating point format consists of:

a sign bit for the mantissa,

an 8-bit 2's complement exponent, and

a 23-bit mantissa.

A 68000 can perform operations on 16-bits of information at a time, so operations on the 23-bit mantissa require the same time as operations on a 32-bit mantissa. Operations on the 8-bit exponent require the same time as operations on a 16-bit exponent. Further, the EMuRSS processors have to strip out unwanted data at the end of each operation, whereas the MPP processors have little or no unwanted data.

The specialized floating point hardware eliminates much of the overhead involved with the handling of unwanted data. This is why EMuRSS, when equipped with the floating point processor, becomes very similar in performance to MPP. For a 32-bit floating point addition, EMuRSS is 9 percent slower than MPP, but for a 32-bit floating point multiplication, EMuRSS is over 56 percent faster than MPP. Further, the EMuRSS specialized floating point processor has hardware implementations for sine, cosine, and tangent, all of which must have custom programs written for their calculations on MPP. Further, each floating point processor is independent from the other floating point processors in EMuRSS, i.e., they are not synchronized. Thus, no processor must be idled for any point in time during these calculations to wait for another processor to finish a calculation whose execution is data dependent, e.g., to perform a cosine no synchronization is required during the intermediate computations. This makes EMuRSS even more competitive with MPP because using the algorithms in [Har68], there are conditional instructions that are required for the calculation of the trigonometric functions.

To be able to tolerate a single fault with no degradation in response time, MPP uses an additional 4-by-128 array of PEs. A one PU EMuRSS equipped with the bypass boxes discussed earlier requires one additional PU to be capable of withstanding the fault of a single PU without loss of processing speed. Both MPP and EMuRSS require some form of bypass hardware to bypass a fault.

MPP and EMuRSS are tolerant to a single fault. MPP is not tolerant to multiple faults, unless they are all in the same 4-by-128 array of PEs that is bypassed. The number of usable PUs in EMuRSS is one more than N minus the number of failed PUs (since an N-PU EMuRSS has one spare PU). Thus, in general, in the event of a multiple fault, EMuRSS can continue to be used with a minimal degradation in performance. For MPP, there is no provision for operation in a degraded mode when multiple faults occur.

Any of the inter-processor nearest neighbor communication operations that MPP can perform can also be handled by EMuRSS. MPP can process images by assigning one pixel to each PE, or by dividing the image to be processed into square neighborhoods that are processed by the PEs. For an M-by-M image, each PE would hold subimages that are M/128 pixels on a side. An image to be processed by EMuRSS must be divided into stripes extending from the top of the scene to the bottom. Any inter-row communications in MPP are internal to a PU in EMuRSS. Any inter-column communications in MPP are either internal in a PU in EMURSS or are between adjacent PUs using the shared memory. Adjacent processors will process adjacent stripes.

Both EMuRSS and MPP have a memory organization that will allow an external processor to store information in one order and the processors to read the information in another, without significant processing. MPP uses the staging memory to perform image transformations and formatting for input and output. Because of the way the EMuRSS host accesses the memory, either row or column format data can be loaded.

Architecturally, EMuRSS and MPP differ in the processor-to-processor connections. EMuRSS does not have a true interconnection network. Instead, EMuRSS implements a network with shared memory banks. This technique allows memory to be used for both storage and communication, meaning that no special communication protocol is necessary. Data transfer is treated like a memory write.

In conclusion, MPP is faster than EMuRSS (with the floating point hardware) on fixed point operations and some floating point operations. EMuRSS compares reasonably with MPP on floating point multiplication and division. EMuRSS has a hardware unit capable of performing floating point trigonometric and inverse-trigonometric functions. Because the floating point units are not run in lock-step, for any floating point operation, e.g., steps during the calculation of cosine, EMuRSS effectively becomes an MIMD machine, whereas MPP must perform these operations in lock-step.

Any processor-to-processor communication that is required for an MPP implementation of an algorithm can be handled by EMuRSS. Both MPP and EMuRSS can handle a single fault with no degradation in performance; however, only the fault-tolerant EMuRSS can handle multiple faults (with some degradation).

3.10. Conclusions

MuRSS, an SIMD architecture with as many as 1024 processors, was presented. It was shown that N processors in the SIMD mode of operation could perform various context independent (e.g., maximum likelihood classification) and window based (e.g., smoothing, contextual classification, and image correlation) image processing tasks almost N times faster than one processor of the same type. The application of MuRSS to these tasks was discussed.

Through the use of the EMuRSS SIMD architecture, computationally demanding remote sensing processes can be implemented efficiently. This will not only reduce the computation time required to perform remote sensing tasks, but will also allow the investigation of techniques which may otherwise be considered infeasible.

Because of the architecture of MuRSS, multiple faults seriously degraded its performance. The architecture of MuRSS was altered to increase MuRSS' tolerance to faults, creating EMuRSS. EMuRSS was then compared to MPP in the areas of performance, fault tolerance and capabilities.

CHAPTER 4 MODELS FOR USE IN THE DESIGN OF SPECIAL PURPOSE MACRO-PIPELINED PARALLEL PROCESSORS

4.1. Introduction

For certain applications, such as speech processing, time is an important factor. In such applications, there is a need to process many data sets in the same way e.g., performing an FFT for every frame of input data. Previous analysis, such as that performed in [Dem83, TuA83, YoS82, Vic79], shows that for many types of tasks, a general purpose processor is not sufficient. In this chapter, an approach is proposed for modeling off the shelf hardware and for modeling parallel algorithms, along with a design methodology to use the information provided by these models, to design a class of macro-pipelined special purpose parallel architectures. The goal is to use models such as the ones proposed here to develop computer aided design tools.

Special purpose processing systems (such as those used for dedicated real-time analysis) are typically sold in small quantities. As a result, the cost of the design can make the resulting system prohibitively expensive. Computer aided design tools for this process would reduce the cost involved and are therefore desirable.

This chapter uses nine parameters to correlate the hardware to be designed to the applications software to be executed and the I/O environment in which the machine is to operate, i.e., what data rates the machine must handle, the format of the incoming data, the format of the outgoing data, etc. A macro-pipelined layered approach to task decomposition is demonstrated. Each portion of the decomposed task in a scenario is then assigned to a specifically designed special purpose processing unit. This implies that each processing unit may either be a traditional serial type design or a parallel design. Once this initial decomposition is established, techniques such as those used to adjust the execution time and throughput of a pipeline in [HwB84] can be applied.

In this approach to reaching the goal of automated computer design, a functional descriptions (models) of the hardware components that may be used in the design must be combined into a database. Included in such a database is information about the cost, size, power consumption, and heat dissipation of the device, an enumeration of all the operations that it can execute, the pathwidth and execution times for those operations, the number and size of the registers, and a simulation routine for the device. More complex taxonomies, such as those found in [Han77], [Han81], [HoJ81], and [Gil83] are not needed for the database because they specify architectural information. Here, only information that affects the processing speed of the unit are considered. While the architectural information provided by more complex taxonomies can yield similar information, handling of the additional data is cumbersome.

The information in the database will be used to select the "best" hardware to execute a given algorithm. As suggested in [Gon78], it is desirable to establish and order according to importance, the criteria used to rank designs.

The criteria used here will be (in order of importance): speed and cost. Speed refers to both response time and throughput. The **response time** is the time between receiving the input and transmission of the corresponding result. The **throughput** is the number of data sets processed per unit time. Other criteria might include: space, power requirements, and cooling requirements.

Using information about each sub-task in a scenario, a specific hardware organization can be arranged to execute the required algorithm when possible. Consider a task that is composed of several sub-tasks. An example of such a task might be isolated word recognition [YoS82]. For isolated word recognition, a typical processing scenario might be: digital filtering, autocorrelation analysis, linear predictive coding (LPC) analysis, linear time warping, and dynamic time warping. Each of these processes (sub-tasks) represents a portion of the scenario. An example of the scenario is in Fig. 4.1.1. Each of the sub-tasks will be called a **layer.** Using information about each sub-task, a special-purpose architecture can be developed to execute the sub-task within some time and cost constraints. The special-purpose hardware that is assigned to each layer will be called a **level.**

For the present, only a simple scenario (one in which there is no feedback) is considered. Initially, the sub-tasks will be chosen according to conceptual differences, i.e., digital filtering is different from autocorrelation analysis, so each should be a different layer. It is assumed that in general, conceptually different portions of the task, i.e., the sub-tasks, require different hardware resources. A more complete discussion of the application of such a design to an isolated word recognition system may be found in Section 4.6.

It is the goal of this scheme to achieve a higher throughput by decomposing a scenario into layers. Because each layer requires fewer

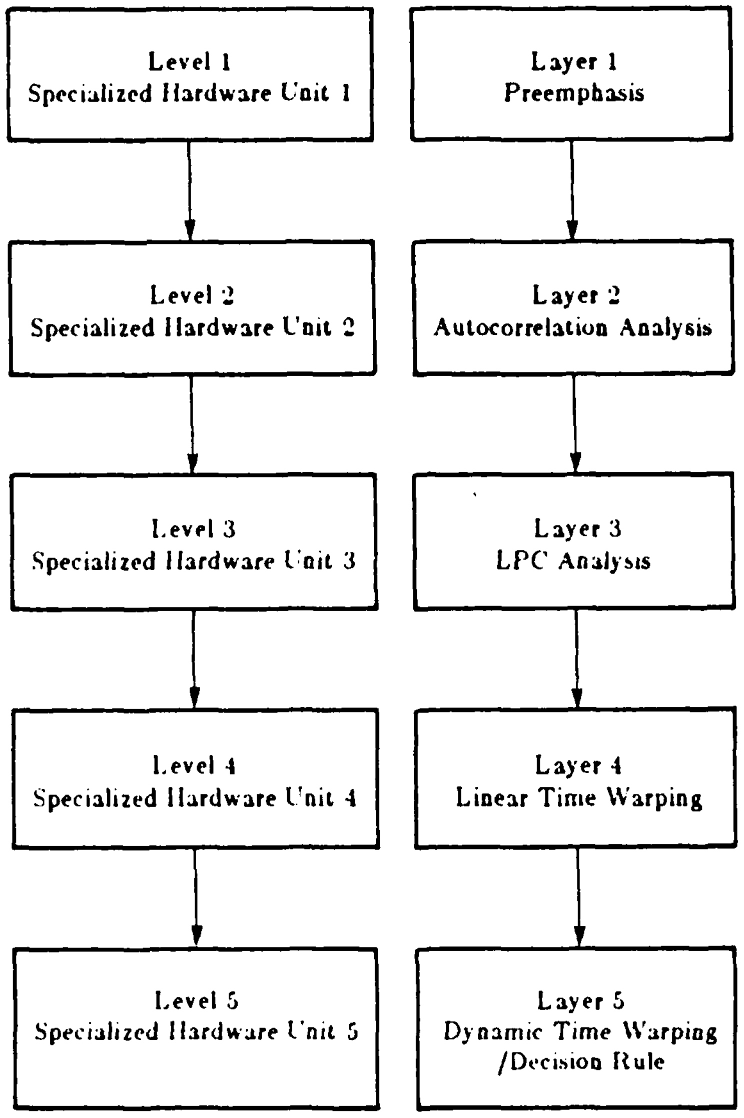


Fig. 4.1.1 Layering of isolated word recognition system

computations than the entire scenario, connecting the levels in a macro-pipeline and pipelining the data sets through the machine should increase the throughput of the resulting system. This type of parallelism is referred to as **vertical** parallelism. Since each layer is executing on specially designed hardware, which may consist of multiple computational units, the response time of the resulting system is decreased. The parallelism occurring within a given level, where multiple units are performing operations on different portions of the data set simultaneously, is referred to as **horizontal** parallelism. Vertical and horizontal parallelism are similar to the techniques of subdivision and replication discussed for pipelines in [HwB84] or the "purely pipelined" and the "purely parallel" architectures discussed in [WoC84]. Throughput constraints may require that a layer be further divided into smaller processes. These will not represent new layers, but **sub-layers,** which will correspond to **sub-levels** of hardware, consistent with the previous nomenclature.

It is possible to sub-divide the layers to the point where each sub-level performs exactly one instruction. The result would be a special purpose, dedicated, instruction-level, data flow machine, capable of performing only a single task. A minor alteration in the program would require an alteration in the hardware. For all but the least complex scenarios, the hardware cost would be overwhelming. Analogously, layers can be combined to the point where one level performs an entire task. This is the case with a traditional serial machine. Presumably, the throughput of such a machine would be too small.

By developing a method to transform a task description into a potential macro-pipelined architecture, a machine can be built with the necessary characteristics to execute the task quickly and without excessive amounts of hardware. A basis for such a method is examined in Section 4.5. A similar goal can be found in [WoC84], where the goal is that of an automated tool for planning and integrating signal processing systems in a distributed computing environment. [WoC84] examines the performance of a system to satisfy requirements for throughput and robustness with respect to hardware allocation strategies, i.e., how can processors be added or deleted from a system to optimize performance. A valuable result from the work in [WoC84] is the detailed analysis of the resulting system. These techniques can also be applied to load balancing between processors. The type of systems that are considered in [WoC84] are either purely parallel (SIMD or MIMD) [Fly66], or purely pipelined. A purely parallel system corresponds to the parallelism within a level (horizontal parallelism), while a purely pipelined system corresponds to the level to level and sub-level to sub-level relationships (vertical parallelism). Thus, this research is a useful tool in the analysis of both the macro (level to level) and the micro (within a level) characteristics of the system. Here, the major concern is the underlying concepts behind a model relating specific algorithms to the requirements they place on hardware. The research here expands on the work in [WoC84] by allowing both forms of parallelism at any level.

The analysis categories in [WoC84] can be applied to any given level that contains one or more combinations of these parallel types. This will allow each level to be designed for a specific sub-task, having a special hardware complement to more quickly execute that sub-task, resulting in a machine that can complete a processing scenario within some time constraint. For the case to be discussed in Sections 4.6 and 4.7, the time constraint will be that the proposed system must understand isolated words in real-time.

It is the goal of this chapter to introduce methods of modeling hardware and algorithms so that an accurate estimation of the execution time of an algorithm is possible. The proposed hardware database is discussed in Section 4.2. Response time and its relation to the system hardware is considered in Section 4.3. Section 4.4 will discuss the two types of parallelism and their affect on the overall performance of the system. Section 4.5 will present nine parameters and discuss their relationship to the hardware of the corresponding level. In addition, the parameters are related to the application software of the corresponding layer. By applying both of these relationships, the software can be related to the hardware. This is done in Sections 4.6, 4.7, and 4.8, where the concepts discussed in Sections 4.2 through 4.5 are applied to an isolated word recognition system.

4.2. The Hardware Database

A processor description in the database consists of an 9-tuple, a 6-tuple, and a set of three N-tuples and three N+1-tuples, where N is the number of assembly language instructions (the "+1" includes the instruction fetch unit, which can, on some systems, overlap execution with certain instructions). The 9-tuple consists of the processor name, cost, package size, thermal dissipation requirements, power requirements, clock speed, data pathwidth, address pathwidth, and virtual address space. The package size, thermal dissipation, and power requirements, are included for applications, such as those aboard a satellite, where information about all three categories may be crucial. For some processors, such as the PDP-11/70, the virtual address space and the real address space differ, so both are required for specification of the processor.

The 6-tuple consists of the size and speed of on-board cache, the size and speed of on-board memory, and the number and size of the registers. The N- and N+1- tuples must provide information about: the type of machine instructions, the execution time for a single operation for each instruction, the number of stages in any pipelines, the replication of units, and the overlap of operations. The tuples corresponding to the last three information categories are N+1-tuples to account for any pipelining, functional overlap, and parallelism that can occur within the instruction fetch unit. By combining the information contained in the various tuples, it is possible to derive a precise estimation of the execution time of all operations whose times are constant, (e.g., floating point operations on units like the AMD9511A, require variable amounts of time to execute the same operation on different arguments, thus only an estimation or expected processing time may be derivable). By combining information in different tuples, much information can be gained. For a simple example, by combining the number of stages in a pipelined unit with the single operation execution time of the unit, it is possible to determine the throughput of the unit.

Because different processors have different instruction sets, it is logical that N not be the same for all processors. Consider the case of a simple processor with an instruction set consisting of an 8-bit add, a 16-bit add, a return on zero, a move memory to register (8-bit), and a move register to memory (8-bit). The 9-tuple would look like:

(BRAND/MODEL,$5.00,1.5cm-by-3.0cm,1.5-BTU/hr,
0.15-W,1.3-mu−sec,8-bits,16-bits,16-bits)

Each element of the above tuple corresponds to the above enumeration of the elements of the 9-tuple. For a simple processor, like the 8085, the 6-tuple would be:

(0,0,0,0,(1 8-bit, 3 16-bit(8-bit)))

The four "0's" show that there is no on-board cache or memory, while the parenthesized quantity associated with the 16-bit register width shows that the three 16-bit registers can be addressed in 8-bit units. For a processor that is capable of performing the above instructions (which are a small subset of the instruction set of the 8085), the 5-tuple describing the capabilities would look like:

(8-bit add register to register,
16-bit add register to register,
return if zero,
8-bit move memory to register,
8-bit move register to memory)

Both the source and destination of each operation must be enumerated. This allows for processors (like the 8085) in which the results of a given operation must go to a specific place (the accumulator). The information in the i^{th} element in each of the following tuples corresponds to the i^{th} element of the tuple enumerating the instruction set of the processors. There should be some closed form of notation for this section, for example: iaddXX could be used to represent an integer addition that is XX bits wide. Such a notation would allow the same assembly code to be used on various machines supporting similar operations, this would replace the requirement of knowing the assembly language for each unit in the data base, with knowing one generic assembly language.

The 5-tuple describing single operation execution time of the operations is as follows (all times are in processor cycles):

(5,10,(5/11),7,7)

This 5-tuple describes the information about the time to execute each of the above operations. By describing all the operations of the processor in basic clock cycles, the description of improved versions of a processor can be easily added to the database. For this example, the "return if zero" (third element above) command is associated with two times. This corresponds to the execution time of the conditional if it is false/true. The information in this tuple, combined with the number of times each specific assembly language instruction is executed, provides a worst case timing analysis for a given processor.

The next 5-tuple contains the number of fetches required to execute each operation. This is needed to help describe systems where the instruction fetch can be overlapped with the actual execution of an instruction. For this particular processor, this tuple would look like:

(1,1,1,1,1)

To account for a unit with internal pipelining, the third tuple will contain the number of stages in the pipeline for each operation the processor can perform. When a specific command is not pipelined, the number of stages in the pipeline is 1. (The following tuples must also take the unit performing the instruction fetch into consideration.) Thus, if the 8- and 16-bit addition units were 5-stage pipelines and the rest of the unit was not pipelined, the 6-tuple describing the pipelining would look like:

(5,5,1,1,1)

It is possible for a processor to have two processing units that execute the same operations simultaneously, like a micro-processor equipped with two adders. If for the previous example, there were four adders, two for 8-bit operations and two for 16-bit operations, the next 6-tuple would look like:

(2,2,1,1,1,1)

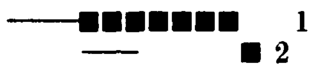
Finally, functional overlap between operations, must be considered. This is done in the final tuple that would look like:

{ memory-register/16-bit add/fetch,
memory-register/8-bit add/fetch,
-- ,
8-bit add/16-bit add,
8-bit add/16-bit add,
8-bit add/16-bit add }

This 6-tuple shows that the 8-bit add can be overlapped with both memory-register operations and the 16-bit add. The 16-bit add can be overlapped with the memory-register operations and the 8-bit add. For this example, the return if zero command cannot be overlapped with any operations, the memory register operations can be overlapped with both addition instructions, and the instruction fetch can be overlapped with the arithmetic operations.

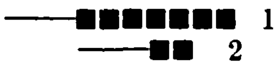
These last four tuples are used to obtain tighter limits on the execution time a given processor will require to execute a given algorithm. For example, if the instruction fetch cannot be overlapped with the execution of any instruction, the previously discussed maximum execution time discussed is a good approximation of the actual execution time. By not allowing the fetch to overlap with any instruction, each instruction must reach completion before fetching the next instruction. This eliminates any possible functional overlap.

If the instruction fetch can be overlapped with the execution of given operations then whenever the execution time for those operations exceeds the time for the fetch of the next operation, the fetch time for the subsequent operation can be deducted from the maximum execution time yielding T_f. Consider the following example (the bold wire represents execution time, the narrow represents fetch time)



Since the execution time of the first operation is overlapped with the fetch time of the second, the second operation can begin at the termination of the first operation, effectively eliminating the fetch time.

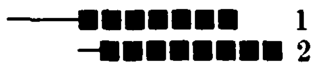
Whenever the execution time of a given operation exceeds the fetch and execution time of following overlappable operations, the execution time of the subsequent operations may be deducted from T_f to yield T_{oe}'.



The fetch overlap was taken into account in the calculation of T_F.

If a unit is not **busy,** i.e., can accept input, and its execution can be overlapped with any currently executing instructions, it is possible to overlap the instruction with the presently executing instructions. An **overlappable** operation is any operation that can be overlapped with the execution of any pending operations and that does not use, any operand that is not complete when its instruction execution begins. It should be noted that multiple units, such as adders and boolean logic units, will not decrease the execution time of an operation. Replication of hardware units will mean that there is a larger pool of units available, i.e., the likelihood of a busy unit is decreased, so the likelihood that there is a unit available for a given operation is increased. A similar effect is noted for pipelined units, where if T_{so} is the time that the pipeline requires for a single operation and there are S stages in the pipeline, the pipeline is available to accept an input in time $\frac{T_{so}}{S}$. Greater depth in the analysis of the timing of parallel and pipelined processing units, can be found in [Che80].

If the execution time of an operation is exceeded by the fetch and execution time of the subsequent overlappable operation, the execution time for the first operation can be deducted minus the fetch time for the subsequent operation is subtracted from T_{oe}' to yield T_{oe}. This is demonstrated as follows (again the thin line represents the instruction fetch and the bold line represents the instruction execution time).



The above list of parameters used to describe processing hardware is by no means conclusive, but it does serve as an example about the type of information that must be stored about processing hardware in the hardware database. It may, for example, be necessary to add an N-tuple describing two or more units that share a common pipeline.

The first two 5-tuples can be related to the notation in [HoJ81], by:

$$E = \left\{ B(2 + ^{5}_{8} \, , \, 2 + ^{10}_{16} \, , \, \mathrm{load}^{7}_{8} \, , \, \mathrm{store}^{7}_{8})/B(\text{compare and jump }\,)^{5/11} \right\}$$

where E is an execution unit and B is a boolean unit. For example, the notation $2 + ^{5}_{8}$ shows that there are two 8-bit addition units that take 5 cycles to produce a single result. The third tuple shows the relative construction of the units and would be specified in [HoJ81] notation as follows:

$$\left\{ + ^{5}_{8} = +1^{1}_{8} - +2^{1}_{8} - +3^{1}_{8} - +4^{1}_{8} - +5^{1}_{8} \right\}$$

$$\left\{ + ^{10}_{16} = +1^{2}_{16} - +2^{2}_{16} - +3^{2}_{16} - +4^{2}_{16} - +5^{2}_{16} \right\}$$

The +1 +2 +3 +4 +5 represent the various stages of the pipeline, while the superscripts and subscripts are used to describe the execution time and pathwidth of the units. Finally, the –'s show that the units are connected in series, showing that there are five stages in the pipelined adders, each stage taking one cycle. A representation of the other functions is not necessary because the third tuple shows that these units are not pipelined. By including the tuples in the database, it is possible to completely re-create the desired timing information stored in the architecture description notation set forth in [HoJ81].

In addition to the functional description, each device may be classified according to its use. The categories useful for this database (as stated in [ArB82]) are: processor, memory, input/output units, vector processors, and array processors. Application of this classification scheme will allow different sets of parameters to be used to classify devices in different categories. In addition, these categories give some idea of the processing different units can perform, although some units may be capable of performing various tasks. For example, to input and store data, unless preprocessing is needed, an input unit can perform the same function as a processor, i.e., the input unit can be used to access a sensor and store the sampled data in memory without interrupting the processor. The hardware can be grouped by category in the database, decreasing the required search time.

For the purposes of this paper, the units considered for the database are either single chips or small boards. The underlying assumption for this scheme is that there is no shared or reconfigurable pipeline units on board. When this assumption becomes false, two $N+1$-tuples will be required to represent shared pipelines and their reconfiguration times.

A functional description such as that found in [ArB82], can be used to accurately categorize each unit according to its functional capabilities. To this point, only processing hardware has been considered. The hardware database can be divided into the functional units of processor, memory, input/output, vector, and array processors. This is consistent with [ArB82]. Each of these functional categories will have a set of tuples used to describe its performance. The tuples will be used with the characteristics of the application algorithm to choose specific hardware for each level of the system.

Included with the hardware descriptions of the processors in the database would be a routine that can simulate the performance of the processor. By combining the simulation procedures with the architectural information of other components in the database, e.g., memories, it is possible to create a simulator for the proposed macro-pipelined architecture. Such a database with simulation routines for each relevant component would be a useful tool for the research community interested in the design of macro-pipelined special purpose systems.

4.3. Response Time -- Its Meanings and Interpretations

The desired response time can be interpreted in various ways depending on the application. For certain applications, the response time may be a function of input. This is discussed in detail in Section 4.5. In such cases, the desired response time can be considered to be an average response time T_{rav} or as an absolute maximum acceptable response time T_{rmax}. Let T_{rdes} be the desired the response time and T_R be the actual response time. If $T_{rdes} = T_{rmax}$ then it is required that $T_R \leq T_{rdes}$. This results in a system that will always respond as fast as or faster than the desired response time and is useful where response time is crucial. Such a system may respond faster than is needed, thus the hardware will not be fully utilized when $T_R < T_{rdes}$.

T_{rdes} can be interpreted to mean T_{rav}. Let D be the number of input data sets to be processed and let T_{R_i} be the actual system response time for the i^th^ data set. If $T_{rdes} = T_{rav}$, then

$$\sum_{i=1}^{D} T_{R_i} = D\, \overline{T}_R$$

The average response time is T_{rdes}, but it is possible for $T_R \geq T_{rdes}$ on multiple consecutive occasions. If the processing times of various data sets are unrelated (independent), the probability that $T_{R_i} > T_{rdes}$ on M consecutive occasions is: 0.5^M. In a real-time environment, if the system falls behind the incoming data, there are two cases that can arise. Either there will not be enough buffering and data will be lost, or there will be enough buffering and results will be delayed. In certain real-time applications, such as air traffic control, neither of these alternatives is desirable, so T_{rmax} should be used instead of T_{rav}.

In addition, it may be necessary to specify both T_{rav} and T_{rmax}. This corresponds to the case where an average response time is desired and where an absolute ceiling on the response time is needed. For the purposes of this paper, it will be assumed that $T_{rdes} = T_{rav}$.

4.4. Parallelism, Task Division, and Design Scenario

It is reasonable to ask: "if given a description of a task, can a computer be designed to execute it?" Since various algorithms that perform a given task require varying types of calculations, memory space, interconnection networks, and execution times. For a simple example of some of the above variations, consider an in-place sort (bubble) [HoS82b], a sort that requires extra memory (bin) [AhH76], and a parallel sorting algorithm ([Pre77]). Assume the sorts are performed on a list containing N_e elements, with the largest element L digits long. The bubble sort will require $O(N_e^2)$ time with no extra memory. The bin sort will require $O(N_eL)$ time in $2N_e$ memory. The fast parallel sorting algorithm proposed in [Pre77] will require $O(\log_2 N_e)$ time using $\lceil N_e \log_2(N_e+1) \rceil /2$ processors. Since the time, memory space, and optimal arrangement of hardware are functions of the algorithm, not the task, it is best to extract the needed features from the algorithm and design an architecture to fit a specific set of algorithms. Thus, to design a system for a given task, it may be necessary to evaluate and compare the use of several different algorithms and their associated hardware requirements.

After the initial layering is performed, an exact statement of the application algorithm to be performed at each level will be used with the hardware description N-tuples to evaluate the performance of each processor in the hardware database. Then information about the desired throughput and average response time ($\overline{T}_{rdes}$) of the system must be gathered. These will be the evaluation criteria, i.e., can a proposed system process the data with the desired throughput and response time.

It is possible that, for each level, the exact algorithm may be available only as a selection of various algorithms, e.g., there may be more than one choice for an algorithm for a level to process. The speed at which a given level operates is a function of the algorithm, so the speed of the corresponding level will be a function of the final algorithm chosen for that level. Since the algorithms determine the layering of a task, not only the architecture of a single level, but the entire system architecture depends on the selection. It is also possible that one selection may require another, e.g., a frequency domain process may require that the data be converted from time domain to frequency domain. Thus, the entire system may be a list of possible alternatives. Since this requires a precise computational model for each level, each possible alternative architecture for the system must be explored.

The first step in the modeling process is to choose all levels to process their incoming data as fast as possible without using vertical or horizontal parallelism within any given level. The resulting design is a macro-pipelined system.

Architectures that are designed along these lines "pipeline" data through the levels, producing a continuous flow of data. The time to process a single data set (the time for data to go from the first level to the last level, i.e., the response time) in such a vertical architecture is not decreased by the parallelism of the multiple levels of hardware. The throughput for multiple data sets is greatly increased because new results are completed at a rate equal to the processing time of the slowest level or sub-level. This is a considerable improvement over a traditional serial design. If the time to go from the first level to the last level is too slow, horizontal parallelism, such as that found in SIMD or MIMD machines, must be applied. The design resulting from the application of the techniques outlined in this scenario, will be neither purely parallel nor purely pipelined, but will be a hybrid combination of both forms of parallelism.

If the processing time for all levels and sub-levels of an architecture were halved, the time to go from the first level to the last level would also be halved. Thus, vertical parallelism can be applied to increase throughput, while horizontal parallelism can be applied to increase throughput and decrease response time.

Horizontal parallelism, however, is not the cure for all slow tasks. The limitation on horizontal parallelism is the inherent parallelism of the subtask to be performed. Further, horizontal parallelism is affected by precedence constraints of the subtask. Vertical parallelism is not affected by precedence constraints because they are still enforced; however, vertical parallelism will not reduce the response time. Thus, there are both associated costs and limitations with both vertical and horizontal parallelism.

The design of a machine suited to a special task can be considered to be a two step process.

1) Create sub-tasks based on conceptual differences (vertical)
2) Break down sub-tasks based on time requirements (horizontal and vertical)

Step 1 creates the initial levels of the hardware. Since the execution time may vary extensively from level to level, and the levels are pipelined, the execution time of each level should be balanced to allow maximum utilization of all hardware present. That is, the overall processing speed of the macro-pipeline will depend on the speed of the slowest level. Step 2 would be employed to increase the throughput and/or reduce the response time of slower levels to help balance the execution times.

The next portion of the design will require an interface between the sub-task and the hardware. Included in this interface is the description of the sub-task in terms that relate it to the requirements that it places on the hardware. This description is used to design candidate architectures, whose performance is evaluated by some measure [SiS82]. It is this portion of the design that is the topic of discussion in the next two sections.

The task division here is similar to those used for Piecewise Data Flow Architectures (PDF) [ReM83] in that a task is divided into basic blocks called sub-tasks. However, instead of scheduling the sub-tasks for execution on a unit, a special unit is designed for each sub-task. The resulting design is more limited in scope than the PDF, but will be better suited for a specific task. By designing each unit with commercially available parts, the overall architecture can be implemented with current technology, like the PDF; however, by designing a special purpose unit, hardware unneeded for the specific task under consideration can be eliminated from the design, reducing the cost of the end result.

In addition, the design resulting from the PDF is composed of several small units, operating in a data flow environment, combined into a larger more powerful "single" unit. A given functional unit in a PDF may be used several times, by various processes in the scenario, while the proposed levels will execute layers on each data set once. Further, the PDF is composed of simple atomic units, while a level in the proposed design is a combination of traditional serial, SIMD, MIMD, and pipelined designs. This allows the level associated with each sub-task to be designed to achieve the desired combination of throughput and response time.

The research in both [ReM83] and [WoC84] relate to work done in [Vic79], in which a distributed computing system is analyzed with respect to implementation of switching, bussing, and interconnection; partitioning criteria; and testing methodology. In [Vic79], a dynamically reconfigurable system similar to a PDF is considered. There, graph theory is used to go from the algorithm to the actual hardware. The pertinence of the research to the work under consideration here is that the analysis of what amounts to a distributed machine has been considered with respect to robustness and throughput.

There are two limitations on the type and amount of parallelism applied at each level. The first is that there must be an upper bound on the cost. An additional limitation is placed on the type and amount of parallelism by requiring that all parts be "off the shelf." This second limitation forces the architecture to be buildable with present day technology. These limitations assume that an algorithm can be structured for parallel execution. If an algorithm is unsuitable for parallel execution then vertical parallelism is required.

The minimum horizontal parallelism at any level is a single unit, while the maximum horizontal parallelism is limited by the inherent parallelism of the sub-task and cost of the units. Typically, each additional processor used for horizontal parallelism may not increase the execution speed by exactly the same amount, i.e., the speedup may not be N_p using N_p processors for any N_p. This is discussed in [Sto73]. As mentioned earlier, the minimum vertical parallelism is one processor and the maximum vertical parallelism is one processor per instruction.

To propose and evaluate candidate architectures for levels, a mapping is required between a layer and its corresponding level. Included in this mapping is the description of the layer in terms that relate it to the computational requirements that it places on the hardware. It is this mapping that is the topic of discussion in the next section. Using information from the hardware database discussed in Section 4.2, the performance of the system can be approximated. Simulation is required to insure that the system will perform as desired.

4.5. Evaluation Categories -- The Relationship Between the Layer and the Level

If hardware is to be designed for a specific algorithm, characteristics of the algorithm must be "mapped" onto the hardware. To build hardware for a given level, a user must supply each of the following **evaluation categories** about each layer in the system.

(1) Type, rate, and amount of inputs
(2) Type and number of operations per input datum
(3) Range and accuracy of arithmetic data to be used
(4) Algorithm to be used
(5) Type, frequency, and message length of processor-to-processor communications
(6) Amount of memory required
(7) Type, amount, and benefit of parallelism
(8) Type, rate, and amount of output
(9) Evaluation criteria

It is the goal of this section to use these parameters to form a model of the algorithms in the task. By using the information supplied by this model with the hardware model of Section 4.2 and the design scenario of Section 4.4 it is also the goal of this section to develop a macro-pipelined architecture well suited for performing the task.

The four factors that can influence the architecture of a specific level are the data, algorithm, performance evaluation criteria, and the input/output environment. (1), (6), and (8) are data related; (2), (3), (4), (5), (6), and (7) are algorithm related; (9) is the evaluation criteria; and (1) and (8) are input/output environment related. Since at any layer or sub-layer, the exact algorithm may be one of a set of candidates, the resulting architecture for a given level and all following levels may also be a list of candidates, with one system architecture per candidate algorithm. For the purposes of discussion, the evaluation criterion will be speed and cost, i.e., the faster an algorithm can be executed, the "better" the hardware design; however, the price of the design should not be excessively expensive. Several other evaluation criteria are considered in [SiS82] and [Gon78].

Initially, information about the desired throughput and response time of the system will have to be known. The first step in the design process will choose all levels to process their incoming data as fast as possible without using vertical or horizontal parallelism within any given level. Since this type of design is a macro-pipeline, the slowest level will limit the throughput of the entire pipeline.

To fully utilize the hardware in the system, it is desirable to match the speed of all the levels. There are two design philosophies that can be employed to balance the throughput of the levels. After the initial design (all levels designed to execute their layer as fast as possible with no vertical or horizontal parallelism), the data processing rate of all the levels will be known. If the designed machine meets or exceeds the throughput and response time qualifications of the scenario, faster levels can be combined or built with slower less expensive hardware. This will still maintain the throughput of the system, while increasing the response time. Such a process can be repeated as long as the throughput/response time requirements are met. This will lower the cost of the overall system.

If the resulting macro-pipelined architecture (i.e., one with no parallelism within a given layer) machine fails to meet the throughput qualification for all processors in the database, the execution speed of all levels not meeting the time constraint must be increased. This can be done with either vertical or horizontal parallelism.

If the machine fails to meet the response time constraints, horizontal parallelism can be employed (vertical parallelism will not improve response time). Let $\overline{T}_{L_i}$ be the mean response time for level i to perform its corresponding layer, $\overline{T}_{rdes}$ be the desired average response time for the system, and N_L be the number of levels. One way to meet the response time constraint is to attempt to force:

$$\overline{T}_{L_i} = \frac{\overline{T}_{rdes}}{N_L} \tag{1}$$

Alternatively, the response time criteria may be met even if the equality outlined in (1) is not true for all levels; however, the sum of the $\overline{T}_{L_i}$'s for all levels must be less than or equal to $\overline{T}_{rdes}$. That is, the requirement is:

$$\sum_{i=1}^{N_L} \overline{T}_{L_i} \leq \overline{T}_{rdes} \tag{2}$$

In general, it is possible for equation (2) to be true without equation (1) being true, and still satisfy the throughput requirements (although the execution times for all levels of the macro-pipelined system will not be balanced and the slowest level will determine the throughput). This implies that the required throughput is less than one job every $(\overline{T}_{rdes}/N_L)$ time units; i.e., the required throughput is less than $(N_L/\overline{T}_{rdes})$ jobs per time unit. For this initial study, however, equation (1) will be used as a guideline for the system design. Thus, for all levels where:

$$\overline{T}_{L_i} > \frac{\overline{T}_{rdes}}{N_L}$$

horizontal parallelism must be introduced to attempt to reduce $\overline{T}_{L_i}$.

For simplicity, some form of coordination can be used between the levels. Chapter 5 will explore the effects of lifting this restriction. For this chapter, however, the coordination can be in the form of either (a) a master system clock that tells each level when it can proceed to the next data set or (b) a unit that keeps track of each level and, when all levels are done, signals each to proceed to the next data set. The differences between these two implementations are that (a) will use less hardware than (b), and that (b) will execute at least as quickly as (a). If T_{L_i} is the time required for level i to complete its subtask, then (a) must be set for the maximum possible value of T_{L_i} over all levels for all data sets. The implementation suggested in (b) will require an execution time T_e of:

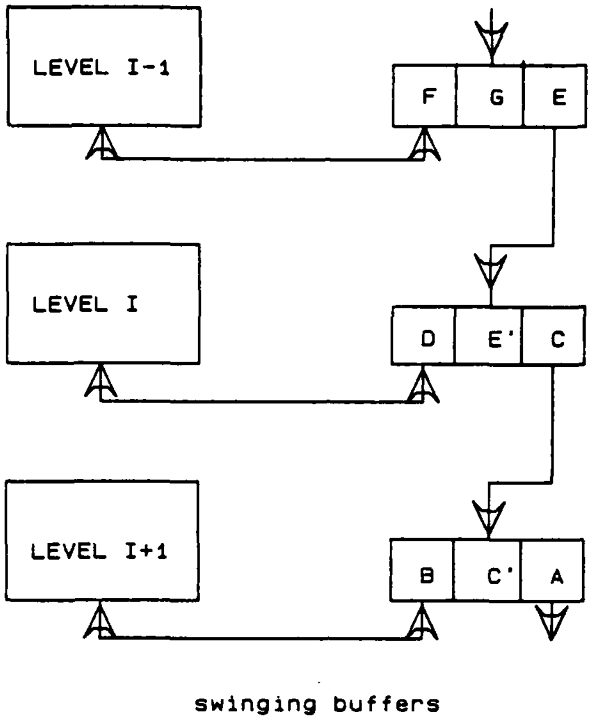
$$T_e = \max(T_{L_1}, T_{L_2}, T_{L_3,} \cdots, T_{L_{N_L}})$$

While in the extreme case, this will be equal to the implementation suggested in (a), normally, T_{L_i} will be less than the T_L for the slowest possible level. The following is an analysis of how each of the categories is derived, and how it affects the architecture of a given level.

Category (1) relates the input characteristics of the system to the I/O environment in which it will execute. It places restrictions on the input buffering, input data rate, and the internal data format of a level. The type of data specifies the format and word width required to process the incoming data. Combined with the rate, the type of data specifies the speed of the input unit. Consider a situation in which 250 32-bit floating point numbers and 1000 8-bit integer numbers must be processed in one second. The types of input data are specified. By combining the type with the rate, 2000 bytes of data per second must be either processed or stored if the unit is to operate without losing data. The input rate is required for the first level only, since all other levels are transferring data by a common clock.

The proposed architecture will overlap data transfer between levels (and sub-levels) with the computation in those units. Consider the example shown in Fig. 4.5.1 (a), where each level is connected to the next level through a buffer. Shown next to each level is its **triple** or **swinging** buffer memory: one unit for data currently being operated upon, one unit for storing data previously generated by that level (and currently being sent to the next level), and one unit to receive data currently being sent by the previous level. This scheme was proposed in [Dem83] and is quite useful towards this application because it allows the overlap of data transmission with the actual data processing, so each level is effectively sending a data set, processing a data set, and receiving a data set simultaneously. It is assumed that the data sets are actually transmitted via some DMA device.

For example, consider the data sets in the figure using level i's swinging buffers. Data set E is being sent by level i-1 (which previously generated it) to level i(which will process it after it finishes processing data set D). Data set D is currently being processed by level i. Data set C is being sent from level i to level i+1 (which will process it after it finishes processing data set B). The transmission of data sets A, C, E, and G, and the processing of data sets F. D. and B, are all occurring simultaneously. The time to perform these



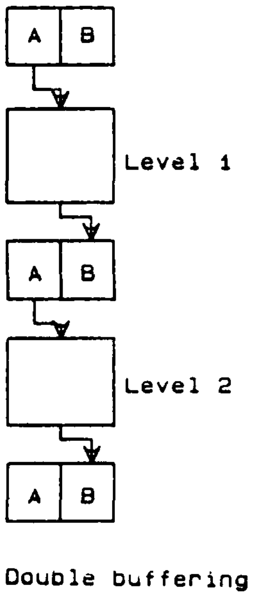


Fig. 4.5.1 Buffering implementations

simultaneous transmissions and computations is called an **interval.** In the next interval, data sets B, D, and F will be transmitted and data sets A, C, E, and G will be processed. Similarly, in the interval prior to the one shown in the figure, data sets A, C, and E were processed and data sets B, D, and F were transmitted. In summary, an interval is the time required for a level to simultaneously receive a data set, transmit a data set, and process a data set, such as level i does with data sets E, C, and D, respectively in Fig. 4.5.1 (a)).

Since temporary results are stored in memory that is local to the processors accessing the buffer, calculation of the size of the buffer between level i and level i+1 is straightforward. If iss_i is the input set size for level i and oss_i is the output set size for level i, the buffer memory required for level i is:

$$\text{memory}_{\text{buffer}} = 3 \times \max(iss_i, oss_i)$$

Under different conditions, such as those discussed in Chapter 5, it is possible that double buffering, as shown in Fig. 4.5.1 (b), can be employed instead of triple buffering. In this scheme, each level processes the data in the "A" portion of its input buffer while writing the results of the calculations in the "B" portion of its output buffer. Then, the levels process the information in the "B" portions of their input buffers and write the results in the "a" portions of their output buffers. For this scheme, the buffer memory becomes:

$$\text{memory}_{\text{buffer}} = 2 \times \max(iss_i, oss_i)$$

The type and amount of operations (2) indicate what must be performed to process incoming data. There are two classes of algorithms that are of concern for this category. There are those that perform the same operations on each data element (data independent) and those that treat each data element differently (data dependent). For data independent algorithms, the number and type of each operation performed is countable from the algorithm. Some examples of algorithms that are data independent are smoothing [SiS83] and maximum likelihood classification [SiS80].

A reasonable indication of the data dependence of an algorithm can be defined by the following test equation:

$$\text{Data Dependency} = \frac{\text{Data Dependent Operations}}{\text{Total Operations}}$$

The smoothing and maximum likelihood classification have Data Dependency 0.

Examples of data dependent algorithms include contour tracing [TuA83], calculation of Fourier descriptors [SiS83], and calculation of center of mass [SiS83]. For a data dependent algorithm, the Data Dependency may not be obtainable from the algorithm alone, as the Data Dependency may, itself, be data dependent. In such cases, the Data Dependency must be determined through simulation on a sample data set. Typically, data dependent algorithms require varying resources and processing times.

The Data Dependency is a valuable measure, as it gives a figure of merit to the number and type of operations performed to process data. In cases where the Data Dependency must be determined by simulation, the average number and type of operations per datum must also be determined. The Data Dependency can be used as an indication of the appropriateness of SIMD or MIMD parallelism.

The class of operations can be divided into five groups: (A) arithmetic, (B) addressing, (C) index calculation, (D) conditional, and (E) data transfer. These classes were chosen to yield information about which unit can process an operation. For example, on some SIMD systems operations in class C can be done in the control unit, overlapped with the parallel execution. The rest of the operations are done by the processing elements. Information about class (E) indicates how much the network will be used. On a system where all processing is done by the same unit, the distinction between the types of operations is diminished; however, for analysis they should prove useful.

Information about the various categories will have to be further subdivided to provide information necessary to choose suitable processing hardware. For example, category (A) should be divided into: floating point additions, subtractions, multiplications, divisions, comparisons, and special functions; and fixed point additions, subtractions, multiplications, divisions, comparisons, and special functions. The usefulness of this list is that it indicates the relative importance of the speed of each operation.

For each floating point or fixed point special function, the number of times each operation is expected to be performed is specified along with an **equivalence relation,** giving the number of "standard" operations needed to implement the specified function in software. If a unit cannot perform a specified function in hardware, the time required to synthesize that function (specified by the equivalence relation) must be calculated. By using an equivalence such as this, various units can be ranked by their execution speed for a given algorithm.

Consider an algorithm that requires only 1000 e^x operations. If a particular unit has the "built in" capability to calculate one e^x, then (using

[Har68]), calculating e^x is worth nine multiplies, nine additions/subtractions, one floor, one square root, and one division. By using this equivalence, if the hardware unit requires 10 msec to calculate each e^x, then any unit not having this special purpose hardware must perform the listed operations 1000 times in ten seconds if they are to be "as fast" as the hardware unit.

The usefulness of this list is that it indicates the relative importance of the speed of each operation. For example, if there is only one floating point divide to be performed on the entirety of a given layer, a hardware floating point divide is likely to have little consequence. It may be necessary, as shown in Section 8, to subdivide the fixed point operations into two categories discriminating between indexing operations and integer calculations. This is required in the event of SIMD parallelism, where the control unit has a different data pathwidth than the processing unit.

Evaluation category (2) helps place a value on T_{L_i} in terms of the actual operations that must be performed. From one data set to another the required processing may vary, so an exact statement of what operations must be performed may be unavailable; however, a reasonable estimate may be calculated for the average case through simulation techniques, as was done with Sobel edge detection in [SiS83]. In this edge detection algorithm, if a pixel is not an edge pixel for an object, it is essentially unused; however, if a pixel is an edge pixel, it is used in the calculation of a chain code that describes the edges of a closed object. This chain code is then used for further processing (in this case, the chain code would be passed on to another level).

Estimates on the calculations to be performed can be used to determine processing speed and special hardware requirements for a given level. Simply, if an algorithm requires large numbers of a given type of operation, the corresponding level should have hardware to perform that operation quickly.

The numerical range and accuracy of a sub-task (3) is a function of algorithm and data. For an algorithm, it is necessary to determine the maximum and minimum values of the range of the calculations. The range of the calculations should be divided according to the range of index values, range of integer arithmetic, and the range of floating point arithmetic. This specifies, in the SIMD case, the word size of the control unit, and the word size of the integer and floating point units. In other cases (SISD/MIMD) [Fly66], the word size of the integer unit is set according to the maximum range required for integer and indexing arithmetic. It is assumed that the floating point and integer hardware can have different widths. An example of this is the PDP-11/70, where a single precision integer is 16-bits and a single precision floating point number is 32-bits.

Category (3) places various limitations on the hardware. Typically, more accurate hardware (larger words) will be slower and more costly. Floating point operations are typically slower than the corresponding integer operations. In certain cases, if the numerical range required for various calculations is small, but out of the range of specific hardware, e.g., underflow, normalization of data can eliminate the need for special hardware at the cost of some processing time. The arithmetic range associated with a set of operations greatly affects the hardware required [SmS81]. If only 8-bit precision is needed, a 32-bit processor, which is typically more expensive, memory intensive, and slower than a corresponding 8-bit processor, will offer no benefits in exchange for the extra word length. Certain processors, such as the Am9511A [Amd82], have varying precision and can be employed in cases where arithmetic ranges vary from loop to loop. Other approaches to dynamic word size machines are presented in [KaK78] and [LiT77].

Determination of the type and amount of processor-to-processor communication (4) for a highly data independent task is straightforward. In the case of non-uniform tasks, the required transfers may vary randomly in size and connection, dependent solely on the data set being processed. In this situation, simulation may be required to achieve accurate estimates for the average case. To minimize the need for simulation, analysis of the data set can yield information about the required connections. For example, if a process is edge tracing small objects relative to the size of the image being processed, global connections are not required, only local (nearest neighbor) connections are needed. If the objects are large, then global connections may be needed.

In addition, with knowledge about the algorithm a level is to process (4), special parallel analysis techniques, such as those discussed in [Ber66], [RaG69], and [KuM72] can be employed to utilize "extra" parallelism. This can be accomplished by breaking the algorithm down into multiple streams, using MIMD parallelism. Applicable loops are those containing variables that can be calculated independently of other variables within the loop. The "breakdown" occurs when a variable can be extracted from a loop and calculated in a separate environment (either a different processor or processors). Other techniques for parallel processing such as the use of "recursive doubling" for calculating sums or maximums [Sto80] can be applied.

The algorithm is required to obtain timing information from the previously discussed tuples. By deriving boundaries on execution time as described in [HuL82], levels requiring large amounts of time can be analyzed. The algorithm must be scanned to determine what operations can be pipelined and/or overlapped. This must be done for each processor in the database. After the amount of time saved by parallelism and pipelines is determined, this time is then subtracted from the execution time for the processor. For systems with reconfigurable pipelines, the reconfiguration times must be multiplied by the number of reconfigurations required by the algorithm. This will give indication as to where each level is spending its execution time.

If consistent variable names are used from layer to layer, similar task decomposition to the above can be applied across levels to allow the combination and/or sub-division of levels as needed. Consider the scenario in Fig. 4.5.2. The three boxes represent levels one, two, and three. If level three calculates a, b, and c independently of the output of level two, and the throughput of level three is too low, the portion of the algorithm calculating a,b, and c can be moved to level 2. If this makes the throughput of level two too low, a separate unit can be employed for the calculations. The result is shown on the right of the figure.

The type, frequency, and message length of the processor-to-processor communications within a layer (5) will dictate the topology of a level and the design of the interconnection network. There are two types of interconnection networks. A global interconnection networks allows a processor to communicate directly with any other processor within a given horizontally parallel structure (e.g., SIMD or MIMD portion of the machine). Typically, a multistage arrangement is used for such a network [Sie85]. The second type of interconnection network is a local interconnection network, which allows a processor to communicate with a specific number of its neighbors (e.g., 4- or 8-nearest neighbors) [SmS81]. In this case, the processors can be viewed as either a one, two, or three dimensional array when determining the connections to be

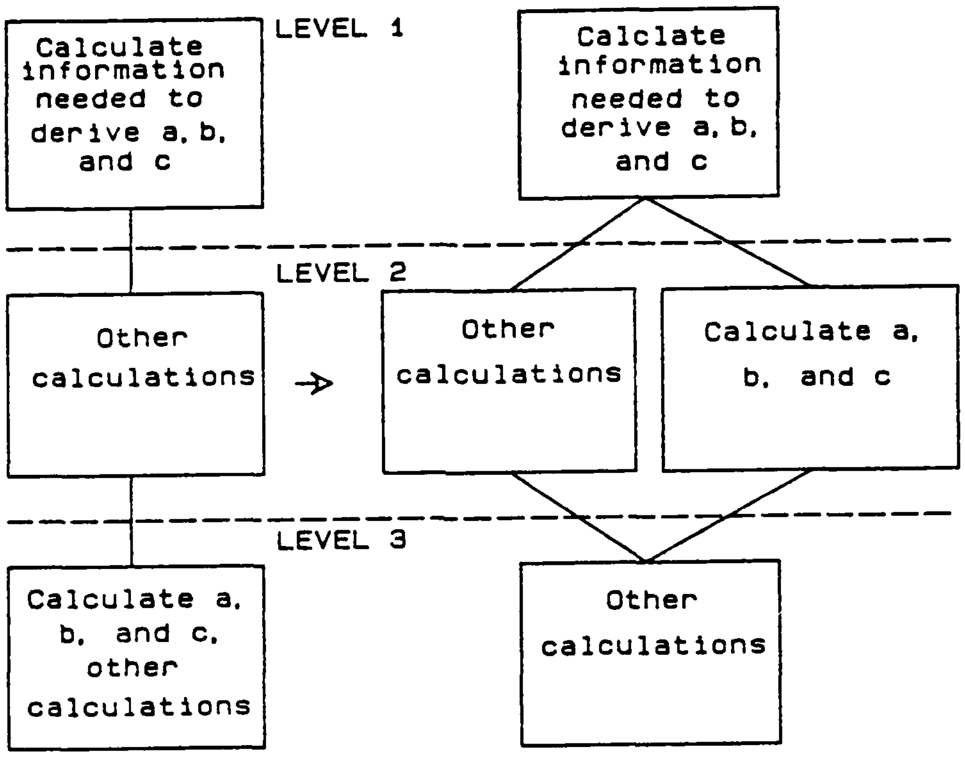

LEVEL 1
Calculate information needed to derive a, b, and c
Calclate information needed to derive a, b, and c
LEVEL 2
Other calculations
Other calculations
Calculate a, b, and c
LEVEL 3
Calculate a, b, and c, other calculations
Other calculations

Fig. 4.5.2 Scenario before and after application of techniques in [HuL82]

made by the network. A network must be capable of making the desired connections efficiently and with a minimal number of collisions, to avoid significant delays during transfers. It would be desirable to have a database of known global connection networks and the permutations that they can perform, so an appropriate connection network can be chosen.

From the type of communications required by a layer, information can be gained about the type of processing that should take place on a given level, i.e., the more random the communications, the more likely that a horizontally parallel level should use MIMD (or asynchronous) parallelism, as opposed to SIMD (synchronous) parallelism. Knowing the size of the transfers will aid the design of the network. For instance, the longer the transfers, the more suitable a circuit switched network becomes. For small transfers, a packet switched network is desirable. Knowing the number of network transfers in conjunction with the size of the average transfer will provide information about the loading of a network with a given transfer speed. Consider an environment where most of a processor's data is stored in memories associated with other processors. In an MIMD environment, a slow network will have collisions within the network in addition to the collisions at the memories. (An ideal network with zero transfer time will only have collisions at the memories.)

The amount of memory (6) is an important factor in the design of a system and is a function of the proposed data set size, data type, and algorithm. Memory usage falls into three classes: program memory, stack memory, and data memory. Program memory (size of the binary) is not determinable from the algorithm. It is a function of the machine and the compiler. The stack memory contains arguments to subroutines, return addresses, and temporary information. Its size is a function of the nesting of subroutines, along with the amount of information that is passed to those subroutines. For data dependent algorithms that use some form of recursion, simulation may be required to determine the appropriate amount of stack memory needed. An alternative to simulation is to place a maximum depth (in terms of calls to specific functions) on the stack. If each specific function is called with a given number of arguments (each with a given size), calculation of the stack size is straightforward.

The data memory size is the sum of the index memory size and the process data size, where the index memory is the memory required to store index variables and loop counters and the data memory is the memory used to store the actual data set and intermediate results. In general, data set size can be calculated from an algorithm.

The particular divisions of memory stem from where the data must be accessed. In an SIMD environment, the stack, index memory, and program memory must be associated with the control unit, while the process data must be accessible by the processing elements. In other environments, this memory is associated with the processor, so the divisions do not matter so much as their total.

The memory size is an important factor in the design of a system. The data set size, number of processors in a level, and algorithm chosen have a very important bearing on how much memory is associated with a processor in a given level. The previously discussed level-level buffering is not considered with this value.

The type of parallelism (7) can be determined by employing a special algorithm for a specific type of machine, or by determining whether an algorithm is best suited for a specific environment. One factor that influences this decision is the Data Dependency, as discussed above. For a general parallel algorithm, the lower the Data Dependency, the more likely an algorithm is suited to SIMD type processing. In SIMD mode, some processors may be disabled while other processors execute portions of an algorithm. MIMD mode does not have this drawback. Instead, MIMD hardware does not typically overlap control unit instructions with processing element instructions. On an MIMD system, it may be necessary to synchronize the processors to insure that certain processing has been done before execution continues.

The amount of parallelism can be determined by several criteria. Typically, the larger the number of processors, the less processing each must perform and the more significant transfer and wait times become. As transfer and wait times become more significant, the processors will spend a larger portion of time idled, so the utilization of a processor will decrease. The question to be answered here is: "At what point is the utilization of a processor more important than raw speed?" If different instructions require approximately the same amounts of time, a reasonable estimate for this figure can be obtained as a ratio of instructions to instructions plus waits. Thus, the utilization can be obtained as a ratio of time spent processing data to the time spent on the entire task (processing and waiting). The Utilization can be defined as:

$$\text{Utilization} = \frac{\text{total processing time}}{\text{total job time}}$$

A variety of performance measures are discussed in [SiS82]. These can be used to determine the relative benefit of each additional processor, allowing one to decide on the number of processors associated with a given level.

Analyzing the algorithm for inherent parallelism with techniques such as those in [Ber66], [RaG69], and [KuM72] provides insight into how additional SIMD or MIMD (horizontal) parallelism can be utilized to increase execution speed. Consider the case of two non-concentric loops which do not require information from each other. If they are processed on two independent machines, as opposed to one machine, the results will be the same, but the execution time will decrease.

The type and amount of parallelism will specify the nature and maximum number of processors associated with a given level. The benefit due to parallelism is specified in two areas: the speedup due to N_p processors and the maximum value of N_p. If speed is the only criterion, then the number of processors associated with a given level could become quite large. Consider smoothing, in which the value for a pixel (picture element) is replaced by the average of itself and the values of its eight nearest neighbors. In a case where transfer time is negligible, the fastest parallel algorithm can smooth a pixel in eight additions and one division. This is the case where each processor is associated with one pixel. For small images this is feasible; however, for larger images the cost due to the large number of processors becomes prohibitive. The cost limitation on a given stage limits the amount of parallelism. This has the side effect of limiting the significance of the network transmission rate (typically, as the number of processors increases, the effect of the network transfer/collision rate becomes more significant).

Knowledge of the type, rate, and amount of output (8) will be required for any formatting that must be done to interface the data to the device gathering the results. In addition, it places constraints on the output data rate.

Finally, the evaluation criteria (9) defined how the merit of a system is to be calculated. By incorporating this into the design procedure, proposed designs not meeting the evaluation criteria can be avoided. In addition, this provides a way to rank various designs.

4.6. An Isolated Word Recognition System -- Task Description

Consider the application of the above theory to an isolated word recognition system. From [Yod82], isolated words are those that are surrounded by distinct pauses. Fig. 4.6.1 [Yod82] is a diagram showing the proposed scenario. The computational portion of this task will be everything past the digitization. To be useful, the resulting design should process the data in real-time. To work with telephone quality speech, the system will have to process 6,670 16-bit words per second. This is the minimum speed limitation on the hardware. In addition, there is a minimum response time. For example, one would not wish to have the delay between an utterance and its recognition of more than a few seconds. Such a system has been discussed in [RaL79].

Conceptually, the task may layered as shown in Fig. 4.6.2. Layer 1 must pre-emphasize the input signal with the following Z-transform:

$$H(z) = 1 - 0.95z^{-1}$$

According to [RaL79], this serves to reduce the variance in later calculations (linear predictive coding). From [Oga70], H(z) translates into:

$$\underline{S}(M) = \underline{S}(M) - 0.95\underline{S}(M-1)$$

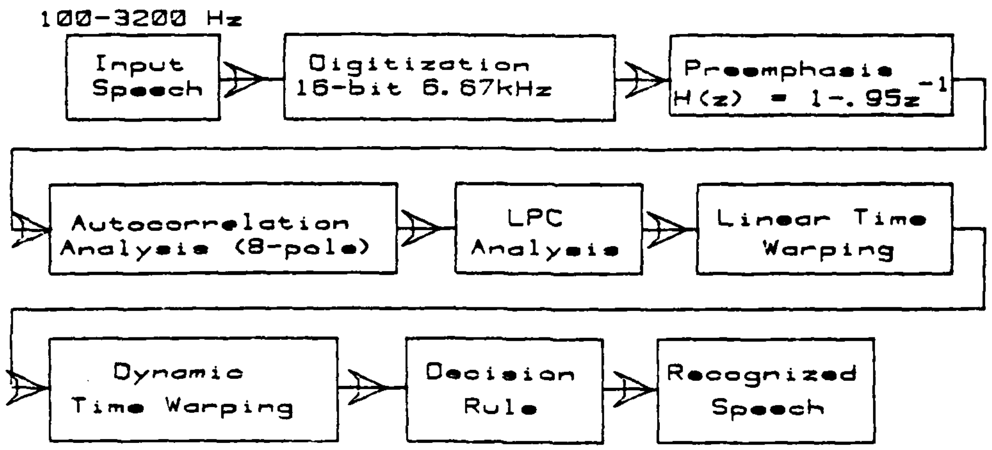


Fig. 4.6.1 Isolated word recognition system

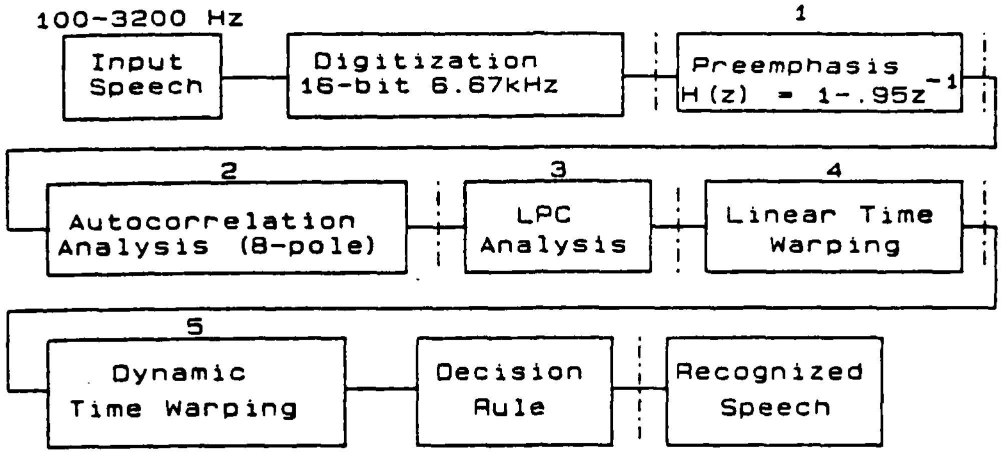
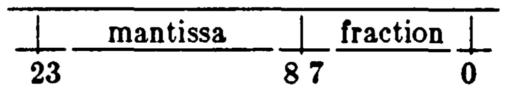


Fig. 4.6.2 Layering of proposed scenario

where S(M) is the M^{th} sample of the incoming signal S.

There has been discussion about what type of arithmetic is required for this process [MaG74]. In [MaG74], it has been shown that the desirable number of total bits in the word is: $\lceil \text{sampling rate in kHz} + 8 \rceil$ Thus, for a 6.67kHz sampling rate, a wordwidth of 15 bits is acceptable. Since the incoming data is 16 bits wide, this represents more accuracy than is needed. To maintain accuracy through the various levels, a word width of at least 24 bits will be considered. This represents an additional 8 bits to minimize error. It should be noted that this is 9 bits larger than the minimum word size suggested in [MaG74] and is included only to minimize any rounding error. Floating point calculations can be avoided by using the following (non-integer) fixed point format:



Thus, $\underline{S}$ may be obtained with one integer fetch, one fixed point multiplication, one fixed point subtraction, one fixed point addition, one fixed point store, and two fixed point register-to-register transfers. In order to keep up with incoming data, the level performing this calculation must perform

6670 16-bit integer fetches

6670 16-bit integer-to-floating point conversions

6670 fixed point multiplications

6670 fixed point subtractions

6670 fixed point stores

13340 fixed point register-to-register moves

every second. The fixed point operations are distinguished from the integer operations only by the 24-bit word width.

This particular scenario performs its analysis by using the autocorrelation method of finding the linear predictive coding (LPC) analysis. The underlying assumption of the autocorrelation method is that S(0) and S(M-1) are both 0 for a window containing M samples. To make this condition true, a Hamming window is applied to $\underline{S}$. The resultant equation is:

$$\underline{s}(m) = \underline{S}(m) \times W(m) \qquad 0 \leq m < M$$

where W(m) [Yod82] is defined by:

$$W(m) = 0.54 - 0.46\cos\left[\frac{2\pi m}{M-1}\right]$$

and M is the number of samples per frame. The frame length is fixed and contains from 100 to 400 samples. Note that now calculations are in terms of frames as a basic unit, as opposed to one data element. A typical method uses 300 sample frames that begin every 100 samples. For the purposes of calculation, W(m) only needs to be calculated once and loaded with the program as a set of constants. Thus, for each 300 sample frame, this portion of the task will require 300 fixed point fetches, multiplications, and stores. In addition, windowing will require one integer load and 299 integer additions. These operations must be performed every 14.9 msec because the frames begin every 100 samples.

After the windowing has been performed, the autocorrelation coefficients are calculated using the following equation:

$$R(i) = \sum_{m=0}^{M-i-1} s(m)s(m+i) \qquad 0 \leq i \leq p$$

Normally p is between 6 and 25. For the purposes of this paper, p is 8. Since there are 300 multiplications and 299 additions that must be performed for every frame, the incoming data must be converted into 32-bit fixed point representation. This can be accomplished by either zero filling or sign extension of the product terms. Thus, this layer will require 2764 fixed point additions and 2773 fixed point multiplications every 14.9 milliseconds. From this point, the data is passed onto the next layer, where LPC analysis is performed.

The goal of LPC analysis is to reduce the number of parameters that are required to represent the speech frame. LPC analysis assumes that each sample is a linear combination of the p previous samples and an excitation. By assuming that the speech is 0 outside the present frame, i.e., $s(m)=0$ for $m<0$ and $m \geq M$, the LPC analysis can be broken down to the following equation [RaS78]:

$$\sum_{k=1}^{p} a(k)R(|i-k|) = R(i) \qquad 1 \leq i \leq p$$

Since p=8, the above equation translates to:

$$
\begin{aligned}
R(1) &= a(1)R(0) + a(2)R(1) + a(3)R(2) + \ldots + a(7)R(6) + a(8)R(7)\\
R(2) &= a(1)R(1) + a(2)R(0) + a(3)R(1) + \ldots + a(7)R(5) + a(8)R(6)\\
R(3) &= a(1)R(2) + a(2)R(1) + a(3)R(0) + \ldots + a(7)R(4) + a(8)R(5)\\
R(4) &= a(1)R(3) + a(2)R(2) + a(3)R(1) + \ldots + a(7)R(3) + a(8)R(4)\\
R(5) &= a(1)R(4) + a(2)R(3) + a(3)R(2) + \ldots + a(7)R(2) + a(8)R(3)\\
R(6) &= a(1)R(5) + a(2)R(4) + a(3)R(3) + \ldots + a(7)R(1) + a(8)R(2)\\
R(7) &= a(1)R(6) + a(2)R(5) + a(3)R(4) + \ldots + a(7)R(0) + a(8)R(1)\\
R(8) &= a(1)R(7) + a(2)R(6) + a(3)R(5) + \ldots + a(7)R(1) + a(8)R(0)
\end{aligned}
$$

This is equivalent to: $\vec{R} = \underline{R}\vec{a}$ where $\underline{R}$ is an 8-by-8 Toeplitz matrix (symmetric with only one unique value along the diagonal) and $\vec{a}$ and $\vec{R}$ are 8-by-1 vectors. Having $\vec{R}$ and $\underline{R}$, the goal is to solve the above equation for $\vec{a}$, which can be done by calculating $\underline{R}^{-1}$. There are algorithms that can utilize the special properties of $\underline{R}$ to calculate $\underline{R}^{-1}$ in fewer than the $O(p^3)$ ($\leq k_1p^3$) operations normally required. One such method is Durbin's Algorithm, shown in Fig. 4.6.3, which calculates the a's, as opposed to $\underline{R}^{-1}$. The computational requirements of Durbin's Algorithm for the calculation of the LPC coefficients for an 8-pole autocorrelation method analysis are:

18 integer initializations,
376 integer/index additions/subtracts,
72 integer comparisons,
8 floating point initializations,
89 fixed point assignments,
28 fixed point additions,
44 fixed point subtractions,
72 fixed point multiplications,
8 fixed point divisions,
225 fixed point fetches,
64 fixed point fetches * ,
64 fixed point stores * ,
128 integer/index operations

every 14.9 msec. (Items marked with an asterisk are required for formatting $\underline{R}$.) After this point, only the a's and R(0) are passed on to the next layer.

```
$E^{(0)} = R(0);$
FOR $i \leftarrow 1$ TO $p$ DO
                                        /* compute k(i) */
    $k(i) \leftarrow 0;$
    FOR $j \leftarrow 1$ TO $i$-$1$ DO
[8]     $k(i) \leftarrow k(i) + a_j^{(i-1)} * R(i-j);$
    $k(i) \leftarrow [R(i) - k(i)] / E^{(i-1)};$
    $E^{(i)} \leftarrow (1-k(i)^2) * E^{(i-1)};$
                                        /* compute $a_j$'s for stage i */
    $a_i^{(i)} \leftarrow k(i);$
    FOR $j \leftarrow 1$ TO $i$-$1$ DO
[11]    $a_j^{(i)} \leftarrow a_j^{(i-1)} - k(i) * a_{i-j}^{(i-1)};$
FOR $j \leftarrow 1$ TO $p$ DO $a_j \leftarrow a_j^{(p)};$
```

Fig. 4.6.3 Durbin's algorithm to compute LPC coefficients a_i from autocorrelation coefficients [Yod82]

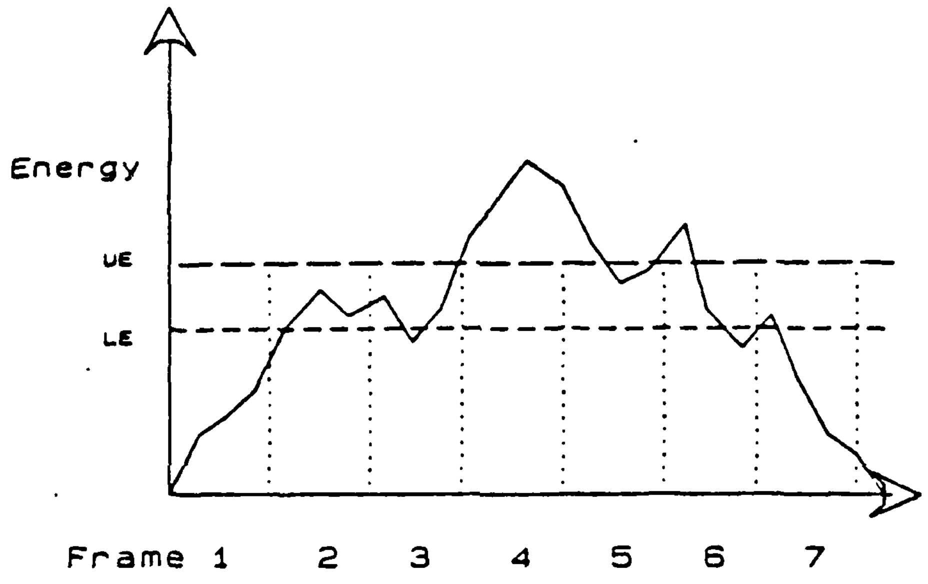


Fig. 4.6.4 Energy of typical utterance

At this point, the beginning and ending points of the word must be found. This will be used to "warp" incoming words, making them the same length (linear time warping). In [RaS78], there is a simple method for determination of the end points in an utterance. This method makes use of the energy ($= R(0)$) of each frame and the number of times the normalized signal changes sign in one frame (zero crossing). It has been shown, however, that the number of zero crossings in telephone quality speech is not effective in detecting word boundaries [LaR81].

Since each word is surrounded by silence, there must be some perturbation to indicate that a word is present. There are only two types of speech, voiced and unvoiced. From [SaR75], a voiced sound will result in a large energy, while other sounds will result in only moderate energy. Thus, setting a lower and an upper bound on the energy will allow one to determine the starting and ending points of a word. The two energies allow the determination of the existence of a word without losing valuable information contained between the lower energy and upper energy thresholds.

Consider Fig. 4.6.4. From frame two to frame six, the energy exceeds the lower energy bound, indicating that the sound is voiced. After the sixth frame, the energy is below the minimum threshold values, indicating that no speech is present. For the purposes of analysis, two constants are defined. They are the upper and lower energy (UE and LE respectively) and are defined as follows:

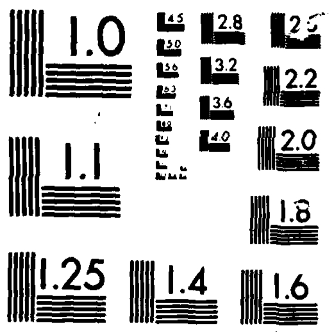
$$LE = MIN(0.03 \times (PEAK-SILENT) + SILENT \text{ , } 4 \times SILENT)$$

$$UE = 5 \times LE$$

PEAK and SILENT are the largest energy over all frames and largest energy

AD-A167 622 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN B W SMITH MAY 85 ARO-18790.17-EL-APP-G DAAG29-82-K-0101 3/4

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

over the ten silent frames respectively. LE and UE can be predetermined to reduce the calculational load. The endpoint detection algorithm is as follows (the value of the frame pointer after the application of this algorithm to Fig. 4.6.4 is shown in braces):

(1) Measure energy for every frame (=R(0)).
(2) Set a pointer to the first frame that exceeds the upper energy threshold {4}.
(3) Back the pointer up until it points to a frame that does not exceed the minimum energy threshold {1}.
(4) Advance the pointer one frame {2}.

The endpoint is located by applying the same procedure in reverse. This process requires 300 fixed point comparisons and a variable number of integer operations depending on word length. In addition, there is a variable number of floating point operations required for each frame as the frame pointer is backed up.

After the beginning and ending points of the word are known, a procedure called linear time warping is applied to make all words the same length. In this procedure, a speech segment containing M frames of data is compressed or expanded to contain F frames of data. This is done by applying the following equation:

$$T(f) = (1-k)\times R(m) + k\times R(m+1) \qquad f=1,\ldots,F$$

where $R(m)$ $1\leq m\leq M$ are the M frames of the input template, $T(f)$ $1\leq f\leq F$ are the F frames of the output template, and:

$$M = \left\lfloor (f-1)\frac{(M-1)}{(F-1)} + 1 \right\rfloor$$

$$k = (f-1)\frac{(M-1)}{(F-1)} + (1-m)$$

k is calculated F times and requires one fixed point multiplication, one fixed point division, one fixed point addition, and four fixed point subtractions. M is calculated once per word and requires one fixed point floor, three fixed point subtractions, one fixed point multiplication, one fixed point division, and one fixed point addition. T(f) is calculated F times and requires two fixed point multiplications, one fixed point addition, one fixed point subtraction, and three integer additions for address calculations. In this paper, the value used for F *will be 40 (frames/word). This represents 596 msec. of speech.*

From here, the data is passed to the next stage, where a process called dynamic time warping (DTW) is applied to each utterance. For speech recognition systems, reference patterns (or templates) for each word the system is to understand are stored in memory. DTW attempts to normalize time in order to make an unknown utterance match each of the template utterances, thus finding the minimum time-normalized distance between the unknown template and the reference templates. Fig. 4.6.5 [Yod82] shows the results of dynamic time warping. In this case, each template is represented by a sequence of feature vectors. Each feature vector contains the LPC coefficients. Since linear time warping has been applied to the two utterances, they are the same length. This reduces the complexity of DTW. The following algorithm [YoS82], considers two patterns A and B, where A and B are sequences of feature vectors a_i and b_j for $1 \leq i \leq I$ and $1 \leq j \leq J$. The a_i and b_j are vectors containing

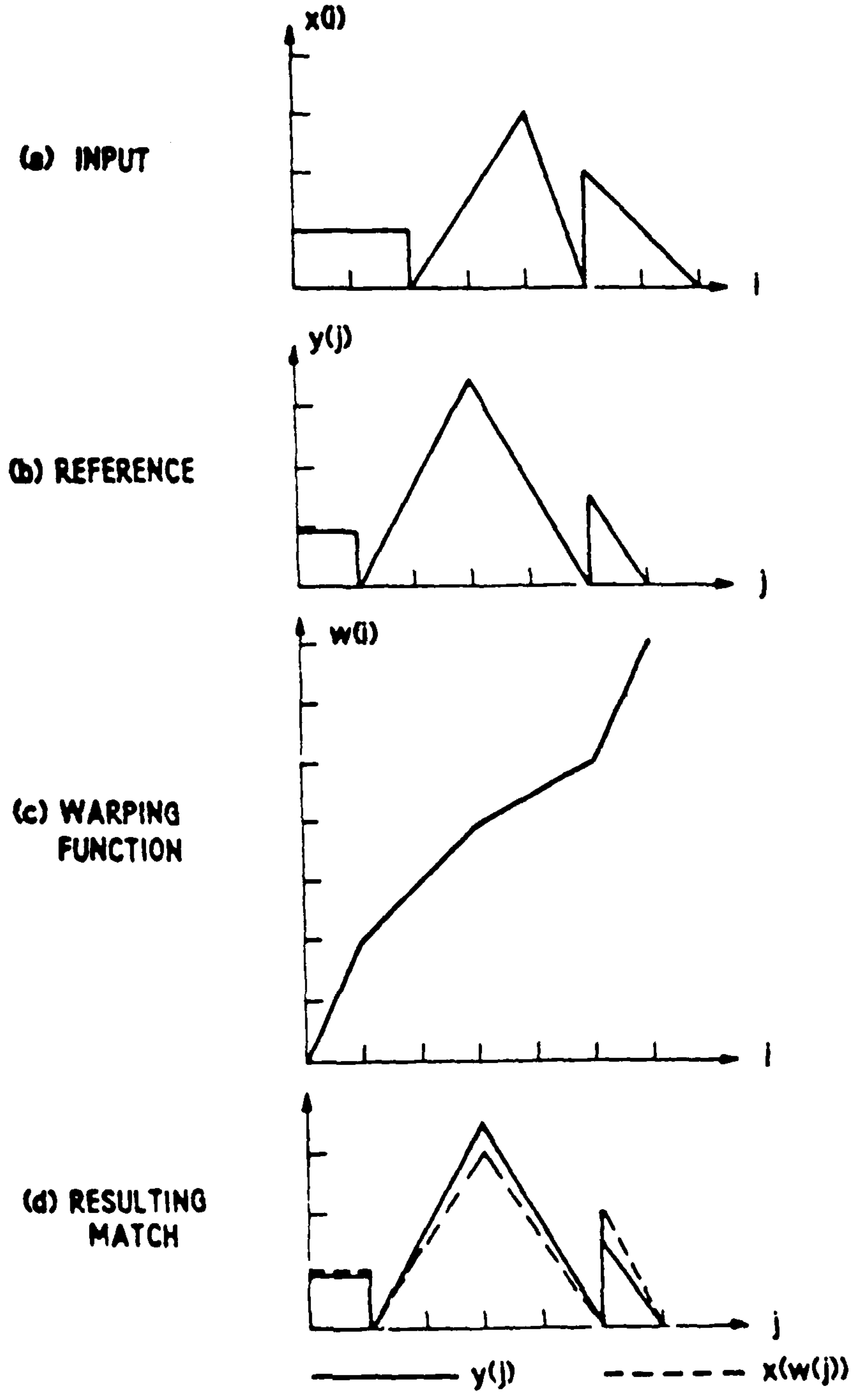


Fig. 4.6.5 Example of time warping [Yod82]

LPC coefficients. Since linear time warping has been performed at the previous level, I and J, the number of frames describing the incoming utterance and the known word template, are equal. The minimum time-normalized distance is found as shown in Fig. 4.6.6. This is accomplished by finding a path connecting (1,1) to (I,J) such that the cumulative distance is a weighted sum of the local distances d(i,j) between the vectors a_i and b_j. d(i,j) is defined to be:

$$d(i,j) = |a_i^2 - b_j^2|$$

One method to find the cumulative distance, g(i,j), restricts the possible path leading to a given point to those shown in the inset in Fig. 4.6.6. Using a recursive definition, g(i,j) can be defined as follows [YoS82]:

$$g(i,j) = d(i,j) + \mathrm{MIN}\begin{bmatrix} g(i-1,j-2)+2d(i,j-1) \\ g(i-1,j-1)+d(i,j) \\ g(i-2,j-1)+2d(i-1,j) \end{bmatrix}$$

$$g(1,1)=2d(1,1)$$

The result of the algorithm is the time normalized distance, g(I,J)/(I+J). Fig. 4.6.7 shows a serial DTW algorithm. For the purposes of this paper, let "r", the amount the algorithm is allowed to "warp" the utterance, be 3 and J, the number of templates of known words, be 10000. This will give the system a vocabulary of 1000 words (since speaker independent word recognition requires ten templates for each word in the vocabulary [Ral79]), at a cost of over 10^8 index operations. By choosing the word corresponding to the minimal distance,

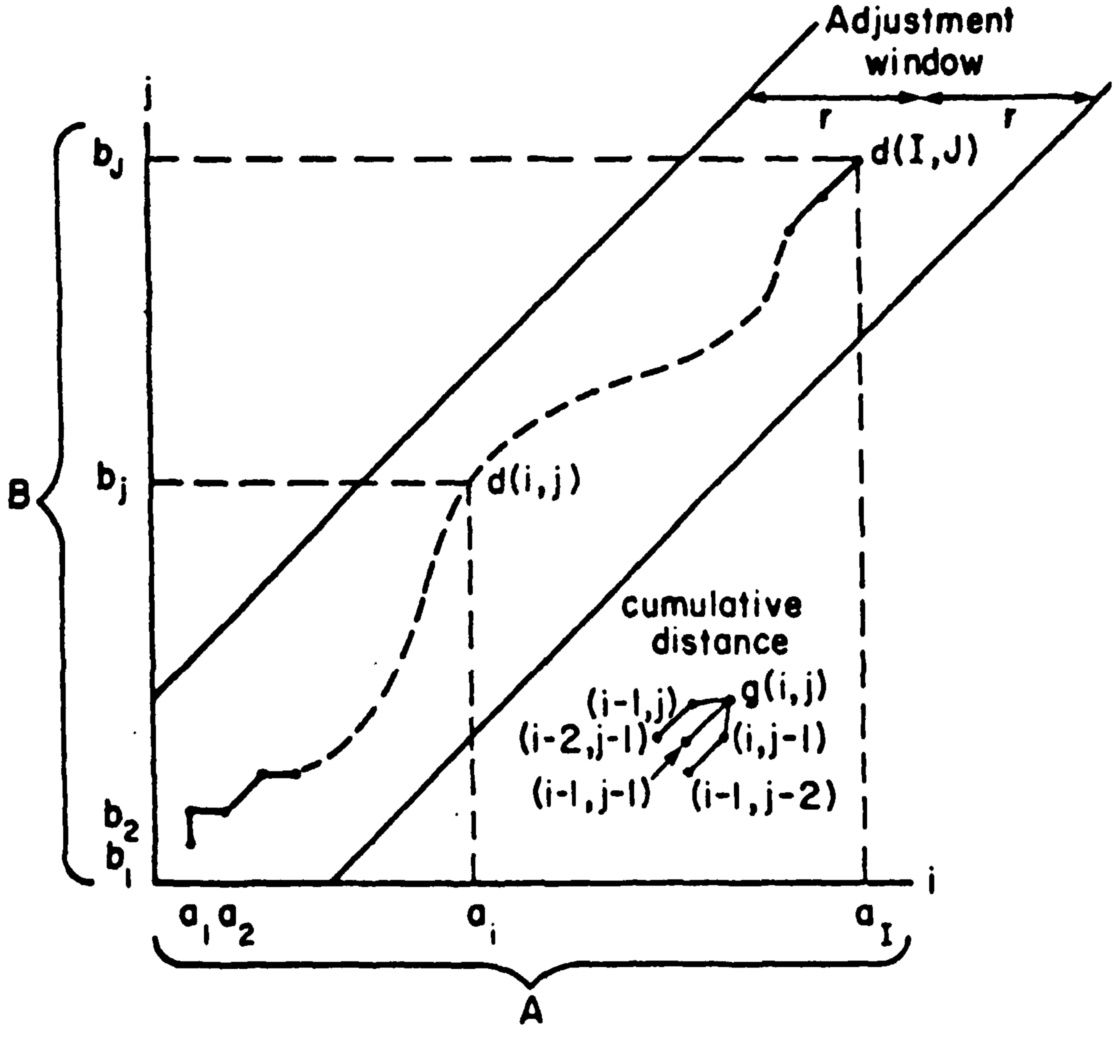


Fig. 4.6.6 Adjustment window of width r [Yod82]

```
hold = ∞ ; template =0;/* initialization */
for k = 1 to 10000 {  /* for each template */
    for j = -1 to 1 { /* initialization */
        for i = -1 to 1 {
            g[i][j]=∞;
            d[i][j]=∞;
        } /* end i */
    } /* end j */
    for j = 1 to 80 { /* for each frame in a and b[k] */
        for i = j-r to j+r { /* each frame within window */
            if(i≤0) i = 1; /* force i to be valid */
            if(i>80) i = j-r+1;
            else {
                d[i][j]=0;
                for h = 1 to 9 {
                    /* compute 'distance' between
                      frames a[i] and b[k][j] */
                    d[i][j]= d[i][j]+
                        (a[i][h]-b[k][j][h])**2;
                } /* end h */
                g[i][j]=min(g[i-1][j-1]+2d[i][j],
                    g[i-1][j-2]+2d[i][j-1]+d[i][j],
                    g[i-2][j-1]+2d[i-1][j]+d[i][j]);
            } /* end i */
        } /* end j */
    D(a,b[k]) = g[80][80];
    if(D(a,b[k]) < hold) { /* store minimum value */
        hold = D(a,b[k]);
        template = k;
    } /* end if */
} /* end k */
```

a	- unknown word (UW)
a[i]	- frame i of UW
a[i][h]	- element h of vector describing frame i of UW
b[k]	- reference word k (RWK)
b[k][i]	- frame i of RWK
b[k][i][h]	- element h of vector describing frame i of RWK
D(a,b[k])	- distance between UW and RWK
g(i,j)	- cumulative distance between a and b[k]
hold	- distance number of best fitting reference word
template	- number of best fitting reference word

Fig. 4.6.7 Sample DTW algorithm

this system will have speaker independent accuracies up to 98% [RaL79].

4.7. Application of Theory to Scenario

The application of the evaluation categories to the Z-transform gives the following information:

1) Input:	16-bit integer/sample, 6670 samples/second
2) Calculations:	1 16-bit fetch, 1 24-bit fixed point multiplication, 1 24-bit fixed point subtraction, 1 24-bit fixed point store, 2 24-bit fixed point register-register transfers
3) Range/Accuracy:	+65535 to −63000/1
4) Communications:	None
5) Memory:	Program + Data + Stack $<$ 2 Kbytes
6) Parallelism:	MIMD
7) Algorithm:	Stated above
8) Output:	6670 32-bit floating point numbers per second

After the Z-transform is performed by the first level, the data is passed onto the next level where the windowing is performed. The computational requirements of this level are as follows:

1) Input:	6670 24-bit fixed point numbers/second
2) Calculations:	3 24-bit fixed point multiplication per number, 6 24-bit fixed point fetches/stores per number
3) Range/Accuracy:	±65535/.008
4) Communications:	Nearest Neighbor 200 24-bit fixed point numbers
5) Memory:	Program + Data + Stack $<$ 5 Kbytes
6) Parallelism:	SIMD/MIMD
7) Algorithm:	Stated above

8) Output:	300 16-bit fixed point numbers/14.9 msec.

Note that this level performs the operation of dividing the data into 300 sample frames that are transmitted every 100 samples (14.9 msec). Every 14.9 msec, the 300 sample frames are transmitted to the next level, where autocorrelation analysis is performed.

The calculational requirements of the autocorrelation analysis are as follows:

1) Input:	300 24-bit fixed point numbers/ 14.9 milliseconds, 300 32-bit fixed point numbers=1 frame
2) Calculations:	2773 32-bit fixed point multiplication/frame, 2764 32-bit fixed point additions/frame, 2782 32-bit fixed point fetches(stores)/frame, 2782 integer additions/ frame (indexing)
3) Range/Accuracy:	$\pm 2^{24}/1$
4) Communications:	None
5) Memory:	Program + Data + Stack $<$ 7 Kbytes
6) Parallelism:	MIMD
7) Algorithm:	Stated above
8) Output:	9 32-bit fixed point numbers/14.9 msec.

The results of the autocorrelation analysis are used for the LPC analysis, where the amount of data is reduced from 300 16-bit fixed point numbers representing a frame to 9 32-bit fixed point numbers. The requirements of the LPC analysis are:

1) Input:	9 32-bit fixed point numbers per frame
2) Calculations:	18 integer initializations, 504 integer/index additions/subtractions, 72 integer comparisons, 8 32-bit fixed point initializations, 89 32-bit fixed point assignments/stores,

	28 32-bit fixed point additions, 44 32-bit fixed point subtractions, 72 32-bit fixed point multiplications, 8 32-bit fixed point divisions, 553 32-bit fixed point fetches
3) Range/Accuracy:	$\pm 2^{24}/\ 0.008$
4) Communications:	None
5) Memory:	Program + Data + Stack $<$ 7 Kbytes
6) Parallelism:	MIMD
7) Algorithm:	Stated above
8) Output:	8 32-bit fixed point numbers/14.9 msec.

After the LPC analysis is performed, the a(i)'s are then passed onto the next level where endpoint detection is performed. Endpoint detection takes place over several frames. The time required for the endpoint detection algorithm is a function of the number of frames in a word. On a per word basis (assuming 80 frames per word (I=80 and J=80), there the calculational requirements of the endpoint detection algorithm are:

1) Input:	720 32-bit fixed point numbers/1.2sec.
2) Calculations:	1520 32-bit fixed point comparisons/word, 324 integer increments(decrements)/word, 4 integer stores, 1280 32-bit fixed point fetches(stores)
3) Range/Accuracy:	$\pm 2^{24}/.008$
4) Communications:	none
5) Memory:	Program + Data + Stack $<$ 10 Kbytes
6) Parallelism:	SIMD/MIMD
7) Algorithm:	Stated above
8) Output:	640 32-bit fixed point integers/word.

These figures represent maximums because it is quite possible to go for many frames without receiving any input. In such a case, it is possible to just require the comparisons with only a memory update to accommodate the incoming frames.

From here the words are passed on to the level that performs the linear time warping. Assuming 80-frame words (which are very long), the calculations required for linear time warping are as follows:

1) Input:	720 32-bit fixed point numbers/1.2 sec.
2) Calculations:	1480 32-bit fixed point fetches(stores)/1.2 sec. 480 32-bit fixed point additions/1.2 sec. 162 32-bit fixed point subtractions/1.2 sec. 800 32-bit fixed point multiplications/1.2 sec. 40 32-bit fixed point divisions/1.2 sec. 40 32-bit fixed point floor operations/1.2 sec.
3) Range/Accuracy:	$\pm 2^{24}/.008$
4) Communications:	Global
5) Memory:	Program + Data + Stack $<$ 5 Kbytes
6) Parallelism:	SIMD/MIMD
7) Algorithm:	Stated above
8) Output:	720 32-bit fixed point integers per word.

After the linear time warping is performed, the data is passed on to the next stage, where dynamic time warping is performed (once for each utterance in the vocabulary).

1) Input:	720 32-bit fixed point numbers/1.2 sec.
2) Calculations:	6.8M index variable assignments/1.0 seconds 0.1M index variable additions 66.1M index variable additions (+1) 67.3M index variable conditional branches 132.7M address calculations 105.5M fixed point additions 5.8M fixed point assignments 11.3M fixed point conditional branches 60.7M fixed point multiplications 60.7M fixed point subtractions
3) Range/Accuracy:	$\pm 2^{24}$ / ± 1
4) Communications:	Global: capable of recursive doubling [Sto79]
5) Memory:	Program +Stack $<$ 10 Kbytes Data (14.5/N)+0.01 Mbytes per processor for reference (template) and incoming utterance storage. Note: One copy of the program is required per

	processor for an MIMD machine; one copy is required for the control unit in an SIMD machine.
6) Parallelism:	MIMD

$$\text{Speedup} = \frac{T}{\frac{T}{N} + \lceil(\log_2 N)\rceil(IC + 2\times NO)}$$

where a single processor takes time T, IC is the time for an integer comparison, and NO is the time for a network operation.

7) Algorithm:	Stated above.
8) Output:	1 45-character word

Consider the derivation of the last set of nine evaluation categories. These nine evaluation categories represent an analysis of the algorithm. Evaluation category 2 is directly determinable from the algorithm. The range and accuracy is determinable from the application. [RaL79] states that 15 bits is a reasonable wordwidth when the sampling rate is 6.67 KHz, as it is for this task. To apply a parallel machine to this algorithm, each PE would need to execute this algorithm on its own portion of the tamplate database computing a local D(A,B). Recursive doubling [Sto79] would then be used to combine the results; i.e., the word associated with the smallest D(A,B) is the chosen word. This requires 2 $\log_2 N$ transfers for the D(A,B)'s and the identifiers for their associated words.

The amount of memory is expressed as a function of N, the number of processors. A "C" language program was coded and compiled to estimate the program size. The DD is small, so either SIMD or MIMD parallelism can be applied to the program; however, the maximum parallelism is 10,000 processors, assuming each PE executes the algorithm for one or more templates. Application of N processors will yield the speedup shown in (6). The output of this system is one word. It is imperative that the system keep up with the input; however, it is desirable to do such with a minimal cost.

The number of each calculation can be multiplied by the single-operand execution times of the tuples for each processor in the database. The sum of the products yields an approximate worst-case execution time for a single copy of each processor in the database to perform this algorithm. Actual execution time could be better due to clever software or special hardware functions. For example, software that is written to ignore redundant calculations. Also, by applying pipeline analysis techniques to this algorithm and using structural information about each processor, such as functional overlap, stages in processing pipelines, and the multiplicity of units, a more precise approximation of the single processor execution times can be obtained.

Based on the desired response time, additional processors of the same type are repetitively added until a level composed of such processors could meet the time requirements. The number of processors is then multiplied by the cost of the associated hardware. To this amount, the price of other devices, such as memory and inter-processor communications links, is added to approximate the cost of the processing hardware involved. The processor chosen used for the design will be chosen based on the least expensive hardware.

Consider the application of a Motorola 68000 to the above task. The tuples enumerating the operations and their respective times contains over 1000 instructions: a partial list is included for brevity:

{add r,#:add r1,r2:add (a)+,r:cond. branch: mov r,#:mov r,(a):mov #,(a):mul r1,r2; mul (a)+,r; sub r,#;sub r1,r2;sub (a)+,r}

where r stands for register, # stands for immediate, (a) stands for memory location stored in register "a", (a)+ stands for memory location stored in register "a" followed by incrementing "a".

The tuple describing the timings (in cycles) is:

{8,4,8,10(true)/8(false),8,12,12,70,74,8,4,8}

The 68000 has a no functional overlap or pipelining other than a five stage instruction decoder. These tuples will be omitted. A 68000 has no special address calculation hardware, so an address calculation required loading a register, multiplying by a memory location, and the addition of two memory locations. Assuming that the index variables are stored in registers and that fixed point numbers are stored in memory, a 12.5 MHz 68000 would take 1579 seconds to perform dynamic time warping on a single word. Using a multistage cube network that takes 1.0 msec for two transfers, 1600 processors in MIMD mode would take .998 seconds to perform dynamic time warping. (A thorough analysis should consider the overlap of CU and PE operations in SIMD mode; e.g., address calculations). Dynamic time warping is normally done with fewer than 100 reference templates because of its great computational complexity.

Such an analysis would be requires for each processor in the database. Then, an actual implementation of the above approach would consider simulating the algorithm on the various processors to obtain a more accurate timing estimation. Finally, if no processor in the database could be used to implement this algorithm, the layer would need to be broken down into sub layers, each of which would be analyzed with the proposed techniques.

4.8. Conclusions

Using the above nine categories, an algorithm can be analyzed according to the requirements it places on a system. By building hardware to efficiently handle these needs, it will be able to effectively process the algorithm. If many hardware components are analyzed and categorized according to their abilities and processing times, a database containing information about each processor can be built. By mapping the organization of each level in a multi-level design, computers can be used to design systems for specific needs of algorithms. Thus automated design of special purpose processors can be achieved.

In summary, this was a preliminary study of how to partially automate and model the design of special purpose systems. Categories of hardware analysis were presented. Their relationship to the hardware and their dependence on the software was discussed. An application of the theory to a software scenario performing speaker independent isolated word recognition was presented. Finally, the computational requirements of the scenario were presented. By bridging the gap between hardware and software, automated special purpose machine design comes closer to being a reality.

CHAPTER 5 ASYNCHRONOUS AND SYNCHRONOUS SYSTEMS ADVANTAGES AND DISADVANTAGES

5.1. Introduction

Chapter 4 introduced a scheme for modeling the hardware requirements of a layer. It also proposed a concise scheme for modeling the capabilities of a computational device. Finally, it showed a method of going from the hardware requirements of a task to the computational device. This was all done with a real-time system in mind. The type of system considered in Chapter 4 was a synchronous macro-pipelined system with potential parallelism at each level. This chapter looks at the performance of an asynchronous macro-pipelined system, a synchronous macro-pipelined system with triple-buffers between levels, and a synchronous macro-pipelined system with double-buffers between levels. It is the goal of this chapter to show the strong and weak points of each of these schemes, along with showing in which situations each of these schemes is most applicable.

Before considering the use of asynchronous stages in the proposed macro-pipelined architectures, analysis techniques to determine inter-level buffer size, expected process wait time, and the likelihood of buffer overflow, are required. If analysis techniques cannot be developed, the use of asynchronous stages will be complicated because no analysis techniques short of simulation will provide meaningful results. It is the goal of this chapter to determine whether it is possible to derive the following parameters about a (possibly parallel) level:

I **Probability of receiving v jobs by some time t**

($P_v(t)$)

II The expected average input queue length ($\overline{\mathbf{Q}}$) for a given layer

$P_v(t)$ is useful in determining the most likely time at which v jobs will arrive at a given level, which is receiving input (which may include feedback from later levels) from r sources. Integrating $tP_v(t)$ with respect to time will yield the expected time at which v arrivals will occur. Within some tolerance (e.g., $\pm$ a standard deviation), this is required to determine the required throughput of a given level. $\overline{Q}$ is required to determine the time that a job spends "waiting" to get processed. This time must be added to the total computation time for a job to determine the total time required to complete a job.

In the previous discussion, each level was allowed to process only one data set at a time. This was a restriction imposed by the synchronous nature of the system. When the levels of a system are asynchronous, this restriction could be removed. For example, a level could contain multiple processors, each working on a different data set. For the purposes of this study, however, this restriction will still be imposed. As shown in Fig. 5.1.1 (a), it will be assumed that all processors in a given level work together to process a single data set. There is a single input queue for each level. This form of replication corresponds to either the SIMD or MIMD parallelism discussed in the previous section. Only a single result is completed at a time by a level. Outgoing jobs are queued (if

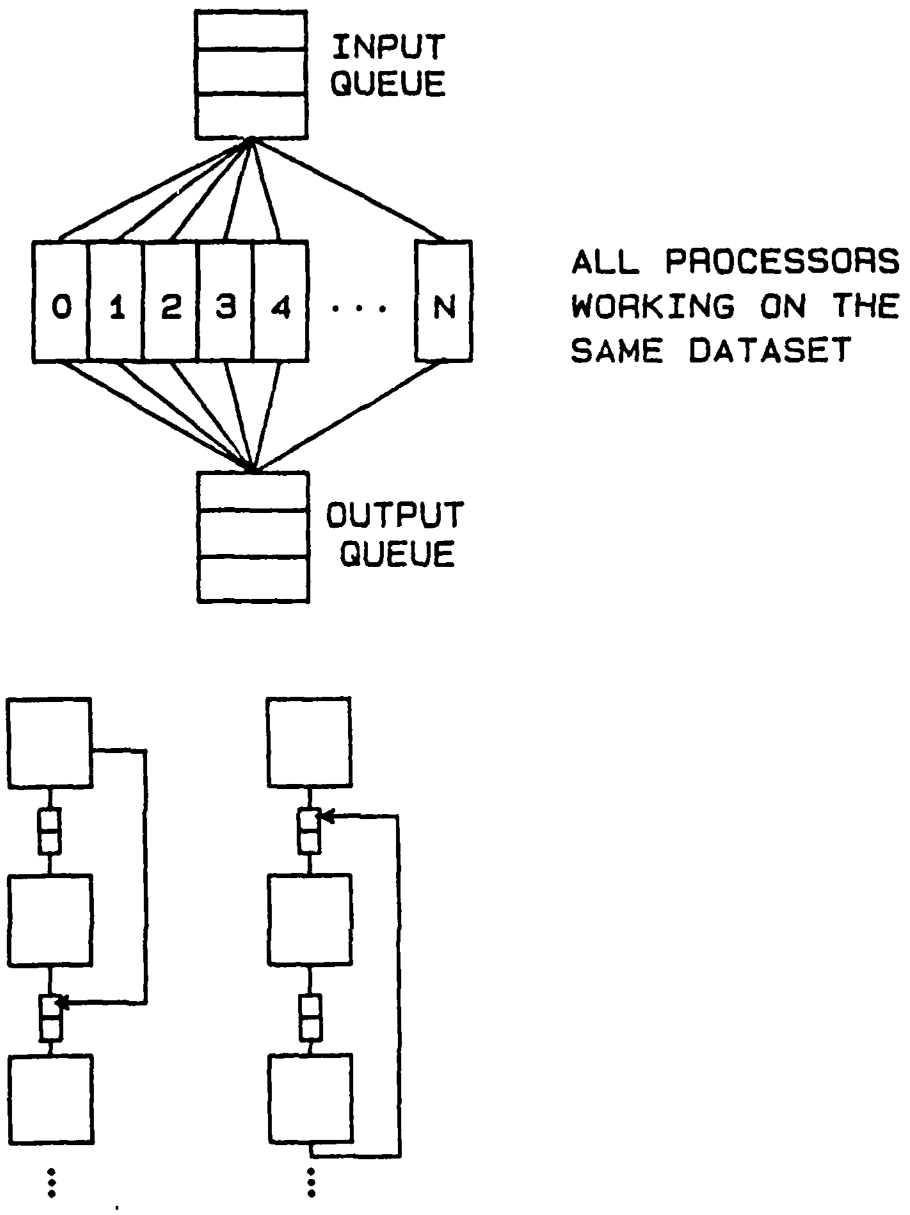


Fig. 5.1.1 Allowable architectures and feedback paths

necessary) in the input queue for the next level.

There are other types of parallelism, such as the multiprocessors with multiple data sets mentioned above, which will not be considered here. Instead, their analysis will be left as future work.

Fig. 5.1.1 (b) shows two asynchronous systems with **feedback**. For the purposes of this research, feedback is defined to be any data set in the input queue for a level i that did not come from level i−1 . By this definition, feedforward (from levels other than i−1) is also treated as feedback. (from levels other than i−1) is treated as feedback. Here, it is assumed that feedback data sets can arrive asynchronously and are normal data sets as far as size and processing requirements are concerned. Feedback may be required when a data set needs further reprocessing, e.g., processing with different parameters because it is later found that some criterion is not met. Feedforward may be used when a particular data set does not need to be processed by a specific level or levels. Synchronous systems, by their nature, cannot have feedback.

Initially, four assumptions will be made. They are:

(1) Two input data sets cannot arrive at a given level simultaneously i.e., feedback is not allowed. (If there is no level-to-level feedback, it is impossible for multiple data sets to arrive at a given level simultaneously.) This restriction will be removed later.

(2) At a given level, the arrival of a particular data set is independent of the arrival of any other data set. Thus, the arrival of data sets to be processed by a level can be treated like events in a Poisson random process.

(3) At a given level, the average arrival rate of data sets to be processed is λ_{ar}.

(4) The probability of an arrival at a given level during a given time interval is a function of the interval duration, not the beginning time of the interval. For a very small time interval Δt, the probability of an arrival is $\lambda_{ar}\Delta t$.

Section 5.2 discusses the determination of $P_v(t)$ for a single input stream, single processing stream system; i.e., no feedback. The information presented in Section 5.2 represents a derivation of the results presented in [Ful75]. It is the purpose of this section to provide necessary background information to clarify the discussion of topics appearing elsewhere in this chapter. Section 5.2 also states the results of the theory for a multiple input stream case; i.e. feedback is allowed. Section 5.3 continues the derivation started in Section 5.2 to determine the expected size of a level-level queue.

Based on the previous sections, Section 5.4 compares the performance of an asynchronous system with the performances of the both double- and triple-buffered synchronous systems. In Section 5.4.3 the performance of both double- and triple-buffered synchronous systems are analyzed for two level systems where the response time of the first level is fixed and the response time of the second level is either a uniform random variable or a Gaussian random variable. Section 5.4.4 applies the techniques presented in Section 5.3 to derive the expected throughput and response time of an asynchronous system. Section 5.4.5 contrasts the performance of the two synchronous systems and the asynchronous system when the response times of the two levels are random variables. $\overline{Q}$ for an asynchronous system is discussed in Section 5.4.6. Section 5.4.7 contains a discussion of the applications of double- and triple-buffering systems. The advantages and disadvantages of synchronous systems and asynchronous systems are summarized in Section 5.4.8.

Sections 5.2, 5.3, and 5.4 all deal with the theoretical expectations of the system throughput. To verify the results presented in these sections, Section 5.5 presents results obtained from simulation.

5.2. Determination of $P_v(t)$ For a Single Input/Processing Stream

The following derivation is similar to that in [Ful75]. Setting Δt as the time interval under consideration, the probability of a data set arriving is:

$$P_{\geq 1}(\Delta t) = \lambda_{ar} \times \Delta t + P_{\geq 2}(\Delta t)$$

where $P_{\geq 2}(\Delta t)$ is the probability of at least two data sets arriving. $P_v(\Delta t)$ for $v>1$ is zero if the previous layer produces at most one result at a time, if there is no feedback, and if Δt is short (i.e., $\Delta t \times \lambda_{ar} << 1$).

The case for $v > 0$ arrivals during a time interval $t+\Delta t$ is:

$$P_v(t+\Delta t) = P_{v-1}(t) \times P_1(\Delta t) + P_v(t) \times P_0(\Delta t)$$

During one time epoch (Δt), at most one arrival can occur. Thus, either zero jobs or one job can arrive during the interval Δt. $\dot{P}_v(t)$ can be obtained through the fundamental definition of differentiation:

$$\dot{P}_v(t) = \lim_{\Delta t \to 0} \frac{P_v(t+\Delta t) - P_v(t)}{\Delta t}$$

Since $P_1(\Delta t) = \lambda_{ar}\Delta t$. $P_0(\Delta t) = 1 - P_1(\Delta t) = 1 - \lambda_{ar}\Delta t$. Thus,

$$\dot{P}_v(t) = P_{v-1}(t)\lambda_{ar} - P_v(t)\lambda_{ar} \quad (v>0)$$

Taking the Laplace transform of this equation (assuming that $\lim_{\Delta t \to 0} P_{v \neq 0}(\Delta t) = 0$), yields:

$$P_v(s) = \frac{1}{(s+\lambda_{ar})} \operatorname{bprod}_{i=1}^{v} \frac{\lambda_{ar}}{(s+\lambda_{ar})}$$

Taking the inverse Laplace transform yields:

$$P_v(t) = \frac{(\lambda_{ar}t)^v}{v!} e^{-\lambda_{ar}t}$$

The application of the above equation is limited to a system with a single input stream capable of sending one job at a given time and a single processing stream producing a single result. This type of analysis makes it possible to determine the probability of a level receiving a given number of arrivals by a certain time t.

By applying the results in [Cin75], P_v for r independent Markovian streams is:

$$P_v(t) = \frac{(\Lambda_L t)^v}{v!} e^{-\Lambda_L t}$$

where Λ_L is: $\sum_{i=1}^{r} \lambda_i$ and λ_i is the arrival rate for stream i.

5.3. The Expected Queue Size of an Asynchronous System

The above results can be used to calculate the **Expected Interarrival Time** (**EIT**) as follows:

$$P\left\{\text{average interarrival time} \leq t\right\} = 1 - P_0(t)$$

taking the derivative yields:

$$p\left\{\text{average interarrival time} = t\right\} = -\, p_0(t)$$

where, $p_v(t) = \dot{P}_v(t)$

$$p_0(t) = e^{-\sum_{i=1}^{r} \lambda_i t} \times \left[\sum_{i=1}^{r} -\lambda_i\right]$$

Thus,

$$\text{EIT} = \left(-\sum_{i=1}^{r} \lambda_i\right) \int_{t=0}^{t=\infty} e^{-\sum_{i=1}^{r} \lambda_i t}\, t\, dt$$

$$= \frac{e^{-\sum_{i=1}^{r} \lambda_i t}}{\sum_{i=1}^{r} \lambda_i} \left(\sum_{i=1}^{r} \lambda_i t + 1\right) \Big|_{t=0}^{\infty} = \frac{1}{\sum_{i=1}^{r} \lambda_i}$$

Service of the arriving jobs is less complex because it is a valid assumption that only one job may be removed from the queue at a time. Because data sets may be related, the servicing of the data sets is not necessarily Markovian. By defining Λ_L to be $\frac{1}{\text{EIT}}$ for jobs arriving at level L, μ_L to be the average throughput of level L, and C to be $\frac{\text{standard deviation of service time}}{\text{expected service time}}$, the Pollaczek-Khinchine formula [Ful75] along with work from [CoM67], can be applied to determine the expected queue length as:

$$\overline{Q} = \frac{2\Lambda_L(\mu_L - \Lambda_L) + \Lambda_L{}^2(C^2 + 1)}{2\mu_L(\mu_L - \Lambda_L)}$$

This is the expected queue length of an M/G/1 queuing structure (Markovian arrival process, General distribution service structure, 1 processor serving queue).

5.4. A Comparison of Synchronous and Asynchronous Systems

5.4.1. Introduction

Since $P_v(t)$ and $\overline{Q}$ can be calculated, the throughput and response time of a system with asynchronous levels can be compared with a system whose levels are running in synchrony. Several metrics must be considered to perform this comparison. While such metrics as expected queue size, wait time (in the queue), and expected run time all have a meaning for an asynchronous system, their use for a synchronous system is limited. Worst case speed in an asynchronous system reflects itself in the expected size of the buffer between two asynchronous levels, but does not bear the same significance in a synchronous system, where it is used to calculate the run time of a system.

5.4.2. Initial System Models -- Three Potential Architectural Schemes

Consider the proposed systems shown in Fig. 5.4.2.1. Each of the proposed systems contains two levels. This model can be extended to systems of multiple (>2) levels, by repetitively analyzing the system in terms of two level pieces. This is an iterative process. For example: for an L level system, all levels $2i$ and $2i+1$ ($i < \frac{L}{2}$) would be analyzed as two level systems. Then, the statistics for these systems (consisting of two levels) would be combined in groups of two. The resulting analysis would then parameterize the performance of the four level "systems." This process can be repeated until there is one set of parameters to describe the throughput of an entire system. Because of the simplicity of analysis and applicability of the analysis, only two level systems are considered here.

The first system in Fig. 5.4.2.1 is a synchronous double-buffered system, the second a synchronous triple-buffered system (as discussed in Section 4.5), and the third an asynchronous system. It will be assumed that the both of the synchronous systems are of the type where both levels report to an arbitrator when they have completed processing (the first level to report waits until the last level reports). It is the goal of this discussion to relate the response time, throughput, and memory requirements of the three types of architectures. To this end, the discussion will assume that the first level can perform its calculations in a fixed amount of time (this restriction will be removed later). It will be assumed that the exact execution time for the second level is unknown, but that it can be described probabilistically. This is similar to the earlier discussion about the isolated speech recognition system, in which the

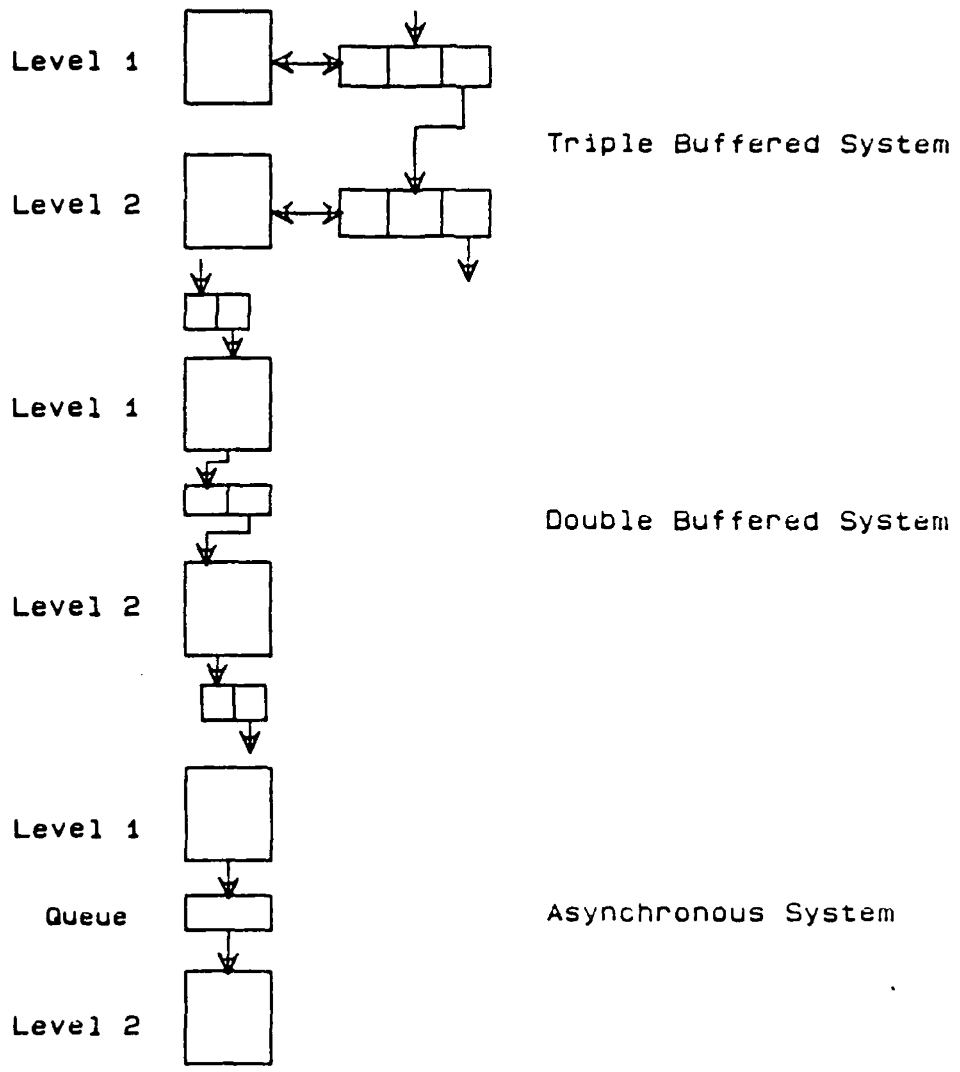


Fig. 5.4.2.1 Three architectures under consideration

fourth level required a fixed computational time and the fifth level required a variable time.

5.4.3. Analysis of Synchronous Models with Two Probabilistic Models

If t_i is the actual time that level i requires to process a given data set and $pr_i(t)$ is probability that level i will process any data set in time t, the expected processing time (**EPT**) of each level for the synchronous systems can be defined as follows:

$$EPT = \int_{t=0}^{t=t_1} t_1\, pr_2(t)dt + \int_{t=t_1}^{t=\infty} t\, pr_2(t)dt$$

Since the system is running in synchrony, the faster level must wait for the slower level to respond before its processing can continue. The addend (first term in the sum) represents the time that the system will spend when the second level responds more quickly than the first. Here, the response time of the system is t_1. The probability that the response time of the system is t_1 is equal to the sum of the probabilities of all cases where t_2 is less than t_1, hence the integral.

The augend (second term in the sum) results from the second level responding more slowly than the first. In this case, the response time of the second level dictates the response time of the entire system, thus the t times $pr_2(t)$ is the expected response time of the system when the second level of the system responds more slowly than the first level.

In a synchronous environment, the minimum time that a data set can spend at a given level is t_1. The processing time for data set D is:

$$\max\left[t_1(\text{for D}), t_2(\text{for D}-1)\right] + \max\left[t_1(\text{for D}+1), t_2(\text{for D})\right]$$

For the double-buffered system, the expected system response time ($\overline{\text{SRT}}$) is: $2 \times \text{EPT}$. In general, this is $N_L \times \text{EPT}$, where N_L is the number of levels in the system. The throughput of this system is 1/EPT. In contrast, the triple-buffered system requires time EPT to transfer the data set from one level to the next, thus $\overline{\text{SRT}}$ of the triple buffered system is: $2 \times \text{EPT}$ (one time unit is required to load data into a level and one time unit is required to process the data). For the two level case considered here, this is: $4N_L \times \text{EPT}$. The throughput of the triple-buffered system is the same as the double-buffered system.

Analysis of the cases where the levels are running in synchrony (i.e., all levels must complete their present data set before any can go onto the next data set) can be obtained by applying this (previously mentioned) equation:

$$\text{EPT} = \int_{t=0}^{t=t_1} t_1 pr_2(t)dt + \int_{t=t_1}^{t=\infty} t pr_2(t)dt$$

The addend of EPT is evaluated as follows (where level two is Gaussian with mean response time and standard deviation of $\bar{t}_2$ and $\sigma_2 = C\bar{t}_2$ respectively):

$$\int_{t=0}^{t=t_1} t_1 pr_2(t)dt$$

$$= t_1 \int_{t=0}^{t=t_1} \frac{1}{\sqrt{2\pi}(C\bar{t}_2)} \exp\left\{\frac{-\left[t-(\bar{t}_2)\right]^2}{2(C\bar{t}_2)^2}\right\} dt$$

$$= t_1\, \Phi\left[\frac{t_1-\bar{t}_2}{C\bar{t}_2}\right]$$

Φ is the Gaussian probability function with mean 0 and standard deviation 1 [Pap65]. σ_2 was set to $C\bar{t}_2$ so that the results could be expressed in terms of t_1. For this last equation to hold, the quantity: $1 - \Phi\left[\frac{1}{C}\right]$ must be zero. For values of C larger than 0.4, a Gaussian distribution function would require some modifications (e.g., a δ function for $pr_2(0)$) to be valid. When $\bar{t}_2 = bt_1$, this equation simplifies to:

$$= t_1\, \Phi\left[\frac{1-b}{Cb}\right]$$

The augend of EPT is evaluated as follows:

$$\int_{t=t_1}^{t=\infty} t\,pr_2(t)dt$$

$$= \int_{t = t_1}^{t = \infty} \frac{t}{\sqrt{2\pi}(C\bar{t}_2)} \exp\left\{\frac{-\left[t - (\bar{t}_2)\right]^2}{2(C\bar{t}_2)^2}\right\} dt$$

by defining $u = t - \bar{t}_2$ ($du = dt$), this simplifies to:

$$= \int_{u = t_1 - \bar{t}_2}^{u = \infty} \frac{u + \bar{t}_2}{\sqrt{2\pi}(C\bar{t}_2)} \exp\left\{\frac{-[u^2]}{2(C\bar{t}_2)^2}\right\} du$$

Splitting the integral into two portions and simplifying the augend yields:

$$= \int_{u = t_1 - \bar{t}_2}^{u = \infty} \frac{u}{\sqrt{2\pi}(C\bar{t}_2)} \exp\left\{\frac{-[u^2]}{2(C\bar{t}_2)^2}\right\} du + \bar{t}_2 \left[1 - \Phi\left(\frac{t_1 - \bar{t}_2}{C\bar{t}_2}\right)\right]$$

Substituting $z = \dfrac{u^2}{2(C\bar{t}_2)^2}$ *yields:*

$$= \int_{z = \frac{(t_1 - \bar{t}_2)^2}{2(C\bar{t}_2)^2}}^{z = \infty} \frac{(C\bar{t}_2)}{\sqrt{2\pi}} e^{-z}\, dz \quad + \quad \bar{t}_2 \left[1 - \Phi\left(\frac{t_1 - \bar{t}_2}{C\bar{t}_2}\right)\right]$$

Performing the integration over the limits yields:

$$= \frac{C\bar{t}_2}{\sqrt{2\pi}} \exp\left[\frac{-(t_1 - \bar{t}_2)^2}{2(C\bar{t}_2)^2}\right] + \bar{t}_2 \left[1 - \Phi\left(\frac{t_1 - \bar{t}_2}{C\bar{t}_2}\right)\right]$$

Allowing $\bar{t}_2 = bt_1$ yields:

$$= \frac{Cbt_1}{\sqrt{2\pi}} \exp\left[\frac{-(1-b)^2}{2(Cb)^2}\right] + bt_1\left[1 - \Phi\left(\frac{1-b}{Cb}\right)\right]$$

The resultant equation for EPT (for a Gaussian time distribution) is as follows:

$$EPT = t_1(1-b)\left[\Phi\left(\frac{1-b}{Cb}\right)\right] + \frac{Cbt_1}{\sqrt{2\pi}} \exp\left[\frac{-(1-b)^2}{2(Cb)^2}\right] + bt_1$$

Here,

$$\overline{SRT}(\text{double buffered}) = 2\times EPT$$

$$\overline{SRT}(\text{triple buffered}) = 4\times EPT$$

and the expected system throughput ($\overline{ST}$)

$$\overline{ST} = \frac{1}{EPT}$$

Table 5.4.3.1 shows the effect of b and C on $\overline{SRT}$ and the system throughput $\overline{ST}$ when the processing time of level two can be represented by a Gaussian distribution function. In addition, Table 5.4.3.1 shows that the greater the probability that t_2 is greater than t_1, the lower the throughput. In addition, a triple-buffered system will have the same throughput as a double-buffered system, but the triple-buffered system will require one extra delay for each level in the system.

Consider the case where the response time of level two can be described by a uniform distribution function. (For this discussion, it is assumed that it is possible for t_2 to be larger that t_1. If this is not the case, $\overline{EPT} = t_1$.) Again, let $\bar{t}_2 = bt_1$ and $\sigma = C\bar{t}_2$. If $t_2(max)$ and $t_2(min)$ are the largest and smallest response times respectively, then $pr_2(t) = \frac{1}{t_2(max)-t_2(min)}$. From [Pap65], it

Table 5.4.3.1.

Double buffered system (DB) and triple buffered system (TB): $\overline{SRT}$ and $\overline{ST}$ when t_1 is fixed and t_2 Gaussian random variable.

C	$\bar{t}_2$	b	$\overline{SRT}$ (DB)	$\overline{SRT}$ (TB)	$\overline{ST}$
1.00	$0.50t_1$	0.50	$2.08t_1$	$4.16t_1$	$0.96/t_1$
1.00	$0.75t_1$	0.75	$2.38t_1$	$4.76t_1$	$0.84/t_1$
0.75	$0.50t_1$	0.50	$2.03t_1$	$4.06t_1$	$0.98/t_1$
0.75	$0.75t_1$	0.75	$2.24t_1$	$4.48t_1$	$0.89/t_1$
0.50	$0.50t_1$	0.50	$2.00t_1$	$4.00t_1$	$0.99/t_1$
0.50	$0.75t_1$	0.75	$2.11t_1$	$4.22t_1$	$0.95/t_1$
0.25	$0.50t_1$	0.50	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.25	$0.55t_1$	0.55	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.25	$0.60t_1$	0.60	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.25	$0.65t_1$	0.65	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.25	$0.70t_1$	0.70	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.25	$0.75t_1$	0.75	$2.01t_1$	$4.01t_1$	$0.99/t_1$
0.25	$0.80t_1$	0.80	$2.01t_1$	$4.02t_1$	$0.99/t_1$
0.25	$0.85t_1$	0.85	$2.01t_1$	$4.02t_1$	$0.99/t_1$
0.25	$0.90t_1$	0.90	$2.03t_1$	$4.06t_1$	$0.98/t_1$
0.25	$0.95t_1$	0.95	$2.09t_1$	$4.18t_1$	$0.96/t_1$
0.10	$0.50t_1$	0.50	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.55t_1$	0.55	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.60t_1$	0.60	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.65t_1$	0.65	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.70t_1$	0.70	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.75t_1$	0.75	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.10	$0.80t_1$	0.80	$2.00t_1$	$4.00t_1$	$1.00/t_1$

can be shown that:

$$\sigma = \frac{t_2(\max)-t_2(\min)}{\sqrt{12}} = C\bar{t}_2$$

Given the the above, the equalities:

$$t_2(\max) = \bar{t}_2 + \frac{t_2(\max) - t_2(\min)}{2}$$

$$t_2(\min) = \bar{t}_2 - \frac{t_2(\max) - t_2(\min)}{2}$$

and $\bar{t}_2 = bt_1$, it can be shown that:

$$t_2(\max) = t_1\,(b + \sqrt{3}Cb)$$

$$t_2(\min) = t_1\,(b - \sqrt{3}Cb)$$

Since only non-negative values of time are allowed, $C \leq \frac{1}{\sqrt{3}}$. The EPT can be determined through the following derivation.

$$\overline{EPT} = \int_{t\,=\,t_2(\min)}^{t_1} t_1\, pr_2(t)dt \quad + \int_{t\,=\,t_1}^{t\,=\,t_2(\max)} t\, pr_2(t)dt$$

$$= \int_{t = t_1(b - \sqrt{3}Cb)}^{t_1} t_1\, pr_2(t)dt + \int_{t = t_1}^{t = t_1(b + \sqrt{3}Cb)} t\, pr_2(t)dt$$

Completing the integration and using the fact that $PR_2(t) = \frac{t-t_2(\min)}{\sqrt{12}Cbt_1}$ ($PR_2(t)$ is the probability that the response time of level two is less than or equal to t) yields:

$$= \frac{t_1\,(1 - b + \sqrt{3}Cb)}{\sqrt{12}Cb} + \frac{(b^2 + \sqrt{12}Cb^2 + 3C^2b^2 - 1)t_1}{2\sqrt{12}Cb}$$

This simplifies to

$$\overline{EPT} = \frac{(1 - 2b + 2\sqrt{3}Cb + b^2 + 2\sqrt{3}Cb^2 + 3C^2b^2)t_1}{4\sqrt{3}Cb}$$

Table 5.4.3.2 shows the expected response times and throughput of systems whose response times can modeled by uniformly distributed random variable.

5.4.4. Analysis of an Asynchronous System -- Two Probabilistic Models

Now, consider the case where the levels operate asynchronously. t_1 is assumed to be constant, as in the synchronous case. If t_2 can never exceed t_1, $\overline{EPT}$ is t_1. For all such cases, a queue of length 1 is sufficient. In general, if λ

Table 5.4.3.2.

$\overline{SRT}$(DB), $\overline{SRT}$(TB), and $\overline{ST}$ for t_1 fixed and t_2 uniform random variable.

C	$\bar{t}_2$	b	$\overline{SRT}$ (DB)	$\overline{SRT}$ (TB)	$\overline{ST}$
0.577	$.50t_1$	0.50	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.577	$.55t_1$	0.55	$2.00t_1$	$4.00t_1$	$1.00/t_1$
0.577	$.60t_1$	0.60	$2.03t_1$	$4.06t_1$	$0.98/t_1$
0.577	$.65t_1$	0.65	$2.07t_1$	$4.14t_1$	$0.97/t_1$
0.577	$.70t_1$	0.70	$2.11t_1$	$4.22t_1$	$0.95/t_1$
0.577	$.75t_1$	0.75	$2.17t_1$	$4.34t_1$	$0.92/t_1$
0.577	$.80t_1$	0.80	$2.23t_1$	$4.46t_1$	$0.89/t_1$
0.577	$.85t_1$	0.85	$2.29t_1$	$4.58t_1$	$0.87/t_1$
0.577	$.90t_1$	0.90	$2.36t_1$	$4.72t_1$	$0.84/t_1$

is the arrival rate of jobs from the previous level $\left(=\frac{1}{t_1}\right)$ and μ is the rate at which the present level processes data sets $\left(=\frac{1}{\int\limits_{t=0}^{\infty} tp_2(t)dt}\right)$ then the expected queue size for an M/G/1 system can be determined by the following equation [Che80]:

$$\overline{Q} = \frac{\lambda}{\mu - \lambda}$$

Clearly, as μ approaches λ, the expected queue size gets arbitrarily large. If t_2 exists over a finite range, then $t_2(\max)$ may be substituted for the ∞. This equation simplifies to:

$$\overline{Q} = \frac{\overline{t}_2}{t_1 - \overline{t}_2}$$

The expected waiting time in the queue can be determined by the equation [Ful75]: $\overline{W} = \overline{t}_2 \times \overline{Q}$. Table 5.4.4.1 shows the expected queuelength ($\overline{Q}$), the expected waiting time in the queue ($\overline{W}$), the expected system response time ($\overline{SRT}$), and the expected system throughput ($\overline{ST}$) as a function of $\overline{t}_2$ when the response time of level 1 is a constant t_1. For the calculation of this table, it was assumed that the data arrived at level 1 no faster than one job per time t_1, i.e., the system could keep up with the incoming data. This is the same assumption made for the synchronous case.

Table 5.4.4.1.

The expected queue size ($\overline{Q}$), expected time spent in a queue ($\overline{W}$), the expected system response time ($\overline{SRT}$), and the expected system throughput ($\overline{ST}$) for an asynchronous system with t_1 fixed and t_2 an arbitrary random variable.

$\overline{t_2}$	b	$\overline{Q}$	$\overline{W}$	$\overline{SRT}$	$\overline{ST}$
0.50 t_1	0.50	1.00	0.50 t_1	2.00t_1	1.00/t_1
0.55 t_1	0.55	1.20	0.66 t_1	2.21t_1	1.00/t_1
0.60 t_1	0.60	1.50	0.90 t_1	2.50t_1	1.00/t_1
0.65 t_1	0.65	1.86	1.21 t_1	2.85t_1	1.00/t_1
0.70 t_1	0.70	2.33	1.63 t_1	3.33t_1	1.00/t_1
0.75 t_1	0.75	3.00	2.25 t_1	4.00t_1	1.00/t_1
0.80 t_1	0.80	4.00	3.20 t_1	5.00t_1	1.00/t_1
0.85 t_1	0.85	5.67	4.82 t_1	6.67t_1	1.00/t_1
0.90 t_1	0.90	9.00	8.10 t_1	10.00t_1	1.00/t_1
0.95 t_1	0.95	19.00	18.05 t_1	20.00t_1	1.00/t_1

$\overline{SRT}$ for the asynchronous system is be calculated as follows:

$$\overline{SRT} = t_1 + \bar{t}_2 + \overline{W}$$

When results of this analysis are compared with the previous results, the asynchronous system will yield up to a 18% greater throughput. Using asynchronous hardware will provide greater utilization of the hardware. Further, the asynchronous system will not need hardware to transfer data from one swinging-buffer to another, unlike the triple-buffering scheme.

For the applications discussed earlier, it would seem that the asynchronous systems are "the way to go." While, in general, they are feasible, there are specific cases where it may not be advantageous to use an asynchronous system; e.g., when response time is critical. While an asynchronous system has a higher average throughput, for specific data sets there may be a significant delay caused by time spent in a queue. The worst-case response time of an asynchronous system could be greater than some threshold. In such an event, an asynchronous system would not be desirable. On the average, however, asynchronous systems offer higher throughput than their synchronous.

5.4.5. Analysis of Systems Composed of Two Levels Whose Response Times Are Random Variables

The previous discussion held t_1 to be a fixed entity. Now, consider the case where t_1 can vary. It will be assumed; however, that t_1 is Markovian. For synchronous systems, the analysis must be divided into three parts because of the three distinct ways that the times for the levels can be related. These three cases are shown in Fig. 5.4.5.1. (The dashed line represents the range of processing times for t_2 and the solid line represents the range of processing times for t_1.) For the first case, EPT is $\overline{t}_1$. This the least complex and least useful situation. Clearly, if there is no overlap between the processing times of the levels, one of the levels is processing more quickly than is needed; consequently, the stages of the pipeline are not balanced. Thus, the faster level could be built with slower and presumably less expensive hardware.

Now, consider the case where the time span for t_1 overlaps with t_2. Assume that the maximum and minimum possible values for t_i are $t_i(\max)$ and $t_i(\min)$. EPT is (recall $pr_i(t)$ is the probability that level i will have response time t and that $PR_i(t)$ is the probability that level i will respond faster than time t):

$$EPT = \int_{t=t_2(\min)}^{t_1(\min)} \overline{t}_1\, pr_2(t)dt + \int_{t=t_1(\min)}^{t_2(\max)} t\, pr_2(t)\, PR_1(t_1 \le t)dt$$

$$+ \int_{t=t_1(\min)}^{t_2(\max)} t\, pr_1(t)\, PR_2(t_2 \le t)dt + \int_{t=t_2(\max)}^{t_1(\max)} t\, pr_1(t)dt$$

This can be transformed into the following equation:

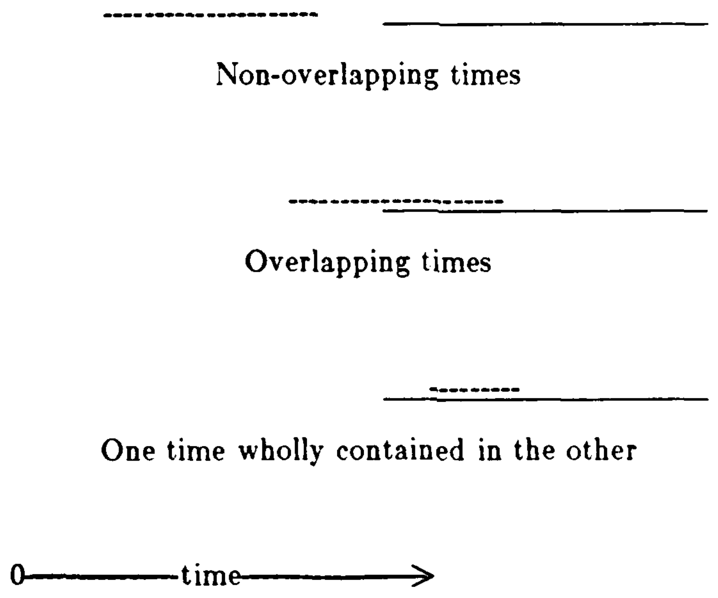


Fig. 5.4.5.1. Three orientations of t_2 rela:ive to t_1

$$\mathrm{EPT} = \int_{t=t_2(\min)}^{t_1(\min)} \bar{t}_1 \, pr_2(t)dt + \int_{t=t_1(\min)}^{t_2(\max)} t \, pr_2(t) \int_{x=t_1(\min)}^{t} pr_1(x) \, dx \, dt$$

$$+ \int_{t=t_1(\min)}^{t_2(\max)} t \, pr_1(t) \int_{x=t_1(\min)}^{t} pr_2(x) \, dx \, dt + \int_{t=t_2(\max)}^{t_1(\max)} t \, pr_1(t)dt$$

Because of the complexity of this integral, its value must be calculated numerically if the distribution of $pr_1(t)$ or $pr_2(t)$ is Gaussian. For the following discussion, assume that $p_1(t)$ and $p_2(t)$ are both uniform distributions. Recall that: $pr_i(t) = \dfrac{1}{t_i(\max) - t_i(\min)}$ that $t_i(\max)$ is $\bar{t}_i(1 + \sqrt{3}C)$ and that $t_i(\min)$ is $\bar{t}_i(1 - \sqrt{3}C)$. If the standard deviation and mean of level 1 are σ_1 and $\bar{t}_1$ respectively and the standard deviation and mean of level 2 are σ_2 and $\bar{t}_2$ respectively, the following equation is the evaluation of the above integral. C_i is defined to be $\dfrac{\sigma_i}{\bar{t}_i}$ and b is $\dfrac{\bar{t}_2}{\bar{t}_1}$. Again, C_i can be no greater than .577 if only positive values of t_i are to be allowed.

$$\mathrm{EPT} = \frac{\bar{t}_1\left[(1-\sqrt{3}C_1)-(b-b\sqrt{3}C_2)\right]}{2b\sqrt{3}C_2}$$

$$+ \frac{\bar{t}_1\left[2b^3(1+\sqrt{3}C_2)^3+(1-\sqrt{3}C_1)^3-3b^2(1+\sqrt{3}C_2)^2(1-\sqrt{3}C_1)\right]}{36C_1C_2b}$$

$$+ \frac{\bar{t}_1\left[(1+\sqrt{3}C_1)^2-(b+bC_2\sqrt{3})^2\right]}{4\sqrt{3}C_1}$$

Table 5.4.5.1 applies this equation to derive the expected response time and system throughput of a double- and triple-buffered synchronous system as a function of b, C_1, and C_2.

The final case, where one interval is contained in the other, is similar to the previous case. EPT can be defined as follows (when the possible response times of level 2 are a subset of those for level 1):

$$\mathrm{EPT} = \int_{t=t_1(\min)}^{t_2(\min)} \bar{t}_2\, pr_1(t)dt + \int_{t=t_2(\min)}^{t_2(\max)} t\, pr_2(t) \int_{x=t_2(\min)}^{t} pr_1(x)\, dx\, dt$$

$$+ \int_{t=t_2(\min)}^{t_2(\max)} t\, pr_1(t) \int_{x=t_2(\min)}^{t} pr_2(x)\, dx\, dt + \int_{t=t_2(\max)}^{t_1(\max)} t\, pr_1(t)dt$$

The analysis for this case is similar to the previous analysis and it is omitted here for brevity.

Now consider an asynchronous system with the same two levels. For this analysis, level 1 is assumed to be Markovian. Any probability function that can describe the response time of level 2 will be allowed. If level 1 runs continuously then application of the Pollaczek-Khinchine formula predicts the expected queue length to be [Ful75]:

Table 5.4.5.1.

Statistics for a synchronous system with both t_1 and t_2 uniform random variables.

b	C_1	C_2	$\overline{SRT}$(DB)	$\overline{SRT}$(TB)	$\overline{ST}$
0.9	0.500	0.577	$2.49t_1$	$4.98t_1$	$0.81/t_1$
0.8	0.500	0.577	$2.42t_1$	$4.84t_1$	$0.83/t_1$
0.7	0.500	0.577	$2.35t_1$	$4.20t_1$	$0.85/t_1$
0.9	0.400	0.577	$2.40t_1$	$4.80t_1$	$0.83/t_1$
0.8	0.400	0.577	$2.36t_1$	$4.72t_1$	$0.85/t_1$
0.7	0.400	0.577	$2.35t_1$	$4.72t_1$	$0.85/t_1$

$$\overline{Q} = \frac{\dfrac{2}{\overline{t}_1}\left(\dfrac{1}{\overline{t}_2} - \dfrac{1}{\overline{t}_1}\right) + \dfrac{(C^2 + 1)}{\overline{t}_1^2}}{\dfrac{2}{\overline{t}_2}\left(\dfrac{1}{\overline{t}_2} - \dfrac{1}{\overline{t}_1}\right)}$$

Here, $C = \frac{\sigma_2}{\overline{t}_2}$. σ_2 is the standard deviation of the processing time for level 2. Multiplying $\overline{Q}$ by $\overline{t}_2$ yields the expected wait time in the queue. Thus, for an asynchronous system the expected SRT is:

$$\overline{SRT} = \overline{t}_1 + \left[\overline{t}_2 + \frac{\dfrac{2}{\overline{t}_1}\left(\dfrac{1}{\overline{t}_2} - \dfrac{1}{\overline{t}_1}\right) + \dfrac{(C^2 + 1)}{\overline{t}_1^2}}{\dfrac{2}{\overline{t}_2^2}\left(\dfrac{1}{\overline{t}_2} - \dfrac{1}{\overline{t}_1}\right)}\right]$$

The expected throughput of the asynchronous system is:

$$\overline{ST} = \frac{1}{\max(t_1, \overline{t}_2)}$$

The memory requirements of the double-buffered system are $2N_L$ data sets (N_L is the number of levels in the system), while the memory requirements of the triple-buffered system is $3N_L$ data sets. Finally, allowing a double input buffer for the first level of the asynchronous system, its memory requirements are $2 + (N_L - 1)\overline{Q}$ data sets.

To relate the response time and throughput of the various systems, there are three cases that can arise. $\overline{t}_2$ can be less than, equal to, or greater than $\overline{t}_1$. Applying the last case to the asynchronous system; it is assumed that a new data set is arriving every $\overline{t}_1$ seconds, thus the queue for level 2 would be required to grow without bound -- clearly not feasible. Synchronous systems would lose data sets in this event. A similar situation arises for the cases where $\bar{t}_2 = \bar{t}_1$.

The following analysis will show how the three architectures behave when $\bar{t}_2 < t_1$. The distributions used to describe t_2 will be the normal distribution (Gaussian) and the exponential distribution. Mean values of t_2 considered will be $.5t_1$ and $.75t_1$.

Figs. 5.4.5.2, 5.4.5.3, and 5.4.5.4 show the effects of $\bar{t}_2$ and C on the expected queue length for the second level of the asynchronous system. Fig. 5.4.5.5 shows the effects of C on the expected queue length of an asynchronous system. C and $\bar{t}_2$ affect the time that a job spends waiting in a queue. For systems in which the ratio of the standard deviation to the mean (C) is small, the expected response time of level two does not have to be as fast as would be required by larger values of C. Note: values of C larger than one are meaningless because such a condition implies that negative processing times are possible for level two. Table 5.4.5.2 shows the response time and throughput of an asynchronous system where each data set is processed only once, i.e., where data sets are not fed back for more processing ($\overline{W}$ is the expected wait time in the queue, and $\overline{ST}$ is the expected system throughput):

This table is true when the service distribution of the for the second level is a general distribution. For an exponential distribution, C is defined to be one. The results printed in this table merit discussion because they are not intuitive. Consider the following diagram:

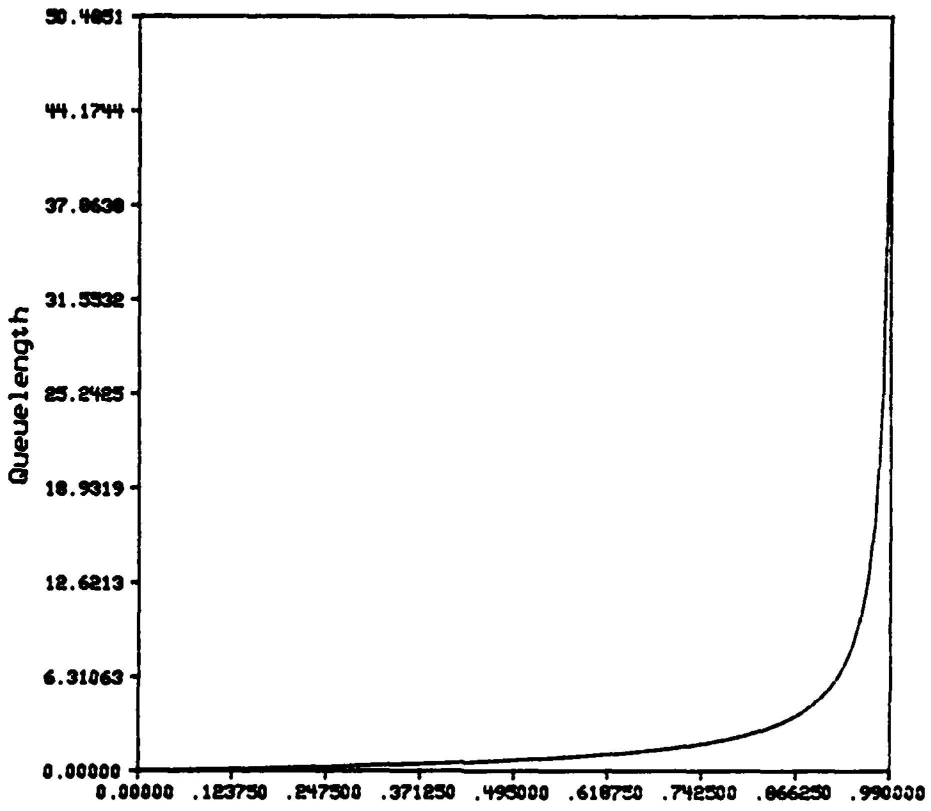


Fig. 5.4.5.2 Queuelength as f(t2) c=0.1

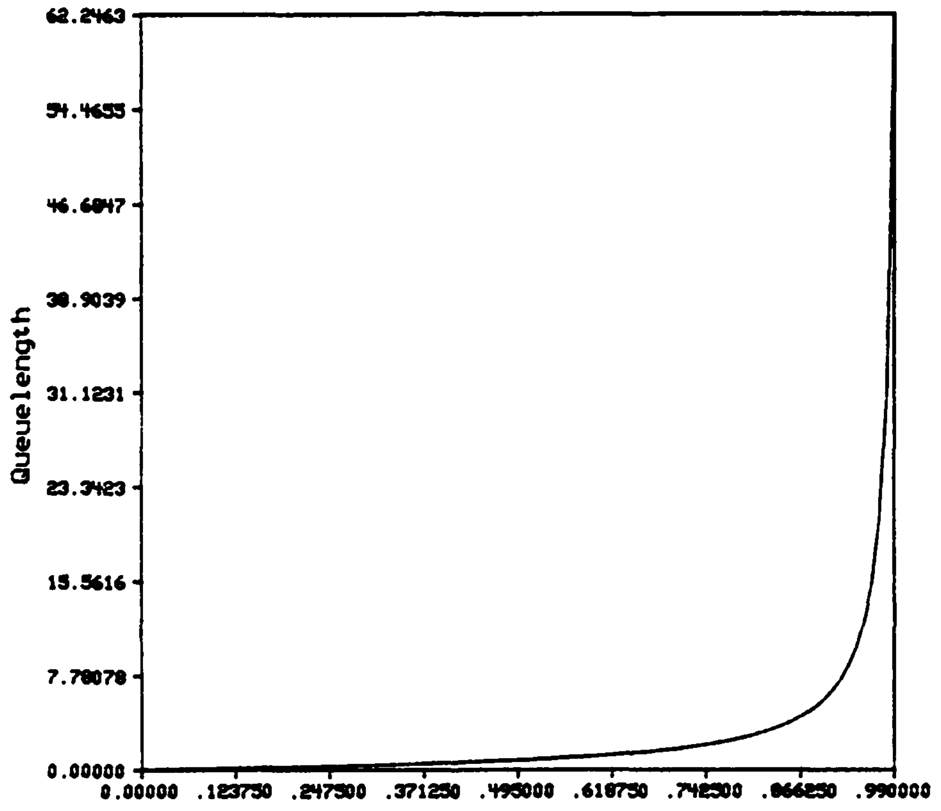


Fig. 5.4.5.3 Queuelength as f(t2) c=0.5

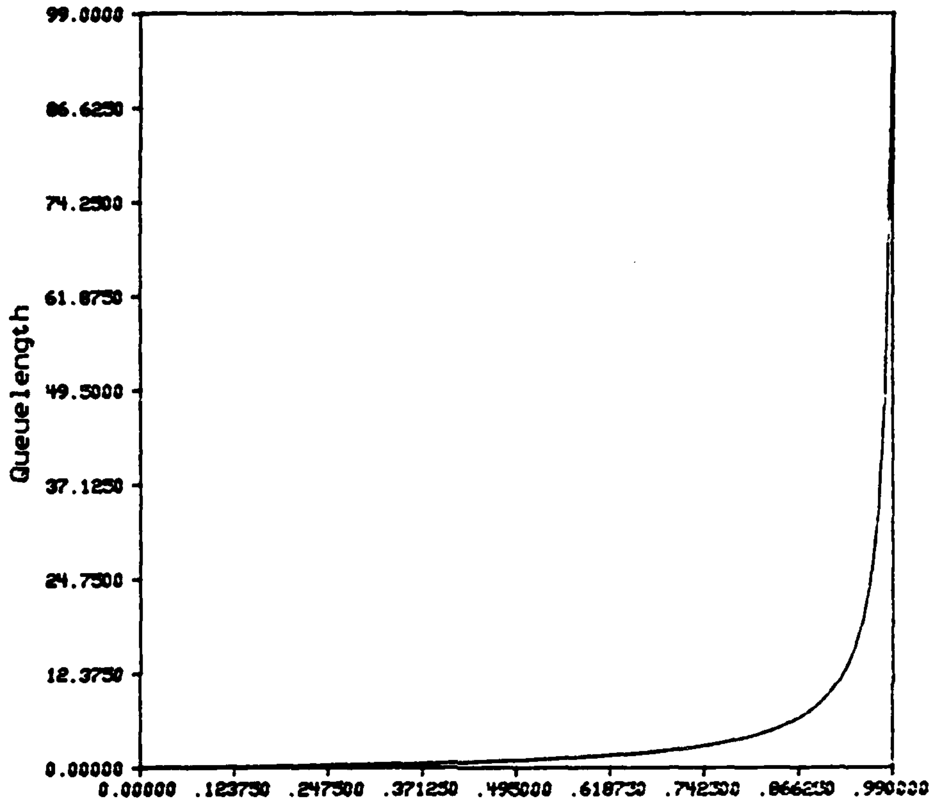


Fig. 5.4.5.4 Queuelength as f(t2) c=1.0

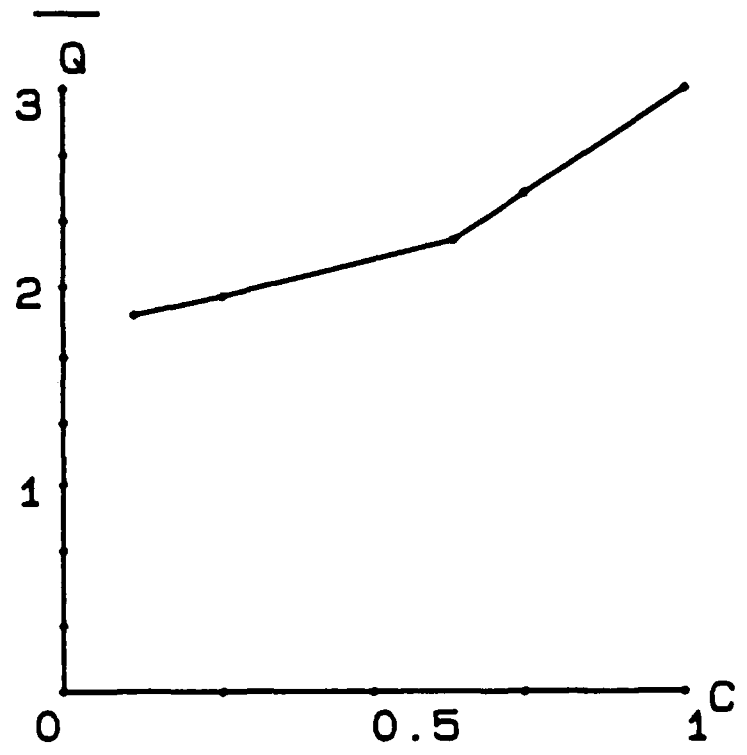
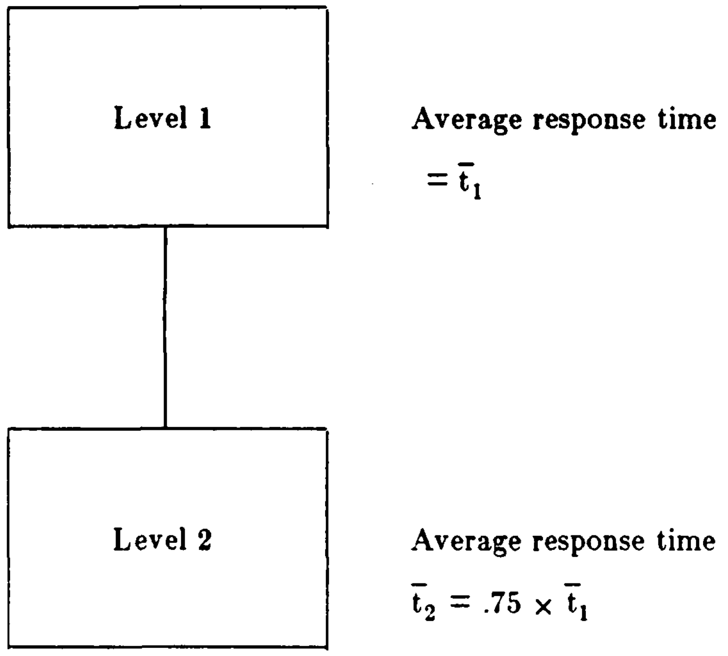


Fig. 5.4.5.5 Q as f(C) [b=0.75]

Table 5.4.5.2.

Asynchronous system statistics (Gaussian).

C	$\bar{t}_2$	$\bar{Q}$	$\bar{W}$	$\overline{SRT}$	$\overline{ST}$
1.00	0.50 t_1	1.00	0.50 t_1	2.00 t_1	$1/t_1$
1.00	0.75 t_1	3.00	2.25 t_1	4.00 t_1	$1/t_1$
0.75	0.50 t_1	0.89	0.45 t_1	1.95 t_1	$1/t_1$
0.75	0.75 t_1	2.51	1.88 t_1	3.63 t_1	$1/t_1$
0.58	0.50 t_1	0.85	0.42 t_1	1.92 t_1	$1/t_1$
0.58	0.55 t_1	0.90	0.50 t_1	2.11 t_1	$1/t_1$
0.58	0.60 t_1	1.20	0.72 t_1	2.32 t_1	$1/t_1$
0.58	0.65 t_1	1.80	0.95 t_1	2.60 t_1	$1/t_1$
0.58	0.70 t_1	2.20	1.25 t_1	2.95 t_1	$1/t_1$
0.58	0.75 t_1	2.25	1.69 t_1	3.44 t_1	$1/t_1$
0.58	0.80 t_1	2.93	2.35 t_1	4.15 t_1	$1/t_1$
0.58	0.90 t_1	6.03	5.67 t_1	7.57 t_1	$1/t_1$
0.50	0.50 t_1	0.80	0.40 t_1	1.90 t_1	$1/t_1$
0.50	0.75 t_1	2.20	1.65 t_1	3.40 t_1	$1/t_1$
0.25	0.55 t_1	0.91	0.50 t_1	2.05 t_1	$1/t_1$
0.25	0.60 t_1	1.07	0.64 t_1	2.24 t_1	$1/t_1$
0.25	0.65 t_1	1.29	0.84 t_1	2.49 t_1	$1/t_1$
0.25	0.70 t_1	1.57	1.10 t_1	2.79 t_1	$1/t_1$
0.25	0.75 t_1	1.95	1.46 t_1	3.21 t_1	$1/t_1$
0.25	0.80 t_1	2.50	2.00 t_1	3.80 t_1	$1/t_1$
0.10	0.50 t_1	0.75	0.38 t_1	1.87 t_1	$1/t_1$
0.10	0.55 t_1	0.88	0.48 t_1	2.03 t_1	$1/t_1$
0.10	0.60 t_1	1.05	0.63 t_1	2.23 t_1	$1/t_1$
0.10	0.65 t_1	1.25	0.82 t_1	2.46 t_1	$1/t_1$
0.10	0.70 t_1	1.52	1.06 t_1	2.76 t_1	$1/t_1$
0.10	0.75 t_1	1.89	1.42 t_1	3.17 t_1	$1/t_1$
0.10	0.80 t_1	2.42	1.94 t_1	3.74 t_1	$1/t_1$



It would seem to be reasonable no queue were needed between these two levels, since on the average the second level completes its processing faster than the first level: however, consider the case where level two requires $1.25\times\bar{t}_1$ to complete two adjacent data sets and the normal $.95\times\bar{t}_1$ to complete the next four data sets, and $.65\times\bar{t}_1$ to complete the final two data sets. The average queuelength is approximately 1. With a wide variation of response times and many data sets, this can cause the expected queuelength to grow.

If the first level completes processing on several consecutive data sets faster than $\bar{t}_1$ then the data sets need to be queued for the second level. As $\bar{t}_2$ approaches $\bar{t}_1$, the likelihood of the first level producing large numbers of jobs faster than the second level can handle them increases.

5.4.6. Analysis of $\overline{Q}$

When considering the expected queue size, the expected wait time in the queue, and the system response time of an asynchronous system, the previous discussion assumed that there would be enough buffer memory to hold all the data sets. The expected queue size is the probabilistic term for the average queue size. Thus, if a buffer memory size equal to the average queue size is used, the probability of overflow is 0.5. At this point, no data sets can be taken. Such an event at level I will cause processing to stop at level I-1 when level I-1 attempts to send its results to level I. Rapidly, this will cause the input queue for level I-1 to fill, halting level I-1. Thus, all levels in the system will process data sets at the rate of level I, which is the slowest level in the system. This effectively slows the asynchronous systems processing rate down to the rate of a synchronous system.

It should be remembered that where an asynchronous system will halt, the synchronous system will issue a signal that it is not ready. Thus, an asynchronous system will be less likely to attempt to stop the stream of input data. Because the asynchronous system queues its jobs, the response time for a particular job can become large; however, this is not taken into account with the synchronous system. Response time only refers to jobs that are in the system, thus, statistically the SRT is biased in favor of the synchronous system.

The probability of halting levels because of queue overflow in subsequent levels can be greatly reduced by allowing an appropriate queue size for the level-level queues. The next question becomes: "What is an appropriate queue size for a level-level queue?" This question can be addressed probabilistically.

From the definition of $\overline{Q}$, P(queuelength $\geq \overline{Q}$) = 0.5, this can be expanded to: P(queuelength $\geq k\,\overline{Q}$) = 0.5^k. Depending on the margin of safety desired, the buffer size can be chosen appropriately. Table 5.4.6.1 shows the probability of the queue overflowing versus the size of the queue. This table shows the expected probability of overflow for a queue that is a multiple of $\overline{Q}$. It does not take account of the processing requirements of the data sets, i.e., how likely is it that level I will complete its processing slower than the rest of the system on this many data sets. Here, the underlying assumption is that level I is as likely to be slow after one job as after 100. Such being the case, this table represents a ceiling on the probability of overflow.

When the memory required for $\overline{Q}$ is large, the cost of overflow protection can be significant, so a tighter limit may be required. If the queue for level I overflows then the throughput of the system will drop to that of a synchronous system. Further, it will have the response time of an asynchronous system with its buffers full for levels 1 - I-1. for a system with l levels, the response time can be calculated as fo.lows:

$$\mathrm{SRT(overflow)} = \sum_{i=1}^{l} \mathrm{EPT}_i + \sum_{i=1}^{I} Q_i \mathrm{EPT}_I + \sum_{i=I+1}^{l} Q_i \mathrm{EPT}_i$$

This can represent a significant amount in the case where a large amount of buffering is allocated (again it should be noted that a synchronous system would halt its input stream). When only k of the previous input buffers fill, the SRT can be shown to be:

$$\mathrm{SRT(k\ levels\ full)} = \sum_{i=1}^{l} \mathrm{EP}_i + \sum_{i=I-k}^{I} Q_i \mathrm{EPT}_I + \sum_{i=I+1}^{l} Q_i \mathrm{EPT}_i$$

The previous tables of asynchronous system values, thus need to be weighted according to the probability of overflow. This is done as follows:

Table 5.4.6.1.

Probability of overflow versus $\overline{Q}$

P(overflow)	$\overline{Q}$ ×
0.50000	1.0
0.25000	2.0
0.12500	3.0
0.06250	4.0
0.03125	5.0
0.15625	6.0
0.00781	7.0
0.00390	8.0
0.00195	9.0
0.00098	10.0

$$\text{SRT(w/overflow)} = (1 - \prod_{i=1}^{l} P_i(\text{overflow}))\overline{\text{SRT}}$$

$$+ \sum_{i=i}^{l} \sum_{j=1}^{i} P_i(\text{overflow affects j levels}) \times (\text{number of jobs queued}) \times t_i(\text{max})$$

The case where the overflow causes level 1 to stop the unit loading the data, lost data sets cannot be counted and cannot be accounted. For calculation, this can be avoided by assuming the level 1 has an arbitrary queue size, thus no jobs are lost, only queued.

The system throughput ($\overline{\text{ST}}$) can be calculated by a weighted summation of the synchronous and asynchronous cases. Define ovf to be the total probability that an overflow will occur. Then 1-ovf will be the probability that an asynchronous system will run asynchronously. For the asynchronous cases presented, this becomes:

$$\text{ovf} = \text{ovf}_1 + \text{ovf}_2$$

(ovf_i) is probability that level i will overflow its input buffer. Using the definition of ovf, the weighted system throughput becomes:

$$\overline{\text{ST}}(\text{with overflow}) = (1 - \text{ovf})\overline{\text{ST}}(\text{asynchronous}) + (\text{ovf})\overline{\text{ST}}(\text{synchronous})$$

Thus, for a system where the response time for the first level is a constant t_1 and the response time for the second level is a Gaussian distribution function, if C=0.75 and b=0.75, and the total probability of overflow is 0.75, $\overline{\text{ST}}$ is:

$$\overline{\text{ST}} = .75 \times \frac{1.0}{t_1} + .25 \times \frac{0.89}{t_1} = \frac{0.97}{t_1}$$

The overflow of a buffer does not have a significant effect on an asynchronous system's throughput. It affects the SRT. A synchronous system (built with similar hardware) would use some form of flow control to inform the device supplying the inbound data to halt. An asynchronous system would have to do the same only when all the internal buffers were filled. For real-*time systems, this is clearly not* desirable because data would be lost. An asynchronous system would be less likely to stop the incoming flow of data than a synchronous system because of the internal queuing.

5.4.7. Double-buffering Versus Triple-buffering -- An Analysis

In general, a synchronous system that is double-buffered will have a faster SRT than a system that is triple-buffered. Both systems will have the same ST. Thus, it is reasonable to question the need for a triple-buffered system. The following discussion will consider this question.

Assume that each level of the proposed system is physically remote from the other levels. The time for a level to write data into the double-buffer would be $\delta t \times dss$, where dss is the size of the data set size. Here, the processor would wait for dss responses from the buffer memory. This would adversely affect the processing speed of the system because the data transfer time is a portion of the processing time. If a triple-buffered system were to be used here, the total data transfer time would have to exceed the maximum processing time of the levels involved before it could have a affect ST. Where levels are not geographically remote, δt is small and has little effect. Thus, triple-buffering is not needed.

A combination of the two buffering strategies is possible where some levels are close and others remote. An example of this can be seen where some levels share a given rack, while others are in another rack. Between levels in a given rack, data transfers are quick, so double-buffering can be applied. Between racks, data transfers may be slow, so triple-buffering may be applied. This technique offers the advantages of a tr'ple-buffered system without unnecessary delays where data does not need to travel a great distance. Further, the throughput of the system is not degraded when the data must travel to a remote location.

Thus, where transmission time between levels is significant, triple-buffering is a useful tool because it overlaps the data transmission time with the data processing times. When transmission time is not a significant problem, the triple-buffered scheme will use extra memory and hardware. Further, use of a triple-buffered system will increase the system response time by the response times of all levels using triple-input-buffers. This can represent a significant increase in system response time over the double-buffered approach.

5.4.8. Synchronous Systems Versus Asynchronous Systems

Where there is no ceiling on the response time of a system, such as in a non-real-time environment, asynchronous systems offer potentially greater throughput than synchronous systems with the same processing hardware. Considering that in a synchronous system, levels that complete their processing sooner than other levels must wait for the slower levels to "catch up." In a real-time environment, synchronous and asynchronous systems built with similar hardware will yield interesting results. If the hardware is built so that the synchronous system can keep up with the incoming data, the asynchronous system will be idle. The response time on the asynchronous system will be slightly faster than on its synchronous counterpart, because the entire synchronous system waits on its slowest level. The entire asynchronous system will not wait on the slowest level (unless the pipeline is full), so the effect of the slowest level is more limited in the asynchronous case.

If two independently designed systems, one synchronous, the other asynchronous, are built to process the same task, There will be little difference in the price. The synchronous system will require more expensive processing hardware, while the asynchronous system will potentially require extra memory. The key issue is that if, from a given database, a synchronous system cannot be built to execute a given task within some time constrains, an asynchronous could be used to up the $\overline{\mathrm{ST}}$ and decrease the $\overline{\mathrm{SRT}}$ by a significant amount. For non-real-time systems, asynchronous systems can be built for less money than the synchronous systems because large amounts of buffer memory are not needed -- thus the asynchronous system would be the better choice. For real-time systems, the asynchronous system has a variable response time, depending on the loading of the pipeline, thus if a variable response time is undesirable, a synchronous system should be used.

5.5. System Simulation -- Results

To either prove or disprove the theoretical results presented in the previous sections, a simulator (shown in Appendix 1, was developed. This section will present and analyze the results of the simulation.

The numbers presented in Table 5.5.1 represent the simulated performance of a two level system, where the response time of the first level is fixed and the response time of the second level is a uniform random variable. Both synchronous and asynchronous statistics are shown. When the synchronous statistics are compared with the statistics shown in Table 5.4.3.2, the theory predicts the the actual results with a maximum error of 0.03. "b (actual)" is the actual ratio of $\bar{t}_2$ to t_1. Due to some inconsistencies in the random number generation, the expected ratio, as defined in the second line of the results, is not the actual ratio. Results for the case when level 2 is a Gaussian random variable are shown in Table 5.5.2. When compared with the results shown in Table 5.4.3.1 (C for this table is approximately 0.25), the simulated results differ from the theoretical results by no more than 1 percent. Thus, for the simulated data sets, the theory presented is an accurate representation or the actual results.

When comparing the results for the asynchronous system in Table 5.5.2 to the proposed results shown in Table 5.4.4.1, it is again necessary to take "b (actual)" into account. For uniform data there are some interesting results that arise from this comparison. Taking the ratios (based on the actual value of b) of Q_{max} and $\overline{Q}$ to the $\overline{Q}$ predicted in Table 5.4.4.1, the first ratio falls in the range 1.45 ± 0.155. The second ratio falls in the range: 0.14 ± 0.05. Comparison of these results shows that the maximum queue size is approximately ten times the average queue size (based on 200000 test runs).

Table 5.5.1

Uniform system response times (200000 samples).

Parameters		Synchronous System			Asynchronous System		
$\frac{t2}{t1}$	b (act)	SRT(DB)	SRT(TB)	ST(Xt1)	Q_{max}	$\overline{Q}$	$\frac{\overline{SRT}}{t1}$
1.00	1.06	2.56	5.12	.78	11596	5764	6113
0.95	1.02	2.48	4.96	.81	2230	976	988
0.90	0.96	2.41	4.82	.83	39	3.49	5.04
0.85	0.90	2.34	4.68	.86	16	1.42	2.99
0.80	0.84	2.26	4.52	.88	7	0.73	2.33
0.75	0.80	2.20	4.40	.91	6	0.54	2.08
0.70	0.74	2.14	4.28	.93	4	0.29	1.89
0.65	0.70	2.08	4.16	.96	2	0.17	1.76

Table 5.5.2.

Gaussian system response times.

Parameters		Synchronous System			Asynchronous System		
Samples	$\frac{t2}{t1}$	SRT(DB)	SRT(TB)	ST(Xt1)	Q_{max}	$\overline{Q}$	$\frac{\overline{SRT}}{t1}$
200000	1.00	2.10	4.20	0.95	520	353.73	255.71
200000	0.95	2.08	4.16	0.96	201	36.45	38.03
200000	0.90	2.03	4.06	0.99	1	0.15	1.92

Further, that with 89% accuracy, 1.45 times the predicted queue size yields the maximum queue size. Thus, for response times that can be modeled as a uniform distribution, 1.45 times the queue size predicted by the application of the Pollaczek-Khinchine is an accurate model of the maximum queue needed for the inter-level buffer of a system where a level with a fixed execution time is feeding data to a level with an execution time that can be modeled by a uniform random variable.

If the execution time of the second level in a system can be modeled by a Gaussian random variable, the Pollaczek-Khinchine rule does not accurately predict the expected queue size of the system when the average response time of the second level is more than .95 times the response time of the first level. This is shown by comparing the results in Table 5.5.2 with those in Table 5.4.4.1.

The statistics presented Table 5.4.5.1 for a synchronous system are 30 percent lower than results achieved through simulation shown in Table 5.5.3. For the simulation, $C_1 = 0.570$ and $C_2 = 0.577$. The results of the simulation show that a synchronous system behave slightly worse than the theory predicts. Finally, the simulation results presented in Table 5.5.4 ($C=0.58$) show that the expected theoretical queue sizes presented in Table 5.4.5.2 are 30 percent greater than the actual average queue size.

The results of the simulation show that synchronous systems behave worse than the theory predicts and that asynchronous systems behave better than the theory predict. To achieve the same throughput, synchronous systems require hardware that is twice as fast as asynchronous systems performing the same task. Further, for a two level system, when the expected response time of the second level is 75 percent of the first level, the asynchronous system will

Table 5.5.3

Synchronous system response times when the response times of both level 1 and level 2 can be modeled by a uniform random variable.

Samples	$\frac{t2}{t1}$	SRT(DB)	SRT(TB)	ST(/t1)
1E+06	1.00	2.71	5.42	0.74
1E+06	0.95	2.64	5.28	0.76
1E+06	0.90	2.60	5.20	0.77
1E+06	0.85	2.53	5.06	0.79
1E+06	0.80	2.46	4.92	0.81
1E+06	0.75	2.42	4.84	0.83
1E+06	0.70	2.37	5.74	0.85
1E+06	0.65	2.32	5.64	0.86

Table 5.5.4

Performance statistics for an asynchronous system whose levels can be modeled by uniform random variables.

Samples		$\frac{t2}{t1}$	Q_{max}	$\overline{Q}$
2E+05	1.00	177	62.8	1.00
2E+05	0.95	39	6.01	1.00
2E+05	0.90	29	2.97	1.00
2E+05	0.85	20	1.76	1.00
2E+05	0.80	13	1.22	1.00
2E+05	0.75	12	0.88	1.00
2E+05	0.70	10	.67	1.00

generate results more quickly. This is the worst case and occurs when the distribution of the second level can be modeled by a uniform distribution. When the second level can be modeled by a Gaussian distribution, the results are more pronounced (the cross-over point occurs when the average response time of the second level is 85 percent of the first level).

5.6. Conclusions

Theory and background information was presented to relate the performance of both asynchronous and synchronous systems. The theory predicted that synchronous systems would have 84 percent of the throughput of asynchronous systems. Through simulation, this figure was shown to be up to 16% high. Application of the theory to asynchronous systems showed that for certain hardware configurations, asynchronous systems had both greater throughput and lower response time. The key disadvantage to asynchronous systems is clearly that the response time can vary by a large amount. For real-time systems, this fact is significant enough that asynchronous systems may not be feasible.

LIST OF REFERENCES

LIST OF REFERENCES

[AhH76] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1976.

[All82] G.R. Allen, "A reconfigurable architecture for arrays of microprogrammable processors," in *Special Computer Architectures for Pattern Processing,* K.S. Fu and T. Ichikawa, eds., CRC Press, Boca Raton, FL, 1982, pp. 65-86.

[Amd82] Advanced Micro Devices Inc., *Am9511A Product Specification Sheet,* AMD Inc., Sunnyvale, CA, Mar. 1982.

[ArB82] R.G. Arnold, R.O. Berg, and J.W. Thomas, "A modular approach to supersystems," *IEEE Trans. Comput.*, Vol. C-31, May 1982, pp. 385-389.

[Bar68] G.H. Barnes, et al., "The ILLIAC IV computer," *IEEE Trans. Comput.,* Vol. C-17, No. 8, Aug. 1968, pp 746-757.

[Bat74] K.E. Batcher, "STARAN parallel processor system hardware," *Proc. 1974 National Computer Conf.*, 1974, pp. 405-410.

[Bat76] K.E. Batcher, "The flip network in STARAN," *1976 Int'l. Conf. Parallel Processing,* Aug. 1976, pp. 65-71.

[Bat77a] K.E. Batcher, "The flip network in STARAN," *1976 Int'l. Conf. Parallel Processing,* Aug. 1976, pp. 65-71.

[Bat77b] K.E. Batcher, "STARAN series E," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 144-153.

[Bat80] K.E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, Vol. C-29, Sept. 1980, pp. 836-840.

[Bat82] K. E. Batcher, "Bit-serial parallel processing systems," *IEEE Trans. Comput.*, Vol. C-31, May 1982, pp. 377-384.

[BeN71] C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

[Ber66] A.J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. Comput.*, Vol. EC-15, Oct. 1966, pp. 757-763.

[Bou72] W.J. Bouknight et al., "The ILLIAC IV system," *Proc. IEEE*, Vol. 60, Apr. 1972, pp. 369-388.

[CDC77a] Control Data Corp., *Cyber-Ikon Image Processing System Concepts.*, Digital Systems Division, Control Data Corp., Minneapolis, MN, Jan. 1977.

[CDC77b] Control Data Corp., *Cyber-Ikon Flexible Processor Programming Textbook*, Digital Systems Division, Control Data Corp., Minneapolis, MN, Nov. 1977.

[Che80] T.C. Chen, "Overlap and Pipeline Processing," in *Introduction to Computer Architecture*, 2nd edition, H.S. Stone, ed., Science Research Associates, Chicago, IL, 1980, pp. 363-425.

[Cin75] E. Cinlar, *Introduction to Stochastic Processes*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1975.

[CoM67] R.W. Conway, W.L. Maxwell, and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[Dem83] G. DeMuth, "A distributed signal processor incorporating VLSI and high order language programming," *1983 Int'l. Conf. Acoustics, Speech, and Signal Processors*, Apr. 1983, pp. 439-442.

[Duf82] M.J.B. Duff, "CLIP4," in *Special Computer Architectures for Pattern Processing*, K.S. Fu and T. Ichikawa, eds., CRC Press, Boca Raton, FL, 1982, pp. 65-86.

[Duf83] M.J.B. Duff, ed., *Computing Structures for Image Processing*, Academic Press, New York, NY, 1983.

[DuL81] M.J.B. Duff and S. Levialdi, eds., *Languages and Architectures for Image Processing*, Academic Press, New York, NY, 1981.

[DuW73] M.J.B. Duff, D.M. Watson, T.J. Fountain, and G.K. Shaw, "A cellular logic array for image processing," *Pattern Recognition*, May 1983, pp. 229-247.

[FeF79] J.D. Feldman and L.C. Fulmer, "RADCAP -- an operation parallel processing facility," *Proc. of 1979 National Computer Conf.*, 1979, pp. 7-15.

[Fly66] M.J. Flynn, "Very high speed computing systems," *Proc. IEEE*, Vol. 54, Dec. 1966, pp. 1901-1909.

[Fou81] T.J. Fountain, "CLIP4: a progress report," in *Languages and Architectures for Image Processing*, M.J.B. Duff and S. Levialdi, eds., Academic Press, New York, NY, 1981.

[FuI82] K.S. Fu and T. Ichikawa, eds., *Special Computer Architectures for Pattern Processing*, CRC Press, Boca Raton, FL, 1982.

[Ful75] S.H. Fuller, in *Introduction to Computer Architecture*, H.S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1975, pp. 474-541.

[Ger83] F.A. Gerritson, "A comparison of the CLIP4, DAP, and and MPP processor array implementations," in *Computing Structures for Image Processing*, M.J.B. Duff, ed., Academic Press, New York, NY, 1983, pp. 15-30.

[Gil83] W.K. Giloi, "Towards a taxonomy of computer architecture based on the machine data type view," *10th Annual Int'l. Symp. Comput. Arch.*, May 1983, pp. 6-15.

[Gon78] M.J. Gonzalez, "Quantitative evaluations of distributed computing systems," *Rocky Mountain Symp. on Microcomputers: Systems, Software, and Arch.*, Aug. 1978, pp. 125-130.

[Gud81] K. Gudmundsson, "Overview of the high level language for PICAP," *Languages and Architectures for Image Processing*, M.J.B. Duff and S. Levialdi, eds., Academic Press, New York, NY, 1981, pp. 147-156.

[Han77] W. Handler, "The impact of classification schemes on computer architecture," *1977 Int'l. Conf. Parallel Processing*, Aug. 1977, pp. 7-15.

[Han81] W. Handler, "Standards, classification, and taxonomy; experiences with ECS," *IFIP Workshop on Taxonomy in Comput. Arch.*, June 1981, pp. 39-75.

[Har68] J.F. Hart, et al, *Computer Approximations*, John Wiley and Sons, Inc., New York, NY, 1968.

[HaS73] R.M. Haralick, K. Shanmugam, and I. Dinstein, "Textural features for image classification," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-3, Nov. 1973, pp. 610-621.

[Hay78] J. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, NY, 1978.

[HoJ81] R.W. Hockney and C.R. Jesshope, *Parallel Computers: Architectures, Programming, and Algorithms*, Adam Hilger Ltd, Bristol, 1981.

[HoS82a] R.M. Hord and D.K. Stevenson, "The ILLIAC IV architecture and its suitability for image processing," in *Special Computer Architectures for Pattern Processing*, K.S. Fu and T. Ichikawa, eds., CRC Press, Boca Raton, FL, 1982, pp. 103-126.

[HoS82b] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, 2nd edition, Comput. Science Press, Rockville, MD, 1982.

[HuL82] R.W. Hueft and W.D. Little, "Improved time and parallel processor bounds for FORTRAN like loops," *IEEE Trans. Comput.*, Vol. C-31, Jan. 1982, pp. 78-81.

[Hun81] D.J. Hunt, "The ICL DAP and its application to image processing," *Languages and Architectures for Image Processing*, M.J.B. Duff and S. Levialdi, eds., Academic Press, New York, NY, 1981, pp. 274-282.

[HwB84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.

[KaK78] S.I. Kartashev and S.P. Kartashev, "Dynamic architectures: problems and solutions," *Computer*, Vol. 11, July 1978, pp. 26-42.

[KeL76] R.L. Kettig and D.A. Landgrebe, "Classification of multispectral image data by extraction and classification of homogeneous objects," *IEEE Trans. Geoscience Electronics*, Vol. GE-14, Jan. 1976, pp. 19-26.

[KrD82] B. Kruse, P.E. Danialsson, and B. Gudmundson, "From PICAP I to PICAP II," *Special Computer Architectures for Image Processing,* K.S. Fu and T. Ichikawa, eds., CRC Press Boca Raton, FL, 1982, pp. 65-86.

[KrG82] B. Kruse, K. Gudmundsson, and D. Antonsson, "PICAP and relational neighborhood processing in FIP," in *Multicomputers and Image Processing Algorithms and Programs,* K. Preston, Jr. and L. Uhr, eds., Academic Press, New York, NY, 1982, pp. 31-47.

[Kuc78] D.J. Kuck, *The Structure of Computers and Computations, Vol. I,* John Wiley and Sons, Inc., New York, NY, 1978.

[KuM72] D.J. Kuck, Y. Muraoka, and S.C. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed up," *IEEE Trans. Comput.*, Vol. C-21, Dec. 1972, pp. 1292-1309.

[LaR81] L.F. Lamel, L.R. Rabiner, A.E. Rosenburg, and J.G. Wilpon, "An improved endpoint detector for isolated word recognition," *IEEE Trans. Acoustics, Speech, and Signal Processing,* Vol. ASSP-11, Aug. 1981, pp. 777-785.

[LiT77] G.J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 165-174.

[LoM80] R.M. Lougheed and D.L. McCubbrey, "The cytocomputer: a practical pipelined image processor," *7th Annual Symp. on Comput. Architecture,* May 1980, pp. 271-277.

[MaG74] J.D. Markel and A.H. Gray, Jr., "Fixed point truncation arithmetic implementation of a linear prediction autocorrelation vocoder," *IEEE Trans. Acoustics, Speech, and Signal Processing,* Vol. ASSP-22, Aug. 1974, pp. 273-282.

[McS82] R.J. McMillen and H.J. Siegel, "A comparison of cube type and data manipulator type networks," *3rd Int'l. Conf. on Distributed Computing Systems,* Oct. 1982, pp. 614-621.

[Met78] J.F. Metzger, ed., *1978 IC Master,* United Technical Publications, Garden City, NY, 1978.

[Mot80] Motorola, *MC68000, 16-Bit Microprocessor User's Manual,* second edition, M68000UM(AD2), Jan. 1980.

[Mot81] Motorola, *68343 fast floating point source/object for MC68000,* M68KFFP specification sheet, Nov. 1981

[NAS72] National Aeronautics and Space Administration, *Earth Resources Technology Satellite: Data Users Handbook,* Goddard Flight Center, Greenbelt, MD., 1972.

[Oga70] K. Ogata, *Modern Control Engineering,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.

[Pap65] A. Papoulis, *Probability, Random Variables, and Stochastic Processes,* McGraw-Hill Book Co., New York, NY, 1965.

[Pot82a] J.L. Potter, "MPP architecture and programming," in *Multicomputers and Image Processing Algorithms and Programs,* K. Preston, Jr., and L. Uhr, eds., Academic Press, New York, NY, 1982, pp. 275-290.

[Pot82b] J.L. Potter, "Pattern processing on STARAN," in *Special Computer Architectures for Pattern Processing,* K.S. Fu and T. Ichikawa, eds., CRC Press, Boca Raton, FL, 1982, pp. 87-102.

[PrD79] K. Preston, Jr., M.J.B. Duff, S. Levialdi, P.E. Norgren, and J.I. Toriwaki, "Basics of cellular logic with some applications in medical image processing," *Proc. IEEE,* Vol. 76, May 1979, pp. 826-855.

[Pre77] F.P. Preparata, "Parallelism in sorting," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 202-206.

[PrU82] K. Preston, Jr., and L. Uhr, eds., *Multicomputers and Image Processing Algorithms and Programs,* Academic Press, New York, NY, 1982.

[RaG69] C.V. Ramamoorthy and M.J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," *1969 Joint Comput. Conf.,* Aug. 1969, pp. 1-15.

[RaL79] L.R. Rabiner, S.E. Levinson, A.E. Rosenburg, and J.G. Wilpon, "Speaker-independent recognition of isolated words using clustering techniques," *IEEE Trans. Acoustics, Speech, and Signal Processing,* Vol. ASSP-27, Aug. 1979, pp. 336-349.

[RaS78] L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Red79] S.F. Reddaway, "The DAP approach," *Infotech State of the Art Report on Supercomputers, Vol. 2,* ICL Research and Advanced Development Center, Stevenage, England, 1979, pp. 311-329.

[ReM83] J.E. Requa and J.R. McGraw, "The piecewise data flow architecture: architectural concepts," *IEEE Trans. Comput.*, Vol. C-32, May 1983, pp. 425-438.

[SaR75] M.R. Sambur and L.R. Rabiner, "A speaker-independent digit-recognition system," *Bell System Technical Journal,* Vol. 54, Aug. 1975, pp. 81-102.

[Sie85] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, Lexington, MA, 1985.

[SiM81a] H.J. Siegel and R.J. McMillen "Using the augmented data manipulator network in PASM," *Computer,* Vol. 14, Feb. 1981, pp. 25-33.

[SiM81b] H.J. Siegel and R.J. McMillen, "The multistage cube: a versatile interconnection network," *Computer,* Vol. 14, Dec. 1981, pp. 65-76.

[SiS80] H.J. Siegel, P.H. Swain, and B.W. Smith, "Parallel processing implementations of a contextual classifier for remote sensing data," *1980 Machine Processing of Remotely Sensed Data Symp.*, June 1980, pp. 19-28.

[SiS81] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller Jr., H.E. Smalley Jr., and S.D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, Vol. C-30, Dec. 1981, pp. 934-947.

[SiS82a] L.J. Siegel, H.J. Siegel, and A.E. Feather, "Parallel processing approaches to image correlation," *IEEE Trans. Comput.*, Vol. C-34, Mar. 1982, pp. 208-217.

[SiS82b] L.J. Siegel, H.J. Siegel, and P.H. Swain, "Performance measures for evaluating algorithms for SIMD machines," *IEEE Trans. Software Eng.*, Vol. SE-8, July 1982, pp. 319-331.

[SiS82c] H.J. Siegel, P.H. Swain, and B.W. Smith, "Remote sensing on PASM and CDC flexible processors," in *Multicomputers and Image Processing Algorithms and Programs,* K. Preston, Jr. and L. Uhr, eds., Academic Press, New York, NY, 1982, pp. 331-342.

[SiS83] L.J. Siegel, H.J. Siegel, P.H. Swain, G.B. Adams III, W.E. Kuhn III, R.J. McMillen, T.A. Rice, K.D. Smith, and D.L. Tuomenoksa, "Distributed computing for signal processing: modeling of asynchronous parallel computation 1983 progress report," *Technical Report TR-EE 83-11,* E.E. School, Purdue University, West Lafayette, IN 47907, Mar. 1983.

[SiS84] H.J. Siegel, T. Schwederski, N.J. Davis IV, and J.T. Kuehn, "PASM: a reconfigurable parallel system for image processing," *ACM SIGARCH Computer Architecture News,* Vol. 12, Sept. 1984, pp. 7-19.

[SmS81] B.W. Smith, H.J. Siegel, and P.H. Swain," Contextual classification on a CDC Flexible Processor system," *1981 Machine Processing of Remotely Sensed Data Symp.,* June 1981, pp. 283-288.

[SmS82] B.W. Smith, H.J. Siegel, and P.H. Swain, "Parallel processing concepts for remote sensing applications," *1982 Machine Processing of Remotely Sensed Data Symp.,* June 1982, pp. 520-526.

[Ste80] S.R. Sternberg, "Cellular computers and biomedical image processing," *U.S. - France Seminar on Biomedical Image Processing,* Grenoble, France, May 1980, pp. 1-26.

[Sto73] H.S. Stone, "Problems of parallel computation," in *Complexity of Sequential and Parallel Numerical Algorithms,* J.F. Traub, ed., Academic Press, New York, NY, 1973.

[Sto80] H.S. Stone, "Parallel computers," in *Introduction to Computer Architecture,* 2nd edition, H.S. Stone, ed., Science Research Associates, Chicago, IL, 1980, pp. 363-425.

[SwD78] P.H. Swain and S. Davis, eds., *Remote Sensing: The Quantitative Approach,* McGraw-Hill Inc., New York, NY, 1978.

[SwS80] P.H. Swain, H.J. Siegel, and B.W. Smith, "Contextual classification of multispectral remote sensing data using a multiprocessor system," *IEEE Trans. Geoscience and Remote Sensing,* Vol. GE-18, Apr. 1980, pp. 197-203.

[SwV81] P.H. Swain, S.B. Vardeman, and J.C. Tilton, "Contextual classification of multispectral image data," *Pattern Recognition,* Vol. 13, Mar. 1981, pp. 429-441.

[Thu76] K.J. Thurber, *Large Scale Computer Architecture, Parallel and Associative Processors,* Hayden Book Company, Rochelle Park, NJ, 1976.

[TiS81] J.C. Tilton, P.H. Swain, and S.B. Vardeman, "Contextual classification of multispectral image data: an unbiased estimator for context distribution," *1981 Machine Processing of Remotely Sensed Data Symp.,* June 1981, pp. 304-313.

[TuA83] D.L Tuomenoksa, G.B. Adams, H.J. Siegel, and O.R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *IEEE Comput. Society Conf. on Comput. Vision and Pattern Recognition,* June 1983, pp. 336-344.

[Vic79] C.R. Vick, *A Dynamically Reconfigurable Distributed Computing System,* Ph.D. Dissertation, E.E. Dept., University of Alabama, Auburn, AL, 1979.

[WeD76] J.S. Weszka, C.R. Dyer, and A. Rosenfeld, "A comparative study of texture measures and terrain classification," *IEEE Trans. Systems, Man, and Cybernetics,* Vol. SMC-6, Apr. 1976, pp. 259-285.

[WeS71] J.R. Welch and K.G. Salter, "A context algorithm for pattern recognition and image interpretation," *IEEE Trans. Systems, Man, and Cybernetics,* Vol. SMC-1, Jan. 1971, pp. 24-30.

[WoC84] C.M. Woodside and D.W. Craig, "Function allocation in a tightly coupled signal processing system," *4th Int'l. Conf. Distributed Computing Systems,* May 1984, pp. 118-125.

[YoS82] M.A. Yoder and L.J. Siegel, "Dynamic time warping algorithms for SIMD machines and VLSI processor arrays," *1982 IEEE Int'l. Conf. Acoustics, Speech, and Signal Processing,* May 1982, pp. 1274-1277.

[YoS84] M.A. Yoder and L.J. Siegel, "Parallel algorithms for isolated and connected word recognition," *Technical Report TR-EE-84-47,* E.E. School, Purdue University, West Lafayette, IN, Dec. 1984.

APPENDIX

SIMULATOR LISTINGS

System Simulator Level 1 Response Time Fixed and Level 2 Response Time Uniform Random Variable

```
#include<stdio.h>

/**********************************************************************/
/*                                                                    */
/* This simulator was written for a Gould Powernode 9080 running UNIX */
/* 4.2 BSD.  It simulates the execution of both a synchronous and an */
/* asynchronous two level system, keeping track of the level-to-level */
/* buffersizes and the response times of the second level.  For this */
/* simulator, it is assumed that the distribution describing the res- */
/* time of the second level is the uniform distribution function, and */
/* that the first level has a constant response time.  It is run on a */
/* total of 200,000 data sets  per  simulation  statistic,  requiring */
/* more than  1.0Mb  storage  and  more  than  2  hours  to  complete */
/*                                                                    */
/**********************************************************************/

#define EVENTS 200000            /* Events per simulation            */
#define FEVENTS 200000.0         /* floating point representation   */
FILE *f1;                         /* file pointers                    */
int density[16];                  /* for density function             */
                                  /* Gould Firebreather 9080 Compiler */
                                  /* needs the NOBASE because         */
                                  /* it needs to use a different      */
                                  /* addressing mode to handle large  */
/* NOBASE */                      /* arrays.                          */
float timelist[100+EVENTS];      /* list of event times              */
int pqueuesize,mqueuesize;        /* present and maximum queuesizes   */
int rqueuesize;                   /* total jobs stored in queues      */
float idletime;                  /* idletime for second level        */
float r1sptime;                  /* response time for first  level   */
float resptime;                  /* response time for second level   */
float rrsptime;                  /* cumulative time for second level */
float sysresptime;               /* system response time             */
float synchresptime;             /* synchronous system response time */

float pcnt[] = {1.00,            /* ratio of:                        */
                0.95, 0.90,      /*        response time of level 2  */
                0.85, 0.80,      /*                                  */
                0.75, 0.70,      /*        response time of level 1  */
                0.65, 0.60,
                0.55, 0.50,
                0.00 };
main()
{       float ktime;             /* time keeper                      */
        float rtime;             /* response time keeper             */
        int eventtime,            /* event duration                   */
        int i;                    /* event counter                    */
        int j;                    /* event counter                    */

        f1=fopen("simulation.results","w"); /* open files            */

        srandom(getpid());       /* initialize random number generator */
        if(f1==NULL)
                exit();
        for(j=0,pcnt[j]>0.0;j++)
        {       pqueuesize=0;    /* initialize all statistical keepers */
                mqueuesize=0;
                rqueuesize=0;
                idletime= 0.0;
```

```
r1sptime= 0.0;
resptime= 0.0;
rrsptime= 0.0;
sysresptime= 0.0;
for(i=0;i<16;i++)
        density[i]=0;

ktime=8.0/16.0;

for(i=0;i<100+EVENTS;i++)  /* generate random events  */
{         timelist[i] = ktime;
          ktime+=0.5;
}
r1sptime=ktime;

synchresptime=0.0;
ktime=0.0;
rrsptime=0.0;
for(i=0;i<EVENTS;i++)
{       eventtime = 017 & random(); /* random number  */
                                    /* 0 and 15       */

        density[eventtime]++;
        eventtime+=1;
        rtime = pcnt[j]/16.0*(float) eventtime;
        rrsptime+=rtime;
        if(rtime<(8.0/16.0))
                synchresptime+=8.0/16.0;
        else
                synchresptime+=rtime;

        statqueue(i+1,ktime,rtime);    /* queuesize   */

        if(ktime<timelist[i])
        {       idletime+=timelist[i]-ktime;
                ktime=timelist[i]+rtime;
        }
        else
                ktime+=rtime;

        resptime+=ktime-timelist[i];
        if(i>1)
        {       sysresptime+=
                        rtime+timelist[i]-timelist[i-1];
        }
}
fprintf(f1,"\f\n\n\n              System simulation\n\n");
fprintf(f1,"Sample set size:                            %6d\n",
        EVENTS);
fprintf(f1,"Processing time of level 2 (times level 1) %6.2f\n",
        pcnt[j]);
fprintf(f1,"Average processing time (level 1)          %6.2f\n",
        r1sptime/(float)(100+EVENTS));
fprintf(f1,"Average processing time (level 2)          %6.2f\n",
        rrsptime/FEVENTS);
fprintf(f1,"\n        Asynchronous system statistics\n");
fprintf(f1,"Average size of level-level queue          %6.2f\n",
        (float) rqueuesize/FEVENTS);
fprintf(f1,"Maximum size of level-level queue          %6d\n",
        mqueuesize);
fprintf(f1,"Average response time   (level 2):         %6.2f\n",
        resptime/FEVENTS);
fprintf(f1,"Average system response time (times t1)    %6.2f\n",
        ((r1sptime/(FEVENTS+100.0))+(resptime/FEVENTS))/
        (r1sptime/(float)(100+EVENTS)));
fprintf(f1,"Approximate Percent Idle time (level 2)    %6.2f\n",
        100.0*idletime/timelist[EVENTS]);
fprintf(f1,"\n        Synchronous system statistics\n");
fprintf(f1,"Synchronous SRT(DB) (x t1):                %6.2f\n",
        (2*synchresptime/FEVENTS)/
        (r1sptime/(float)(100+EVENTS)));
fprintf(f1,"Synchronous SRT(TB) (x t1)                 %6.2f\n",
        (3*synchresptime/FEVENTS)/
        (r1sptime/(float)(100+EVENTS)));
fprintf(f1,"Synchronous ST(/ t1)                       %6.2f\n",
        1.0/((synchresptime/FEVENTS)/
        (r1sptime/(float)(100+EVENTS))));
```

```
                fprintf(f1,"\n\n   Distribution of level 2 response times:\n");
                for(i=0;i<8;i++)
                        fprintf(f1,"%10d - %6d        %2d - %6d\n",
                                i+1,density[i],i+9,density[i+8]);
                fflush(f1);
        }
}
statqueue(pos,prestime,proctime)
int pos;
float prestime, proctime;
{       int j;

        pqueuesize=0;

        for(j=pos;((j<(100+EVENTS))&&(timelist[j]<(prestime+proctime)));j++)
                pqueuesize++;

        if(pqueuesize>mqueuesize)
                mqueuesize=pqueuesize;

        rqueuesize+=pqueuesize;

}
```

System Simulator
Level 1 Fixed and Level 2 Gaussian Random Variable

```
#include<stdio h>

/*************************************************************/
/*                                                           */
/* This simulator was written for a Gould Powernode 9080 running UNIX */
/* 4 2 BSD  It simulates the execution of both a synchronous and an */
/* asynchronous two level system, keeping track of the level-to-level */
/* buffersizes and the response times of the second level.  For this */
/* simulator, it is assumed that the distribution describing the res- */
/* time of the second level is the uniform distribution function, and */
/* that the first level has a constant response time.  It is run on a */
/* total of 200,000 data sets  per  simulation  statistic,  requiring */
/* more than 1. 0Mb  storage  and  more  than  2  hours  to  complete */
/*                                                           */
/*************************************************************/

#define EVENTS 200000              /* Events per simulation            */
#define FEVENTS 200000.0           /* floating point representation    */
FILE *f1;                          /* file pointers                    */
int density[16];                   /* for density function             */
                                   /* Gould Firebreather 9080 Compiler */
                                   /* needs the NOBASE because         */
                                   /* it needs to use a different      */
                                   /* addressing mode to handle large  */
/* NOBASE */                       /* arrays.                          */
float timelist[100+EVENTS];        /* list of event times              */
int pqueuesize,mqueuesize;         /* present and maximum queuesizes   */
int rqueuesize;                    /* total jobs stored in queues      */
float idletime;                    /* idletime for second level        */
float r1sptime;                    /* response time for first  level   */
float resptime;                    /* response time for second level   */
float rrsptime;                    /* cumulative time for second level */
float sysresptime;                 /* system response time             */
float synchresptime;               /* synchronous system response time */

float pcnt[] = {1.00,              /* ratio of:________________________ */
                0.95, 0.90,        /*          response time of level 2 */
                0.85, 0.80,        /*          ------------------------ */
                0.75, 0.70,        /*          response time of level 1 */
                0.65, 0.60,
                0.55, 0.50,
                0.00 };
main()
{       float ktime;               /* time keeper                      */
        float rtime;               /* response time keeper             */
        int eventtime;             /* event duration                   */
        int i;                     /* event counter                    */
        int j;                     /* event counter                    */

        f1=fopen("simulation.results.gauss","w"); /* open files             */

        srandom(getpid());         /* initialize random number generator */
        if(f1==NULL)
                exit();
        for(j=0;pcnt[j]>0 0;j++)
        {       pqueuesize=0;      /* initialize all statistical keepers */
                mqueuesize=0;
                rqueuesize=0;
                idletime= 0 0;
                r1sptime= 0 0;
                resptime= 0 0;
                rrsptime= 0.0;
                sysresptime= 0 0;
                for(i=0;i<16;i++)
                        density[i]=0;

                ktime=8.0/16.0;

                for(i=0;i<100+EVENTS;i++)   /* generate random events  */
                {       timelist[i] = ktime;
                        ktime+=0.5;
                }
                r1sptime=ktime;
                synchresptime=0.0;
                ktime=0.0;
                rrsptime=0.0;
                for(i=0;i<EVENTS;i++)
                {       eventtime = gauss();          /* random number   */
                                                      /* 0 and 15        */

                        density[eventtime]++;
                        eventtime+=1;
                        rtime = pcnt[j]/16.0*(float) eventtime;
                        rrsptime+=rtime;
                        if(rtime<(8.0/16.0))
                                synchresptime+=8.0/16.0;
                        else
                                synchresptime+=rtime;

                        statqueue(i+1,ktime,rtime);     /* queuesize    */

                        if(ktime<timelist[i])
                        {       idletime+=timelist[i]-ktime;
                                ktime=timelist[i]+rtime;
                        }
                        else
                                ktime+=rtime;

                        resptime+=ktime-timelist[i];
                        if(i>1)
                        {       sysresptime+=rtime+timelist[i]-timelist[i-1];
                        }
                }
                fprintf(f1,"\f\n\n\n            System simulation\n\n");
                fprintf(f1,"Sample set size:                              %6d\n",
                        EVENTS);
                fprintf(f1,"Processing time of level 2 (times level 1):%6.2f\n",
                        pcnt[j]);
                fprintf(f1,"Average processing time (level 1):            %6.2f\n",
                        r1sptime/(float)(100+EVENTS));
                fprintf(f1,"Average processing time (level 2):            %6.2f\n",
                        rrsptime/FEVENTS);
                fprintf(f1,"\n       Asynchronous system statistics\n");
                fprintf(f1,"Average size of level-level queue:            %6.2f\n",
                        (float) rqueuesize/FEVENTS);
                fprintf(f1,"Maximum size of level-level queue:            %6d\n",
                        mqueuesize);
                fprintf(f1,"Average response time   (level 2):            %6.2f\n",
                        resptime/FEVENTS);
                fprintf(f1,"Average system response time (times t1):     %6.2f\n",
                        ((r1sptime/(FEVENTS+100.0))+(resptime/FEVENTS))/
                        (r1sptime/(float)(100+EVENTS)));
                fprintf(f1,"Approximate Percent Idle time (level 2)     %6.2f\n",
                        100.0*idletime/timelist[EVENTS]);
                fprintf(f1,"\n        Synchronous system statistics\n");
                fprintf(f1,"Synchronous SRT(DB) (x t1):                   %6.2f\n",
                        (2*synchresptime/FEVENTS)/
                        (r1sptime/(float)(100+EVENTS)));
                fprintf(f1,"Synchronous SRT(TB) (x t1):                   %6.2f\n",
                        (3*synchresptime/FEVENTS)/
                        (r1sptime/(float)(100+EVENTS)));
                fprintf(f1,"Synchronous ST(/ t1):                         %6.2f\n",
                        1.0/((synchresptime/FEVENTS)/
                        (r1sptime/(float)(100+EVENTS))));

                fprintf(f1,"\n\n  Distribution of level 2 response times:\n");
                for(i=0;i<8;i++)
                        fprintf(f1,"%10d - %6d       %2d - %6d\n",
                                i+1,density[i],i+9,density[i+8]);
                fflush(f1);
        }
}
statqueue(pos,prestime,proctime)
int pos;
float prestime, proctime;
{       int j;

        pqueuesize=0;

        for(j=pos;((j<(100+EVENTS))&&(timelist[j]<(prestime+proctime)));j++)
```

```
		pqueuesize++;

	if(pqueuesize>mqueuesize)
		mqueuesize=pqueuesize,

	rqueuesize+=pqueuesize;

}
gauss()		/* generate gaussian rv using central limit theorm */
{	register int i,j,
	j=0;
	for(i=0;i<20;i++)
		j+= 017 & random();
	return(j/20);
}
```

Results of Asynchronous System Simulation Uniform Distribution (Both Levels)

```
                   System simulation
Sample set size:                                   200000
Processing time of level 2 (times level 1):  1.00
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.53

          Asynchronous system statistics
Average size of level-level queue:            5764.08
Maximum size of level-level queue:              11596
Average response time    (level 2):           3056.16
Average system response time (times t1):  6113.30
Approximate Percent Idle time (level 2)          0.00

          Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.56
Synchronous SRT(TB) (x t1):                         3.84
Synchronous ST(/ t1):                               0.78


      Distribution of level 2 response times
             1 -   12479          9 -   12581
             2 -   12480         10 -   12640
             3 -   12608         11 -   12221
             4 -   12499         12 -   12600
             5 -   12579         13 -   12332
             6 -   12506         14 -   12474
             7 -   12592         15 -   12562
             8 -   12453         16 -   12394


                   System simulation
Sample set size:                                   200000
Processing time of level 2 (times level 1):  0.95
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.51

          Asynchronous system statistics
Average size of level-level queue:              976.24
Maximum size of level-level queue:                2230
Average response time    (level 2):             493.88
Average system response time (times t1):    988.76
Approximate Percent Idle time (level 2)          0.00

          Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.48
Synchronous SRT(TB) (x t1):                         3.72
Synchronous ST(/ t1):                               0.81


      Distribution of level 2 response times:
             1 -   12623          9 -   12424
             2 -   12496         10 -   12437
             3 -   12568         11 -   12467
             4 -   12538         12 -   12302
             5 -   12392         13 -   12479
             6 -   12515         14 -   12504
             7 -   12578         15 -   12579
             8 -   12478         16 -   12620
```

```
                 System simulation

Sample set size:                                200000
Processing time of level 2 (times level 1):   0.90
Average processing time (level 1):              0.50
Average processing time (level 2):              0.48

      Asynchronous system statistics
Average size of level-level queue:              3.49
Maximum size of level-level queue:                30
Average response time   (level 2):              2.02
Average system response time (times t1):        5.04
Approximate Percent Idle time (level 2)         4.13

      Synchronous system statistics
Synchronous SRT(DB) (x t1):                     2.41
Synchronous SRT(TB) (x t1):                     3.62
Synchronous ST(/ t1):                           0.83


   Distribution of level 2 response times:
          1 -  12606        9 -  12614
          2 -  12606       10 -  12404
          3 -  12365       11 -  12362
          4 -  12298       12 -  12441
          5 -  12633       13 -  12349
          6 -  12568       14 -  12501
          7 -  12601       15 -  12307
          8 -  12774       16 -  12571

                 System simulation

Sample set size:                                200000
Processing time of level 2 (times level 1):   0.85
Average processing time (level 1):              0.50
Average processing time (level 2):              0.45

      Asynchronous system statistics
Average size of level-level queue:              1.42
Maximum size of level-level queue:                16
Average response time   (level 2):              0.99
Average system response time (times t1):        2.99
Approximate Percent Idle time (level 2)         9.26

      Synchronous system statistics
Synchronous SRT(DB) (x t1):                     2.34
Synchronous SRT(TB) (x t1):                     3.51
Synchronous ST(/ t1):                           0.86


   Distribution of level 2 response times:
          1 -  12458        9 -  12590
          2 -  12573       10 -  12617
          3 -  12504       11 -  12567
          4 -  12363       12 -  12488
          5 -  12369       13 -  12444
          6 -  12643       14 -  12608
          7 -  12475       15 -  12464
          8 -  12385       16 -  12452
```

```
                   System simulation

Sample set size:                                    200000
Processing time of level 2 (times level 1):   0.80
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.42

        Asynchronous system statistics
Average size of level-level queue:                 0.73
Maximum size of level-level queue:                    7
Average response time   (level 2):                 0.67
Average system response time (times t1):       2.33
Approximate Percent Idle time (level 2)        15.11

        Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.26
Synchronous SRT(TB) (x t1):                         3.39
Synchronous ST(/ t1):                                0.88


    Distribution of level 2 response times:
         1 -  12715        9 -  12579
         2 -  12334       10 -  12442
         3 -  12676       11 -  12404
         4 -  12388       12 -  12504
         5 -  12591       13 -  12413
         6 -  12486       14 -  12402
         7 -  12701       15 -  12367
         8 -  12470       16 -  12528

                   System simulation

Sample set size:                                    200000
Processing time of level 2 (times level 1):   0.75
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.40

        Asynchronous system statistics
Average size of level-level queue:                 0.47
Maximum size of level-level queue:                    6
Average response time   (level 2):                 0.54
Average system response time (times t1):       2.08
Approximate Percent Idle time (level 2)        19.79

        Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.20
Synchronous SRT(TB) (x t1):                         3.30
Synchronous ST(/ t1):                                0.91


    Distribution of level 2 response times:
         1 -  12528        9 -  12299
         2 -  12488       10 -  12402
         3 -  12461       11 -  12442
         4 -  12530       12 -  12627
         5 -  12657       13 -  12592
         6 -  12552       14 -  12639
         7 -  12518       15 -  12303
         8 -  12424       16 -  12538
```

```
                  System simulation

Sample set size:                                200000
Processing time of level 2 (times level 1):  0.70
Average processing time (level 1):             0.50
Average processing time (level 2):             0.37

        Asynchronous system statistics
Average size of level-level queue:             0.29
Maximum size of level-level queue:                4
Average response time   (level 2):             0.45
Average system response time (times t1):      1.89
Approximate Percent Idle time (level 2)       25.40

        Synchronous system statistics
Synchronous SRT(DB) (x t1):                    2.14
Synchronous SRT(TB) (x t1):                    3.21
Synchronous ST(/ t1):                          0.93


   Distribution of level 2 response times:
          1 -  12427        9 -  12345
          2 -  12610       10 -  12499
          3 -  12508       11 -  12433
          4 -  12517       12 -  12447
          5 -  12551       13 -  12642
          6 -  12484       14 -  12478
          7 -  12485       15 -  12499
          8 -  12448       16 -  12627

                  System simulation

Sample set size:                                200000
Processing time of level 2 (times level 1):  0.65
Average processing time (level 1):             0.50
Average processing time (level 2):             0.34

        Asynchronous system statistics
Average size of level-level queue:             0.17
Maximum size of level-level queue:                2
Average response time   (level 2):             0.38
Average system response time (times t1):      1.76
Approximate Percent Idle time (level 2)       30.85

        Synchronous system statistics
Synchronous SRT(DB) (x t1):                    2.08
Synchronous SRT(TB) (x t1):                    3.13
Synchronous ST(/ t1):                          0.96


   Distribution of level 2 response times:
          1 -  12503        9 -  12439
          2 -  12736       10 -  12485
          3 -  12482       11 -  12472
          4 -  12569       12 -  12590
          5 -  12294       13 -  12490
          6 -  12422       14 -  12479
          7 -  12698       15 -  12327
          8 -  12458       16 -  12556
```

```
                     System simulation

Sample set size:                                   200000
Processing time of level 2 (times level 1):  0.60
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.32

        Asynchronous system statistics
Average size of level-level queue:                  0.09
Maximum size of level-level queue:                     1
Average response time   (level 2):                  0.34
Average system response time (times t1):            1.67
Approximate Percent Idle time (level 2)            36.08

        Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.05
Synchronous SRT(TB) (x t1):                         3.07
Synchronous ST(/ t1):                               0.98


   Distribution of level 2 response times:
          1 -  12414        9 -  12525
          2 -  12558       10 -  12522
          3 -  12514       11 -  12353
          4 -  12525       12 -  12470
          5 -  12521       13 -  12583
          6 -  12388       14 -  12300
          7 -  12469       15 -  12738
          8 -  12530       16 -  12590

                     System simulation

Sample set size:                                   200000
Processing time of level 2 (times level 1):  0.55
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.29

        Asynchronous system statistics
Average size of level-level queue:                  0.03
Maximum size of level-level queue:                     1
Average response time   (level 2):                  0.30
Average system response time (times t1):            1.60
Approximate Percent Idle time (level 2)            41.29

        Synchronous system statistics
Synchronous SRT(DB) (x t1)                          2.02
Synchronous SRT(TB) (x t1)                          3.02
Synchronous ST(/ t1):                               0.99


   Distribution of level 2 response times:
          1 -  12715        9 -  12467
          2 -  12387       10 -  12420
          3 -  12443       11 -  12800
          4 -  12518       12 -  12424
          5 -  12543       13 -  12510
          6 -  12420       14 -  12515
          7 -  12615       15 -  12493
          8 -  12421       16 -  12309
```

```
                    System simulation

Sample set size:                                  200000
Processing time of level 2 (times level 1):         0.50
Average processing time (level 1)                   0.50
Average processing time (level 2):                  0.27

            Asynchronous system statistics
Average size of level-level queue:                  0.00
Maximum size of level-level queue:                     0
Average response time   (level 2):                  0.27
Average system response time (times t1):            1.54
Approximate Percent Idle time (level 2)            45.85

            Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.00
Synchronous SRT(TB) (x t1):                         3.00
Synchronous ST(/ t1):                               1.00

     Distribution of level 2 response times:
           1 -   12431          9 -   12626
           2 -   12573         10 -   12516
           3 -   12501         11 -   12449
           4 -   12542         12 -   12599
           5 -   12570         13 -   12414
           6 -   12438         14 -   12152
           7 -   12574         15 -   12523
           8 -   12538         16 -   12554
```

Results of Simulation
Gaussian Distribution

```
                System simulation
Sample set size:                                      200000
Processing time of level 2 (times level 1):             1.00
Average processing time (level 1):                      0.50
Average processing time (level 2):                      0.50

            Asynchronous system statistics
Average size of level-level queue:                    253.73
Maximum size of level-level queue:                       520
Average response time   (level 2):                    127.36
Average system response time (times t1):              255.71
Approximate Percent Idle time (level 2)                 0.00

            Synchronous system statistics
Synchronous SRT(DB) (x t1):                             2.10
Synchronous SRT(TB) (x t1):                             3.15
Synchronous ST(/ t1):                                   0.95

     Distribution of level 2 response times:
           1 -      0          9 -  49018
           2 -      0         10 -  13789
           3 -      0         11 -   1476
           4 -     34         12 -     56
           5 -   1307         13 -      2
           6 -  12841         14 -      0
           7 -  47401         15 -      0
           8 -  74076         16 -      0


                System simulation
Sample set size:                                      200000
Processing time of level 2 (times level 1):             0.95
Average processing time (level 1):                      0.50
Average processing time (level 2):                      0.48

            Asynchronous system statistics
Average size of level-level queue:                     36.45
Maximum size of level-level queue:                       201
Average response time   (level 2):                     18.52
Average system response time (times t1):               38.03
Approximate Percent Idle time (level 2)                 3.03

            Synchronous system statistics
Synchronous SRT(DB) (x t1):                             2.08
Synchronous SRT(TB) (x t1):                             3.12
Synchronous ST(/ t1):                                   0.96

     Distribution of level 2 response times
           1 -      0          9 -  49526
           2 -      0         10 -  13784
           3 -      0         11 -   1521
           4 -     45         12 -     63
           5 -   1302         13 -      0
           6 -  12442         14 -      0
           7 -  47324         15 -      0
           8 -  73993         16 -      0
```

```
              System simulation

Sample set size:                                 200000
Processing time of level 2 (times level 1):        0.90
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.45

           Asynchronous system statistics
Average size of level-level queue:                 0.15
Maximum size of level-level queue:                    1
Average response time  (level 2):                  0.46
Average system response time (times t1):           1.92
Approximate Percent Idle time (level 2)           10.49

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                        2.03
Synchronous SRT(TB) (x t1):                        3.04
Synchronous ST(/ t1):                              0.99

    Distribution of level 2 response times:
            1 -      0         9 -  49272
            2 -      0        10 -  13872
            3 -      1        11 -   1494
            4 -     56        12 -     59
            5 -   1322        13 -      0
            6 -  12323        14 -      0
            7 -  47329        15 -      0
            8 -  74272        16 -      0
```

```
              System simulation

Sample set size:                                 200000
Processing time of level 2 (times level 1):        0.85
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.43

           Asynchronous system statistics
Average size of level-level queue:                 0.05
Maximum size of level-level queue:                    1
Average response time  (level 2):                  0.44
Average system response time (times t1):           1.87
Approximate Percent Idle time (level 2)           13.87

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                        2.01
Synchronous SRT(TB) (x t1):                        3.02
Synchronous ST(/ t1):                              0.99

    Distribution of level 2 response times:
            1 -      0         9 -  49852
            2 -      0        10 -  13797
            3 -      0        11 -   1471
            4 -     51        12 -     50
            5 -   1257        13 -      2
            6 -  12544        14 -      0
            7 -  47199        15 -      0
            8 -  73777        16 -      0
```

```
                 System simulation

Sample set size:                                  200000
Processing time of level 2 (times level 1):         0.80
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.40

           Asynchronous system statistics
Average size of level-level queue:                  0.00
Maximum size of level-level queue:                     1
Average response time   (level 2):                  0.40
Average system response time (times t1):            1.80
Approximate Percent Idle time (level 2)            20.18

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.00
Synchronous SRT(TB) (x t1):                         3.00
Synchronous ST(/ t1):                               1.00


     Distribution of level 2 response times:
            1 -      0         9 -  49437
            2 -      0        10 -  13690
            3 -      1        11 -   1501
            4 -     39        12 -     59
            5 -   1249        13 -      0
            6 -  12583        14 -      0
            7 -  47684        15 -      0
            8 -  73757        16 -      0

                 System simulation

Sample set size:                                  200000
Processing time of level 2 (times level 1):         0.75
Average processing time (level 1):                  0.50
Average processing time (level 2):                  0.38

           Asynchronous system statistics
Average size of level-level queue:                  0.00
Maximum size of level-level queue:                     1
Average response time   (level 2):                  0.38
Average system response time (times t1):            1.76
Approximate Percent Idle time (level 2)            24.48

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                         2.00
Synchronous SRT(TB) (x t1):                         3.00
Synchronous ST(/ t1):                               1.00


     Distribution of level 2 response times:
            1 -      0         9 -  49183
            2 -      0        10 -  13767
            3 -      1        11 -   1545
            4 -     48        12 -     39
            5 -   1338        13 -      0
            6 -  12661        14 -      0
            7 -  47178        15 -      0
            8 -  74240        16 -      0
```

```
                System simulation

Sample set size:                                 200000
Processing time of level 2 (times level 1):        0.70
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.35

           Asynchronous system statistics
Average size of level-level queue:                 0.00
Maximum size of level-level queue:                    1
Average response time   (level 2):                 0.35
Average system response time (times t1):           1.71
Approximate Percent Idle time (level 2)           29.39

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                        2.00
Synchronous SRT(TB) (x t1):                        3.00
Synchronous ST(/ t1):                              1.00

   Distribution of level 2 response times:
          1 -       0         9 -  48985
          2 -       0        10 -  13707
          3 -       0        11 -   1447
          4 -      49        12 -     50
          5 -    1231        13 -      0
          6 -   12670        14 -      0
          7 -   47641        15 -      0
          8 -   74220        16 -      0

                System simulation

Sample set size:                                 200000
Processing time of level 2 (times level 1):        0.65
Average processing time (level 1):                 0.50
Average processing time (level 2):                 0.33

           Asynchronous system statistics
Average size of level-level queue:                 0.00
Maximum size of level-level queue:                    0
Average response time   (level 2):                 0.33
Average system response time (times t1):           1.66
Approximate Percent Idle time (level 2)           34.31

           Synchronous system statistics
Synchronous SRT(DB) (x t1):                        2.00
Synchronous SRT(TB) (x t1):                        3.00
Synchronous ST(/ t1):                              1.00

   Distribution of level 2 response times:
          1 -       0         9 -  49488
          2 -       0        10 -  13743
          3 -       0        11 -   1526
          4 -      44        12 -     61
          5 -    1323        13 -      0
          6 -   12660        14 -      0
          7 -   47239        15 -      0
          8 -   73916        16 -      0
```

```
                System simulation

Sample set size:                                   200000
Processing time of level 2 (times level 1).          0.60
Average processing time (level 1):                   0.50
Average processing time (level 2):                   0.30

     Asynchronous system statistics
Average size of level-level queue:                   0.00
Maximum size of level-level queue:                      0
Average response time   (level 2):                   0.30
Average system response time (times t1):             1.60
Approximate Percent Idle time (level 2)             40.02

     Synchronous system statistics
Synchronous SRT(DB) (x t1):                          2.00
Synchronous SRT(TB) (x t1):                          3.00
Synchronous ST(/ t1):                                1.00


  Distribution of level 2 response times:
         1 -       0          9 -  49150
         2 -       0         10 -  13856
         3 -       1         11 -   1537
         4 -      46         12 -     54
         5 -    1281         13 -      0
         6 -   12724         14 -      0
         7 -   47220         15 -      0
         8 -   74131         16 -      0

                System simulation

Sample set size:                                   200000
Processing time of level 2 (times level 1):          0.55
Average processing time (level 1):                   0.50
Average processing time (level 2):                   0.28

     Asynchronous system statistics
Average size of level-level queue:                   0.00
Maximum size of level-level queue:                      0
Average response time   (level 2):                   0.27
Average system response time (times t1):             1.55
Approximate Percent Idle time (level 2)             45.23

     Synchronous system statistics
Synchronous SRT(DB) (x t1):                          2.00
Synchronous SRT(TB) (x t1):                          3.00
Synchronous ST(/ t1):                                1.00


  Distribution of level 2 response times:
         1 -       0          9 -  49458
         2 -       0         10 -  13845
         3 -       0         11 -   1499
         4 -      45         12 -     59
         5 -    1244         13 -      0
         6 -   12714         14 -      0
         7 -   47474         15 -      0
         8 -   73662         16 -      0
```

```
                 System simulation

Sample set size:                                 200000
Processing time of level 2 (times level 1):  0.50
Average processing time (level 1):             0.50
Average processing time (level 2):             0.25

         Asynchronous system statistics
Average size of level-level queue:             0.00
Maximum size of level-level queue:                0
Average response time   (level 2):             0.26
Average system response time (times t1):       1.51
Approximate Percent Idle time (level 2)       48.77

         Synchronous system statistics
Synchronous SRT(DB) (x t1):                    2.00
Synchronous SRT(TB) (x t1):                    3.00
Synchronous ST(/ t1):                          1.00


   Distribution of level 2 response times:
          1 -       0         9 -  49421
          2 -       0        10 -  13776
          3 -       0        11 -   1568
          4 -      40        12 -     47
          5 -    1322        13 -      2
          6 -   12726        14 -      0
          7 -   46966        15 -      0
          8 -   74132        16 -      0
```

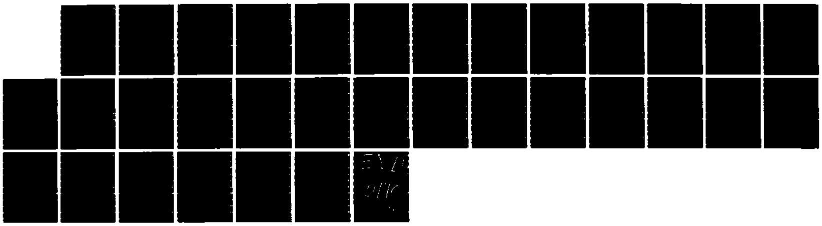
SIMULATOR LISTINGS

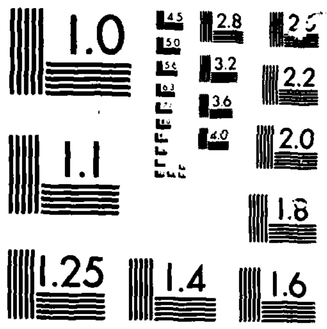
Asynchronous System Simulator -- Both Levels Uniform Random Variables

```
#include<stdio.h>

/*****************************************************************/
/*                                                               */
/* This simulator was written for a Gould Powernode 9080 running UNIX */
/*  4.2   BSD.   It   simulates   the   execution   of   an      */
/* asynchronous two level system, keeping track of the level-to-level */
/* buffersizes and the response times of the second level.  For this */
/* simulator, it is assumed that the distribution describing the res- */
/* time of the both levels is the uniform distribution function,  and */
/* It is run on a total of 200,000 data sets per simulation statistic */
/* requiring more than  1.0Mb  storage  and  more  than  2  hours  to */
/* complete.                                                     */
/*                                                               */
/*****************************************************************/

#define EVENTS 200000             /* Events per simulation          */
#define FEVENTS 200000.0          /* floating point representation  */
FILE *f1;                          /* file pointers                  */
int density[16];                   /* for density function           */
                                   /* Gould Firebreather 9080 Compiler */
                                   /* needs the NOBASE because       */
                                   /* it needs to use a different    */
                                   /* addressing mode to handle large */
/* NOBASE */                       /* arrays.                        */
float timelist[100+EVENTS];       /* list of event times            */
int pqueuesize,mqueuesize;         /* present and maximum queuesizes */
int rqueuesize;                    /* total jobs stored in queues    */
float idletime;                   /* idletime for second level      */
float r1sptime;                   /* response time for first  level */
float resptime;                   /* response time for second level */
float rrsptime;                   /* cumulative time for second level */
float sysresptime;                /* system response time           */

float pcnt[] = {1.00,             /* ratio of:_____________________ */
                0.95, 0.90,        /*          response time of level 2 */
                0.85, 0.80,        /*          ----------------------- */
                0.75, 0.70,        /*          response time of level 1 */
                0.65, 0.60,
                0.55, 0.50,
                0.00 };
main()
{       float ktime;              /* time keeper                    */
        float rtime;              /* response time keeper           */
        int eventtime;             /* event duration                 */
        int i;                     /* event counter                  */
        int j;                     /* event counter                  */
        int sum;                   /*  gaussian only                 */

        f1=fopen("simulation results","w"); /* open files           */

        srandom(getpid());         /* initialize random number generator */
        if(f1==NULL)
                exit();
        for(j=0;pcnt[j]>0.0;j++)
        {       pqueuesize=0;      /* initialize all statistical keepers */
                mqueuesize=0;
                rqueuesize=0;
                idletime= 0.0;
                r1sptime= 0.0;
                resptime= 0.0;
                rrsptime= 0.0;
                sysresptime= 0.0;
                for(i=0;i<16;i++)
                        density[i]=0;

                ktime=8.0/16.0;

                timelist[0] = (float)((1+017&random()))/16.0;
                for(i=1;i<100+EVENTS;i++)  /* generate random events   */
                {                               /* 1-16                        */
                  timelist[i] = timelist[i-1] +
                    ((float)(1+(017&random())))/16.0;
                }

                r1sptime=timelist[EVENTS-1];
                ktime=0.0;                          /* present time    */
                rrsptime=0.0;
                for(i=0;i<EVENTS;i++)
                {        eventtime = 017&random();  /* random number   */
                                                    /* 0 and 15        */

                         density[eventtime]++;
                         eventtime+=1;
                         rtime = pcnt[j]/16.0*(float) eventtime;
                         rrsptime+=rtime;
                         statqueue(i+1,ktime,rtime);    /* queuesize   */

                         if(ktime<timelist[i])  /* update present time */
                         {        idletime+=timelist[i]-ktime;
                                  ktime=timelist[i]+rtime;
                         }
                         else
                                  ktime+=rtime;

                         resptime+=ktime-timelist[i];

                }
                fprintf(f1,"            System simulation0);
                fprintf(f1,"Sample set size:                          %6d0,
                         EVENTS);
                fprintf(f1,"Processing time of level 2 (times level 1):%6.2f0,
                         pcnt[j]);
                fprintf(f1,"Average processing time (level 1):        %6.2f0,
                         r1sptime/(FEVENTS-1.0));
                fprintf(f1,"Average processing time (level 2):        %6.2f0,
                         rrsptime/FEVENTS);
                fprintf(f1,"Average size of level-level queue:        %6.2f0,
                         (float) rqueuesize/FEVENTS),
                fprintf(f1,"Maximum size of level-level queue:        %6d0,
                         mqueuesize);
                fprintf(f1,"Average system response time (times t1):  %6.2f0,
                         1.0+(rrsptime/FEVENTS)/(r1sptime/FEVENTS)
                         +((rqueuesize/FEVENTS)*
                         (rrsptime/FEVENTS)/(r1sptime/(FEVENTS-1.0))));
                fprintf(f1,"Approximate Percent Idle time (level 2)   %6.2f0,
                         100.0*idletime/timelist[EVENTS]);

                fprintf(f1,"   Distribution of level 2 response times:0);
                for(i=0;i<8;i++)
                         fprintf(f1,"%10d - %6d        %2d - %6d0,
                                i+1,density[i],i+9,density[i+8]);
                fflush(f1);
        }
}
statqueue(pos,prestime,proctime)
int pos;
float prestime, proctime;
{         int j,

          pqueuesize=0,

          for(j=pos;((j<(100+EVENTS))&&(timelist[j]<(prestime+proctime)));j++)
                  pqueuesize++;

          if(pqueuesize>mqueuesize)
                  mqueuesize=pqueuesize;

          rqueuesize+=pqueuesize;

}
```

AD-A167 622 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN B W SMITH MAY 85 ARO-18790.17-EL-APP-G DAAG29-82-K-0101 4/4

UNCLASSIFIED F/G 9/2 NL





MICROCOP' CHART

Asynchronous System Simulator -- Both Levels Gaussian Random Variables

```
#include<stdio.h>

/*********************************************************************/
/*                                                                   */
/* This simulator was written for a Gould Powernode 9080 running UNIX */
/* 4.2 BSD.  It simulates the execution of both a synchronous and  an */
/* asynchronous two level system, keeping track of the level-to-level */
/* buffersizes and the response times of the second level.  For this */
/* simulator, it is assumed that the distribution describing the res- */
/* time of the both levels is the uniform distribution function,  and */
/* It is run on a total of 100,000 data sets per simulation statistic */
/* requiring more than  1.0Mb  storage  and  more  than  2  hours  to */
/* complete.                                                         */
/*                                                                   */
/*********************************************************************/

#define EVENTS 100000               /* Events per simulation          */
#define FEVENTS 100000.0            /* floating point representation */
FILE *f1;                            /* file pointers                 */
int density[16];                     /* for density function           */
                                     /* Gould Firebreather 9080 Compiler */
                                     /* needs the NOBASE because       */
                                     /* it needs to use a different   */
                                     /* addressing mode to handle large */
/* NOBASE */                         /* arrays.                        */
float timelist[100+EVENTS];         /* list of event times            */
int pqueuesize,mqueuesize;           /* present and maximum queuesizes  */
int rqueuesize;                      /* total jobs stored in queues     */
float idletime;                     /* idletime for second level      */
float r1sptime;                     /* response time for first  level */
float resptime;                     /* response time for second level  */
float rrsptime;                     /* cumulative time for second level */
float sysresptime;                  /* system response time           */

float pcnt[] = {1.00,               /* ratio of ________________________ */
                0.95, 0.90,          /*          response time of level 2  */
                0.85, 0.80,          /*          ------------------------  */
                0.75, 0.70,          /*          response time of level 1  */
                0.65, 0.60,
                0.55, 0.50,
                0.00 };
main()
{       float ktime;                /* time keeper                    */
        float rtime;                /* response time keeper           */
        int eventtime;               /* event duration                 */
        int i;                      /* event counter                  */
        int j;                       /* event counter                  */
        int sum;                    /*  gaussian only                 */

        f1=fopen("simulation results gauss","w"); /* open files           */

        srandom(getpid());          /* initialize random number generator */
        if(f1==NULL)
                exit();
        for(j=0;pcnt[j]>0.0;j++)
        {       pqueuesize=0;       /* initialize all statistical keepers */
                mqueuesize=0;
                rqueuesize=0;
                idletime= 0.0;
                r1sptime= 0.0;
                resptime= 0.0;
                rrsptime= 0.0;
                sysresptime= 0.0;
                for(i=0;i<16;i++)
                        density[i]=0;

                ktime=8.0/16.0;

                timelist[0] = (float)((1+017&random())) 16.0.
                for(i=1;i<100+EVENTS;i++)  /* generate random events  */
                {                          /* 0-15                    */
                  timelist[i] = timelist[i-1] +
                   ((float)gauss())/16.0;
                }
                r1sptime=timelist[EVENTS-1];
                ktime=0.0;                                /* present time     */
                rrsptime=0.0;
                for(i=0;i<EVENTS;i++)
                {       eventtime = gauss();              /* random number    */
                                                          /* 0 and 15         */

                        density[eventtime]++;
                        eventtime+=1;
                        rtime = pcnt[j]/16.0*(float) eventtime;
                        rrsptime+=rtime;
                        statqueue(i+1,ktime,rtime);    /* queuesize     */

                        if(ktime<timelist[i])  /* update present time */
                        {       idletime+=timelist[i]-ktime;
                                ktime=timelist[i]+rtime;
                        }
                        else
                                ktime+=rtime;

                        resptime+=ktime-timelist[i];

                }
                fprintf(f1,"             System simulation0);
                fprintf(f1,"Sample set size:                               %6d0,
                        EVENTS);
                fprintf(f1,"Processing time of level 2 (times level 1):%6.2f0,
                        pcnt[j]);
                fprintf(f1,"Average processing time (level 1):         %6.2f0,
                        r1sptime/(FEVENTS-1.0));
                fprintf(f1,"Average processing time (level 2)          %6.2f0,
                        rrsptime/FEVENTS);
                fprintf(f1,"Average size of level-level queue.         %6.2f0,
                        (float) rqueuesize/FEVENTS);
                fprintf(f1,"Maximum size of level-level queue          %6d0,
                        mqueuesize);
                fprintf(f1,"Average system response time (times t1):   %6.2f0,
                        1.0+(rrsptime/FEVENTS)/(r1sptime/FEVENTS)
                        +((rqueuesize/FEVENTS)*
                        (rrsptime/FEVENTS)/(r1sptime/(FEVENTS-1.0))));
                fprintf(f1,"Approximate Percent Idle time (level 2)    %6.2f0,
                        100.0*idletime/timelist[EVENTS]);

                fprintf(f1,"   Distribution of level 2 response times:0);
                for(i=0;i<8;i++)
                        fprintf(f1,"%10d - %6d        %2d - %6d0,
                                i+1,density[i],i+9,density[i+8]);
                fflush(f1);
        }
}
statqueue(pos,prestime,proctime)
int pos;
float prestime, proctime;
{       int j;

        pqueuesize=0;

        for(j=pos;((j<(100+EVENTS))&&(timelist[j]<(prestime+proctime)));j++)
                pqueuesize++;

        if(pqueuesize>mqueuesize)
                mqueuesize=pqueuesize;

        rqueuesize+=pqueuesize;

}
gauss()
{       register int i,j;

        j=0;
        for(i=0;i<20;i++)
                j+= 017 & random();
        return(j/20);
}
```

Synchronous System Simulator -- Both Levels Uniform Random Variables

```
#include<stdio.h>
#define EVENTS 1000000
#define FEVENTS 1000000.0
FILE *f1;                           /* file pointers                        */
int density1[16];                   /* for density function level 1          */
int density2[16];                   /* for density function level 2          */
float sysresptime;                  /* system response time                 */
float pcnt [] = { 1.00,             /* ratio of mean values of levels       */
                0.95, 0.90,
                0.85, 0.80,
                0.75, 0.70,
                0.65, 0.60,
                0.55, 0.50,
               -1 } ;

main()
{       float l1time;               /* time keeper level 1                  */
        float l2time;               /* time keeper level 2                  */
        float t1time;               /* time keeper level 1                  */
        float t2time;               /* time keeper level 2                  */
        int eventtime1;              /* event duration level 1               */
        int eventtime2;              /* event duration level 2               */
        int i;                       /* event counter                        */
        int j;                       /* pointer to statistical info          */
        float xbar;                 /* for simplicity -- avg resp time of   */
                                    /* level 1                              */

        f1=fopen("dblbuf","a"); /* open files                               */

        if(f1==NULL)
                exit();

        l1time=0.0;
        l2time=0.0;
        t1time=0.0;
        t2time=0.0;
        sysresptime = 0.0;

        srandom(getpid());           /* initialize random number generator */
        for(j=0,pcnt[j]>0;j++)
        {       l1time = 0.0;           /* time keeper level 1              */
                l2time = 0.0;           /* time keeper level 2              */
                t1time = 0.0;           /* time keeper level 1              */
                t2time = 0.0;           /* time keeper level 2              */
                sysresptime = 0.0;
                for(i=0;i<16;i++)
                        density1[i]=density2[i]=0.
                for(i=0;i<EVENTS;i++)
                {       eventtime1 = 017&random().   /* random number   */
                        eventtime2 = 017&random().   /* random number   */
                                                     /* 0  to 15        */
                        density1[eventtime1]++;      /* density fun     */
                        density2[eventtime2]++;      /* density fun     */

                        l1time = (float) eventtime1 / 16.0 ;
                        l2time = pcnt[j] * (float) eventtime2 / 16.0 ;

                        t1time+=l1time.              /* avg resp time */
                        t2time+=l2time;              /* avg resp time */

                        if(l1time<l2time)
                                sysresptime += l2time;
                        else
                                sysresptime += l1time.
                }
                xbar = t1time/FEVENTS,
                fprintf(f1,"0,
                fprintf(f1,"                  synchronous system0).
                fprintf(f1,"Sample set size                              %6d0,
                        EVENTS).
                fprintf(f1,"Processing time of level 2 (times level 1) %6.2f0,
                        pcnt[j]).
                fprintf(f1,"Average processing time (level 1)          %6.2f0
                xbar);
fprintf(f1,"Average processing time (level 2):          %6.2f0,
                t2time/FEVENTS);
fprintf(f1,"Average DB system response time (* t1):     %6.2f0,
                (2.0*sysresptime/FEVENTS)/xbar);
fprintf(f1,"Average TB system response time (* t1):     %6.2f0,
                (3.0*sysresptime/FEVENTS)/xbar);

fprintf(f1,"    Distribution of level 1 response times:0);
for(i=0;i<8;i++)
                fprintf(f1,"%10d - %6d       %6d - %6d0,
                        i,density1[i],i+8,density1[i+8]);
fprintf(f1,"    Distribution of level 2 response times:0);
for(i=0;i<8;i++)
                fprintf(f1,"%10.2f - %6d     %8.2f - %6d0,
                 pcnt[j]*(float)i,density2[i],
                 pcnt[j]*(float)(i+8),density2[i+8]);
        }
}
```

Synchronous System Simulator -- Both Levels Gaussian Random Variables

```
#include<stdio.h>
#define EVENTS 100000
#define FEVENTS 100000.0
FILE *f1;                          /* file pointers                   */
int density1[16];                  /* for density function level 1    */
int density2[16];                  /* for density function level 2    */
float sysresptime;                 /* system response time            */
float pcnt [] = { 1.00,            /* ratio of mean values of levels  */
                  0.95, 0.90,
                  0.85, 0.80,
                  0.75, 0.70,
                  0.65, 0.60,
                  0.55, 0.50,
                  -1 } ;

main()
{
        float l1time;              /* time keeper level 1             */
        float l2time;              /* time keeper level 2             */
        float t1time;              /* time keeper level 1             */
        float t2time;              /* time keeper level 2             */
        int eventtime1;            /* event duration level 1          */
        int eventtime2;            /* event duration level 2          */
        int i;                     /* event counter                   */
        int j;                     /* pointer to statistical info     */
        float xbar;                /* for simplicity -- avg resp time of */
                                   /* level 1                         */
        int k;                     /* for generation of a gaussian fun */
                                   /* using the central limit theorm  */

        f1=fopen("dblbuf.gauss","a"); /* open files                   */

        if(f1==NULL)
                exit();

        l1time=0.0;
        l2time=0.0;
        t1time=0.0;
        t2time=0.0;
        sysresptime = 0.0;

        srandom(getpid());         /* initialize random number generator */
        for(j=0;pcnt[j]>0;j++)
        {       l1time = 0.0;      /* time keeper level 1             */
                l2time = 0.0;      /* time keeper level 2             */
                t1time = 0.0;      /* time keeper level 1             */
                t2time = 0.0;      /* time keeper level 2             */
                for(i=0;i<16;i++)
                    density1[i]=density2[i]=0;
                sysresptime = 0.0;
                for(i=0;i<EVENTS;i++)
                {   eventtime1=0;
                    for(k=0;k<20;k++)
                       eventtime1 += 017&random();   /* random number    */
                    eventtime2=0;
                    for(k=0;k<20;k++)
                       eventtime2 += 017&random();   /* random number    */
                                                      /* 0 to 15          */
                    eventtime1 /= 20;                 /* normalization    */
                    eventtime2 /= 20;                 /* normalization    */
                    density1[eventtime1]++;           /* density fun      */
                    density2[eventtime2]++;           /* density fun      */

                    l1time = (float) eventtime1 / 16.0 ;
                    l2time = pcnt[j] * (float) eventtime2 / 16.0 ;

                    t1time+=l1time;                   /* avg resp time    */
                    t2time+=l2time;                   /* avg resp time    */

                    if(l1time<l2time)
                        sysresptime += l2time;
                    else
                        sysresptime += l1time;
                }
```

```
                        xbar = t1time/FEVENTS;
                        fprintf(f1,"0;
                        fprintf(f1,"              synchronous system0);
                        fprintf(f1,"Sample set size:                            %6d0,
                                EVENTS);
                        fprintf(f1,"Processing time of level 2 (times level 1):%6.2f0,
                                pcnt[j]);
                        fprintf(f1,"Average processing time (level 1):         %6.2f0,
                                xbar);
                        fprintf(f1,"Average processing time (level 2).         %6.2f0,
                                t2time/FEVENTS);
                        fprintf(f1,"Average DB system response time (* t1)     %6.2f0,
                                (2.0*sysresptime/FEVENTS)/xbar);
                        fprintf(f1,"Average TB system response time (* t1)     %6.2f0,
                                (3.0*sysresptime/FEVENTS)/xbar);

                        fprintf(f1,"    Distribution of level 1 response times 0);
                        for(i=0;i<8;i++)
                                fprintf(f1,"%10d - %6d       %6d - %6d0,
                                    i,density1[i],i+8,density1[i+8]);
                        fprintf(f1,"    Distribution of level 2 response times 0);
                        for(i=0;i<8;i++)
                                fprintf(f1,"%10.2f - %6d     %8.2f - %6d0,
                                  pcnt[j]*(float)i,density2[i],
                                    pcnt[j]*(float)(i+8),density2[i+8]);
                }
        }
```

Results of Simulation
Uniform Distribution

```
Asynchronous System Simulation -- Results (uniform)
Sample set size:                                  200000
Processing time of level 2 (times level 1)          1.00
Average processing time (level 1):                  0.53
Average processing time (level 2):                  0.53
Average size of level-level queue:                 62.81
Maximum size of level-level queue:                   177
Average system response time (times t1):           64.62
Approximate Percent Idle time (level 2)             0.30
   Distribution of level 2 response times:
         1 -  12656        9 -  12426
         2 -  12499       10 -  12245
         3 -  12356       11 -  12446
         4 -  12700       12 -  12438
         5 -  12523       13 -  12664
         6 -  12513       14 -  12548
         7 -  12596       15 -  12557
         8 -  12448       16 -  12385

Asynchronous System Simulation -- Results (uniform)

Sample set size:                                  200000
Processing time of level 2 (times level 1)          0.95
Average processing time (level 1):                  0.53
Average processing time (level 2):                  0.51
Average size of level-level queue:                  6.01
Maximum size of level-level queue:                    39
Average system response time (times t1):            7.69
Approximate Percent Idle time (level 2)             4.58
   Distribution of level 2 response times:
         1 -  12396        9 -  12541
         2 -  12475       10 -  12281
         3 -  12571       11 -  12637
         4 -  12447       12 -  12542
         5 -  12378       13 -  12560
         6 -  12510       14 -  12606
         7 -  12395       15 -  12489
         8 -  12474       16 -  12698
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                    200000
Processing time of level 2 (times level 1):          0.90
Average processing time (level 1)                    0.53
Average processing time (level 2)                    0.48
Average size of level-level queue                    2.97
Maximum size of level-level queue                      24
Average system response time (times t1)              4.59
Approximate Percent Idle time (level 2)              9.57

   Distribution of level 2 response times:
         1 -  12628        9 -  12550
         2 -  12766       10 -  12376
         3 -  12350       11 -  12691
         4 -  12295       12 -  12562
         5 -  12450       13 -  12502
         6 -  12438       14 -  12383
         7 -  12380       15 -  12606
         8 -  12666       16 -  12357

Asynchronous System Simulation -- Results (uniform)

Sample set size:                                    200000
Processing time of level 2 (times level 1):          0.85
Average processing time (level 1):                   0.53
Average processing time (level 2):                   0.45
Average size of level-level queue:                   1.76
Maximum size of level-level queue:                     30
Average system response time (times t1):             3.34
Approximate Percent Idle time (level 2)             14.88

   Distribution of level 2 response times:
         1 -  12623        9 -  12649
         2 -  12349       10 -  12420
         3 -  12544       11 -  12562
         4 -  12540       12 -  12613
         5 -  12274       13 -  12665
         6 -  12527       14 -  12443
         7 -  12507       15 -  12379
         8 -  12696       16 -  12209
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                  200000
Processing time of level 2 (times level 1):         0.80
Average processing time (level 1):                  0.53
Average processing time (level 2):                  0.43
Average size of level-level queue:                  1.22
Maximum size of level-level queue                     13
Average system response time (times t1):            2.78
Approximate Percent Idle time (level 2)            19.63

   Distribution of level 2 response times:
         1 -  12590        9 -  12536
         2 -  12320       10 -  12548
         3 -  12349       11 -  12517
         4 -  12514       12 -  12639
         5 -  12452       13 -  12622
         6 -  12427       14 -  12323
         7 -  12540       15 -  12636
         8 -  12371       16 -  12616
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                  200000
Processing time of level 2 (times level 1):        0.75
Average processing time (level 1):                 0.53
Average processing time (level 2):                 0.40
Average size of level-level queue:                 0.88
Maximum size of level-level queue:                   12
Average system response time (times t1):           2.41
Approximate Percent Idle time (level 2)           24.66

   Distribution of level 2 response times:
         1 -  12579        9 -  12619
         2 -  12537       10 -  12394
         3 -  12579       11 -  12335
         4 -  12721       12 -  12459
         5 -  12571       13 -  12209
         6 -  12365       14 -  12529
         7 -  12434       15 -  12611
         8 -  12605       16 -  12453
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                    200000
Processing time of level 2 (times level 1):           0.70
Average processing time (level 1):                    0.53
Average processing time (level 2):                    0.37
Average size of level-level queue:                    0.67
Maximum size of level-level queue:                      10
Average system response time (times t1):              2.17
Approximate Percent Idle time (level 2)              29.63

   Distribution of level 2 response times:
         1 -  12265        9 -  12373
         2 -  12506       10 -  12328
         3 -  12435       11 -  12543
         4 -  12437       12 -  12528
         5 -  12560       13 -  12696
         6 -  12626       14 -  12677
         7 -  12504       15 -  12512
         8 -  12468       16 -  12542
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                    200000
Processing time of level 2 (times level 1):           0.65
Average processing time (level 1):                    0.53
Average processing time (level 2):                    0.35
Average size of level-level queue:                    0.51
Maximum size of level-level queue:                       9
Average system response time (times t1):              1.98
Approximate Percent Idle time (level 2)              34.64

   Distribution of level 2 response times:
         1 -  12339        9 -  12635
         2 -  12409       10 -  12318
         3 -  12493       11 -  12399
         4 -  12467       12 -  12665
         5 -  12456       13 -  12631
         6 -  12596       14 -  12433
         7 -  12619       15 -  12743
         8 -  12347       16 -  12450
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                200000
Processing time of level 2 (times level 1):      0.60
Average processing time (level 1):               0.53
Average processing time (level 2):               0.32
Average size of level-level queue:               0.38
Maximum size of level-level queue:                  8
Average system response time (times t1):         1.83
Approximate Percent Idle time (level 2)         39.78

    Distribution of level 2 response times:
          1 -  12506        9 -  12391
          2 -  12429       10 -  12380
          3 -  12453       11 -  12482
          4 -  12642       12 -  12359
          5 -  12535       13 -  12655
          6 -  12729       14 -  12476
          7 -  12487       15 -  12473
          8 -  12504       16 -  12499
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                200000
Processing time of level 2 (times level 1):      0.55
Average processing time (level 1):               0.53
Average processing time (level 2):               0.29
Average size of level-level queue:               0.28
Maximum size of level-level queue:                  7
Average system response time (times t1):         1.71
Approximate Percent Idle time (level 2)         44.67

    Distribution of level 2 response times:
          1 -  12512        9 -  12606
          2 -  12568       10 -  12391
          3 -  12451       11 -  12582
          4 -  12568       12 -  12374
          5 -  12453       13 -  12295
          6 -  12348       14 -  12304
          7 -  12692       15 -  12663
          8 -  12556       16 -  12637
```

```
Asynchronous System Simulation -- Results (uniform)

Sample set size:                                  200000
Processing time of level 2 (times level 1):         0.50
Average processing time (level 1):                  0.53
Average processing time (level 2):                  0.27
Average size of level-level queue:                  0.22
Maximum size of level-level queue:                     6
Average system response time (times t1):            1.61
Approximate Percent Idle time (level 2)            48.80

   Distribution of level 2 response times:
          1 -   12477          9 -   12426
          2 -   12700         10 -   12431
          3 -   12353         11 -   12400
          4 -   12307         12 -   12585
          5 -   12569         13 -   12624
          6 -   12480         14 -   12546
          7 -   12374         15 -   12581
          8 -   12416         16 -   12731
```

Results of Synchronous Simulation
Uniform Distribution (both levels)

```
            Synchronous System -- Simulation Results
Sample set size:                                     1000000
Processing time of level 2 (times level 1):           1.00
Average processing time (level 1):                     0.47
Average processing time (level 2):                     0.47
Average DB system response time (* t1):                2.71
Average TB system response time (* t1):                4.06
   Distribution of level 1 response times:
          0 -   62611           8 -   62431
          1 -   61913           9 -   62431
          2 -   63096          10 -   62631
          3 -   62507          11 -   62470
          4 -   62055          12 -   62663
          5 -   62443          13 -   62532
          6 -   62348          14 -   62681
          7 -   62763          15 -   62425
   Distribution of level 2 response times:
       0.00 -   62649        8.00 -   62404
       1.00 -   62391        9.00 -   62432
       2.00 -   62940       10.00 -   62296
       3.00 -   62249       11.00 -   62418
       4.00 -   62664       12.00 -   62761
       5.00 -   62227       13.00 -   62460
       6.00 -   62898       14.00 -   62451
       7.00 -   62640       15.00 -   62120
```

```
        Synchronous System -- Simulation Results

Sample set size:                                     1000000
Processing time of level 2 (times level 1):           0.95
Average processing time (level 1):                     0.47
Average processing time (level 2):                     0.45
Average DB system response time (* t1):                2 64
Average TB system response time (* t1):                3 96


   Distribution of level 1 response times
          0 -   62729           8 -   62399
          1 -   61969           9 -   62502
          2 -   62659          10 -   62460
          3 -   62611          11 -   62471
          4 -   62624          12 -   62204
          5 -   62544          13 -   62323
          6 -   62803          14 -   62397
          7 -   62507          15 -   62798


   Distribution of level 2 response times
       0 00 -   62583         7 60 -   62943
       0 95 -   62580         8 55 -   62715
       1 90 -   62783         9 50 -   62573
       2 85 -   62311        10 45 -   62474
       3 80 -   62338        11 40 -   62521
       4 75 -   62128        12 35 -   62309
       5 70 -   62499        13 30 -   62339
       6 65 -   62205        14 25 -   62699
```

```
          Synchronous System -- Simulation Results

Sample set size:                                   1000000
Processing time of level 2 (times level 1):           0.90
Average processing time (level 1):                    0.47
Average processing time (level 2):                    0.43
Average DB system response time (* t1):               2.60
Average TB system response time (* t1):               3.90

    Distribution of level 1 response times:
            0 -  62482             8 -  62883
            1 -  62514             9 -  62571
            2 -  62397            10 -  62278
            3 -  62336            11 -  62241
            4 -  62323            12 -  62517
            5 -  62845            13 -  62655
            6 -  62754            14 -  62402
            7 -  62367            15 -  62435

    Distribution of level 2 response times:
         0.00 -  62667          7.20 -  62417
         0.90 -  62225          8.10 -  62387
         1.80 -  62505          9.00 -  62754
         2.70 -  62616          9.90 -  62483
         3.60 -  62609         10.80 -  62309
         4.50 -  62199         11.70 -  62213
         5.40 -  62466         12.60 -  62600
         6.30 -  62864         13.50 -  62686

          Synchronous System -- Simulation Results

Sample set size:                                   1000000
Processing time of level 2 (times level 1):           0.85
Average processing time (level 1)                     0.47
Average processing time (level 2)                     0.40
Average DB system response time (* t1)                2.53
Average TB system response time (* t1)                3.79

    Distribution of level 1 response times
            0 -  62428             8 -  62482
            1 -  62506             9 -  62590
            2 -  62764            10 -  62259
            3 -  62376            11 -  62471
            4 -  62475            12 -  62529
            5 -  62311            13 -  62364
            6 -  62407            14 -  62766
            7 -  62716            15 -  62556

    Distribution of level 2 response times
         0.00 -  62561          6.80 -  62185
         0.85 -  62777          7.65 -  62818
         1.70 -  62992          8.50 -  62654
         2.55 -  62255          9.35 -  62668
         3.40 -  62105         10.20 -  62882
         4.25 -  62415         11.05 -  62146
         5.10 -  62301         11.90 -  62399
         5.95 -  62656         12.75 -  62186
```

```
        Synchronous System -- Simulation Results

    Sample set size:                                    1000000
    Processing time of level 2 (times level 1):           0.80
    Average processing time (level 1):                    0.47
    Average processing time (level 2):                    0.37
    Average DB system response time (* t1):               2.46
    Average TB system response time (* t1):               3.69

       Distribution of level 1 response times:
              0 -   62454             8 -   62272
              1 -   62655             9 -   62635
              2 -   62566            10 -   62962
              3 -   62384            11 -   62366
              4 -   62621            12 -   62582
              5 -   62335            13 -   62383
              6 -   62344            14 -   62251
              7 -   62599            15 -   62591

       Distribution of level 2 response times
           0.00 -   62720          6.40 -   62582
           0.80 -   62034          7.20 -   62221
           1.60 -   62451          8.00 -   62485
           2.40 -   62340          8.80 -   62636
           3.20 -   62784          9.60 -   62649
           4.00 -   62567         10.40 -   62375
           4.80 -   62931         11.20 -   62231
           5.60 -   62688         12.00 -   62306

        Synchronous System -- Simulation Results

    Sample set size                                     1000000
    Processing time of level 2 (times level 1)            0.75
    Average processing time (level 1)                     0.47
    Average processing time (level 2)                     0.36
    Average DB system response time (* t1)                2.42
    Average TB system response time (* t1)                3.63

       Distribution of level 1 response times
              0 -   62331             8 -   62582
              1 -   62942             9 -   62269
              2 -   62886            10 -   62372
              3 -   62392            11 -   62994
              4 -   62057            12 -   62223
              5 -   62419            13 -   62382
              6 -   62594            14 -   62471
              7 -   62848            15 -   62238

       Distribution of level 2 response times:
           0.00 -   62128          6.00 -   62473
           0.75 -   62267          6.75 -   62158
           1.50 -   62201          7.50 -   63099
           2.25 -   62393          8.25 -   62702
           3.00 -   62795          9.00 -   62648
           3.75 -   61868          9.75 -   62474
           4.50 -   62857         10.50 -   62456
           5.25 -   62884         11.25 -   62597
```

```
          Synchronous System -- Simulation Results

Sample set size:                                        1000000
Processing time of level 2 (times level 1):              0.70
Average processing time (level 1):                       0.47
Average processing time (level 2):                       0.33
Average DB system response time (* t1):                  2.37
Average TB system response time (* t1):                  3.55

   Distribution of level 1 response times:
          0 -  62515            8 -  62226
          1 -  62257            9 -  62664
          2 -  62571           10 -  62478
          3 -  62362           11 -  62714
          4 -  61982           12 -  62357
          5 -  62765           13 -  62692
          6 -  62725           14 -  62722
          7 -  62889           15 -  62081

   Distribution of level 2 response times:
       0.00 -  62576         5.60 -  62441
       0.70 -  62834         6.30 -  62721
       1.40 -  62677         7.00 -  62600
       2.10 -  62308         7.70 -  62026
       2.80 -  62822         8.40 -  62664
       3.50 -  62174         9.10 -  62390
       4.20 -  62251         9.80 -  62684
       4.90 -  62605        10.50 -  62427

          Synchronous System -- Simulation Results

Sample set size:                                        1000000
Processing time of level 2 (times level 1):              0.65
Average processing time (level 1):                       0.47
Average processing time (level 2):                       0.31
Average DB system response time (* t1):                  2.32
Average TB system response time (* t1):                  3.48

   Distribution of level 1 response times:
          0 -  62564            8 -  62357
          1 -  62201            9 -  62561
          2 -  62354           10 -  62669
          3 -  62774           11 -  62677
          4 -  61901           12 -  62567
          5 -  62713           13 -  62744
          6 -  62601           14 -  62409
          7 -  62490           15 -  62418

   Distribution of level 2 response times:
       0.00 -  62496         5.20 -  62277
       0.65 -  62778         5.85 -  62153
       1.30 -  62732         6.50 -  62602
       1.95 -  62464         7.15 -  62435
       2.60 -  62456         7.80 -  62618
       3.25 -  62163         8.45 -  62660
       3.90 -  62800         9.10 -  62509
       4.55 -  62666         9.75 -  62191
```

```
          Synchronous System -- Simulation Results

      Sample set size:                                1000000
      Processing time of level 2 (times level 1):  0.60
      Average processing time (level 1):              0.47
      Average processing time (level 2):              0.28
      Average DB system response time (* t1):         2.27
      Average TB system response time (* t1):         3.41

         Distribution of level 1 response times:
                0 -   63046             8 -  62723
                1 -   62426             9 -  62312
                2 -   62174            10 -  62531
                3 -   62471            11 -  62439
                4 -   62266            12 -  62410
                5 -   62583            13 -  62851
                6 -   62791            14 -  62163
                7 -   62584            15 -  62230

         Distribution of level 2 response times:
             0.00 -   62734          4.80 -  62556
             0.60 -   62192          5.40 -  62259
             1.20 -   62138          6.00 -  62521
             1.80 -   62455          6.60 -  62235
             2.40 -   62515          7.20 -  63334
             3.00 -   62996          7.80 -  62668
             3.60 -   62689          8.40 -  61948
             4.20 -   62349          9.00 -  62411

          Synchronous System -- Simulation Results

      Sample set size:                                1000000
      Processing time of level 2 (times level 1):  0.55
      Average processing time (level 1):              0.47
      Average processing time (level 2):              0.26
      Average DB system response time (* t1):         2.24
      Average TB system response time (* t1):         3.35

         Distribution of level 1 response times:
                0 -   62725             8 -  62429
                1 -   62429             9 -  62470
                2 -   63000            10 -  62348
                3 -   62635            11 -  62509
                4 -   62747            12 -  62617
                5 -   62252            13 -  62215
                6 -   62702            14 -  62425
                7 -   62414            15 -  62083

         Distribution of level 2 response times:
             0.00 -   62570          4.40 -  62951
             0.55 -   62149          4.95 -  62277
             1.10 -   62603          5.50 -  62386
             1.65 -   62217          6.05 -  62541
             2.20 -   62276          6.60 -  62312
             2.75 -   62464          7.15 -  62526
             3.30 -   62866          7.70 -  63072
             3.85 -   62456          8.25 -  62334
```

```
        Synchronous System -- Simulation Results

Sample set size:                                   1000000
Processing time of level 2 (times level 1):          0.50
Average processing time (level 1)                    0.47
Average processing time (level 2)                    0.25
Average DB system response time (* t1):              2.21
Average TB system response time (* t1):              3.32

     Distribution of level 1 response times:
            0 -  62562           8 -  62326
            1 -  62529           9 -  62018
            2 -  62673          10 -  62100
            3 -  62358          11 -  62485
            4 -  63036          12 -  62492
            5 -  62981          13 -  62256
            6 -  62613          14 -  62439
            7 -  62465          15 -  62667

     Distribution of level 2 response times:
         0.00 -  62453        4.00 -  62375
         0.50 -  62586        4.50 -  62258
         1.00 -  62399        5.00 -  62501
         1.50 -  63013        5.50 -  62157
         2.00 -  62285        6.00 -  62615
         2.50 -  62390        6.50 -  62515
         3.00 -  62418        7.00 -  62643
         3.50 -  62794        7.50 -  62598
```

Results of Asynchronous System Simulation Gaussian Distribution (Both Levels)

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1):        1.00
Average processing time (level 1):                 0.44
Average processing time (level 2):                 0.50
Average size of level-level queue:               6279.71
Maximum size of level-level queue:                12493
Average system response time (times t1):         7178.86
Approximate Percent Idle time (level 2)            0.00

   Distribution of level 2 response times:
         1 -      0         9 -  24743
         2 -      0        10 -   6875
         3 -      0        11 -    758
         4 -     23        12 -     25
         5 -    620        13 -      0
         6 -   6287        14 -      0
         7 -  23522        15 -      0
         8 -  37147        16 -      0

Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1).        0.95
Average processing time (level 1):                 0.44
Average processing time (level 2):                 0.48
Average size of level-level queue:               3988.28
Maximum size of level-level queue:                 7903
Average system response time (times t1):         4332.44
Approximate Percent Idle time (level 2)            0.00

   Distribution of level 2 response times:
         1 -      0         9 -  24703
         2 -      0        10 -   6857
         3 -      0        11 -    757
         4 -     32        12 -     25
         5 -    669        13 -      0
         6 -   6245        14 -      0
         7 -  23753        15 -      0
         8 -  36959        16 -      0
```

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1):  0.90
Average processing time (level 1):               0.44
Average processing time (level 2):               0.45
Average size of level-level queue:             1432.57
Maximum size of level-level queue:               2801
Average system response time (times t1):  1475.46
Approximate Percent Idle time (level 2)        0.00

   Distribution of level 2 response times:
         1 -      0         9 -  24510
         2 -      0        10 -   7019
         3 -      0        11 -    772
         4 -     28        12 -     27
         5 -    630        13 -      1
         6 -   6371        14 -      0
         7 -  23830        15 -      0
         8 -  36812        16 -      0

Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1):  0.85
Average processing time (level 1):               0.44
Average processing time (level 2):               0.43
Average size of level-level queue:               1.00
Maximum size of level-level queue:                  7
Average system response time (times t1):        2.95
Approximate Percent Idle time (level 2)         2.92

   Distribution of level 2 response times:
         1 -      0         9 -  24832
         2 -      0        10 -   6991
         3 -      0        11 -    732
         4 -     29        12 -     29
         5 -    652        13 -      1
         6 -   6272        14 -      0
         7 -  23746        15 -      0
         8 -  36716        16 -      0
```

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                 100000
Processing time of level 2 (times level 1):        0.80
Average processing time (level 1):                 0.44
Average processing time (level 2):                 0.40
Average size of level-level queue:                 0.41
Maximum size of level-level queue:                    3
Average system response time (times t1):           2.28
Approximate Percent Idle time (level 2)            8.69

    Distribution of level 2 response times:
          1 -      0         9 -  24700
          2 -      0        10 -   6847
          3 -      0        11 -    792
          4 -     19        12 -     24
          5 -    650        13 -      0
          6 -   6257        14 -      0
          7 -  23787        15 -      0
          8 -  36924        16 -      0

Asynchronous System Simulation Results (Gaussian)

Sample set size:                                 100000
Processing time of level 2 (times level 1):        0.75
Average processing time (level 1):                 0.44
Average processing time (level 2):                 0.38
Average size of level-level queue:                 0.18
Maximum size of level-level queue:                    2
Average system response time (times t1):           2.01
Approximate Percent Idle time (level 2)           14.30

    Distribution of level 2 response times:
          1 -      0         9 -  24621
          2 -      0        10 -   7038
          3 -      0        11 -    712
          4 -     25        12 -     21
          5 -    636        13 -      0
          6 -   6242        14 -      0
          7 -  23769        15 -      0
          8 -  36936        16 -      0
```

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                    100000
Processing time of level 2 (times level 1):           0.70
Average processing time (level 1):                    0.44
Average processing time (level 2):                    0.35
Average size of level-level queue:                    0.07
Maximum size of level-level queue:                       1
Average system response time (times t1):              1.86
Approximate Percent Idle time (level 2)              20.00

   Distribution of level 2 response times:
          1 -       0        9 -  24799
          2 -       0       10 -   6774
          3 -       0       11 -    759
          4 -      24       12 -     34
          5 -     640       13 -      1
          6 -    6340       14 -      0
          7 -   23759       15 -      0
          8 -   36870       16 -      0

Asynchronous System Simulation Results (Gaussian)

Sample set size:                                    100000
Processing time of level 2 (times level 1):           0.65
Average processing time (level 1):                    0.44
Average processing time (level 2):                    0.33
Average size of level-level queue:                    0.03
Maximum size of level-level queue:                       1
Average system response time (times t1):              1.76
Approximate Percent Idle time (level 2)              25.75

   Distribution of level 2 response times:
          1 -       0        9 -  24622
          2 -       0       10 -   6779
          3 -       0       11 -    775
          4 -      28       12 -     33
          5 -     660       13 -      0
          6 -    6287       14 -      0
          7 -   23682       15 -      0
          8 -   37134       16 -      0
```

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1):         0.60
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.30
Average size of level-level queue:                  0.01
Maximum size of level-level queue:                     1
Average system response time (times t1):            1.69
Approximate Percent Idle time (level 2)            31.47

   Distribution of level 2 response times:
         1 -      0        9 -  24573
         2 -      0       10 -   6950
         3 -      0       11 -    756
         4 -     17       12 -     25
         5 -    632       13 -      0
         6 -   6423       14 -      0
         7 -  23585       15 -      0
         8 -  37039       16 -      0

Asynchronous System Simulation Results (Gaussian)

Sample set size:                                  100000
Processing time of level 2 (times level 1):         0.55
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.28
Average size of level-level queue:                  0.00
Maximum size of level-level queue:                     1
Average system response time (times t1):            1.63
Approximate Percent Idle time (level 2)            37.16

   Distribution of level 2 response times
         1 -      0        9 -  24798
         2 -      0       10 -   6967
         3 -      0       11 -    777
         4 -     23       12 -     33
         5 -    631       13 -      0
         6 -   6226       14 -      0
         7 -  23377       15 -      0
         8 -  37168       16 -      0
```

```
Asynchronous System Simulation Results (Gaussian)

Sample set size:                                100000
Processing time of level 2 (times level 1):      0.50
Average processing time (level 1):               0.44
Average processing time (level 2):               0.25
Average size of level-level queue:               0.00
Maximum size of level-level queue:                  1
Average system response time (times t1):         1.57
Approximate Percent Idle time (level 2)         42.88

   Distribution of level 2 response times:
        1 -      0         9 -  24834
        2 -      0        10 -   6753
        3 -      0        11 -    726
        4 -     23        12 -     29
        5 -    634        13 -      0
        6 -   6126        14 -      0
        7 -  23789        15 -      0
        8 -  37086        16 -      0
```

Results of Synchronous System Simulation
Gaussian Distribution (both levels)

```
Synchronous Gaussian System -- Simulation Results
Sample set size:                                  100000
Processing time of level 2 (times level 1):         1.00
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.44
Average DB system response time (* t1):             2.17
Average TB system response time (* t1):             3.25
   Distribution of level 1 response times:
         0 -        0             8 -  24749
         1 -        0             9 -   6850
         2 -        0            10 -    786
         3 -       20            11 -     32
         4 -      673            12 -      0
         5 -     6335            13 -      0
         6 -    23588            14 -      0
         7 -    36967            15 -      0
   Distribution of level 2 response times.
      0.00 -        0          8.00 -  24531
      1.00 -        0          9.00 -   6943
      2.00 -        0         10.00 -    695
      3.00 -       21         11.00 -     27
      4.00 -      651         12.00 -      0
      5.00 -     6251         13.00 -      0
      6.00 -    23641         14.00 -      0
      7.00 -    37240         15.00 -      0


Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1).         0.95
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.42
Average DB system response time (* t1):             2.13
Average TB system response time (* t1):             3.19


   Distribution of level 1 response times:
         0 -        0             8 -  24966
         1 -        0             9 -   6866
         2 -        0            10 -    682
         3 -       20            11 -     20
         4 -      635            12 -      0
         5 -     6442            13 -      0
         6 -    23574            14 -      0
         7 -    36795            15 -      0


   Distribution of level 2 response times
      0.00 -        0          7.60 -  24461
      0.95 -        0          8.55 -   7009
      1.90 -        0          9.50 -    774
      2.85 -       28         10.45 -     24
      3.80 -      621         11.40 -      0
      4.75 -     6246         12.35 -      0
      5.70 -    23636         13.30 -      0
      6.65 -    37201         14.25 -      0
```

```
Synchronous Gaussian System -- Simulation Results

Sample set size                                 100000
Processing time of level 2 (times level 1):      0.90
Average processing time (level 1):               0.44
Average processing time (level 2):               0.40
Average DB system response time (* t1):          2.08
Average TB system response time (* t1):          3.13

    Distribution of level 1 response times:
          0 -      0           8 -  24470
          1 -      0           9 -   6930
          2 -      0          10 -    761
          3 -     18          11 -     37
          4 -    695          12 -      0
          5 -   6283          13 -      0
          6 -  23527          14 -      0
          7 -  37279          15 -      0

    Distribution of level 2 response times:
       0.00 -      0        7.20 -  24830
       0.90 -      0        8.10 -   6899
       1.80 -      0        9.00 -    754
       2.70 -     28        9.90 -     31
       3.60 -    667       10.80 -      1
       4.50 -   6305       11.70 -      0
       5.40 -  23770       12.60 -      0
       6.30 -  36715       13.50 -      0

Synchronous Gaussian System -- Simulation Results

Sample set size                                 100000
Processing time of level 2 (times level 1):      0.85
Average processing time (level 1):               0.44
Average processing time (level 2):               0.37
Average DB system response time (* t1):          2.05
Average TB system response time (* t1):          3.08

    Distribution of level 1 response times:
          0 -      0           8 -  24784
          1 -      0           9 -   6885
          2 -      1          10 -    713
          3 -     21          11 -     32
          4 -    686          12 -      0
          5 -   6295          13 -      0
          6 -  23724          14 -      0
          7 -  36859          15 -      0

    Distribution of level 2 response times:
       0.00 -      0        6.80 -  24793
       0.85 -      0        7.65 -   6972
       1.70 -      0        8.50 -    730
       2.55 -     20        9.35 -     26
       3.40 -    660       10.20 -      0
       4.25 -   6233       11.05 -      0
       5.10 -  23732       11.90 -      0
       5.95 -  36834       12.75 -      0
```

```
Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1):         0.80
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.35
Average DB system response time (* t1):             2.03
Average TB system response time (* t1):             3.05

   Distribution of level 1 response times:
         0 -       0             8 -  24558
         1 -       0             9 -   6827
         2 -       0            10 -    791
         3 -      22            11 -     28
         4 -     687            12 -      0
         5 -    6395            13 -      0
         6 -   23484            14 -      0
         7 -   37208            15 -      0

   Distribution of level 2 response times:
      0.00 -       0          6.40 -  24872
      0.80 -       0          7.20 -   6804
      1.60 -       0          8.00 -    762
      2.40 -      16          8.80 -     24
      3.20 -     641          9.60 -      0
      4.00 -    6249         10.40 -      0
      4.80 -   23388         11.20 -      0
      5.60 -   37244         12.00 -      0

Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1):         0.75
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.33
Average DB system response time (* t1):             2.02
Average TB system response time (* t1):             3.02

   Distribution of level 1 response times:
         0 -       0             8 -  24654
         1 -       0             9 -   6876
         2 -       1            10 -    728
         3 -      24            11 -     26
         4 -     623            12 -      1
         5 -    6283            13 -      0
         6 -   23691            14 -      0
         7 -   37093            15 -      0

   Distribution of level 2 response times:
      0.00 -       0          6.00 -  24758
      0.75 -       0          6.75 -   6873
      1.50 -       0          7.50 -    764
      2.25 -      35          8.25 -     39
      3.00 -     684          9.00 -      1
      3.75 -    6295          9.75 -      0
      4.50 -   23733         10.50 -      0
      5.25 -   36818         11.25 -      0
```

```
Synchronous Gaussian System -- Simulation Results

Sample set size:                                100000
Processing time of level 2 (times level 1):  0.70
Average processing time (level 1):             0.44
Average processing time (level 2):             0.31
Average DB system response time (* t1):        2.01
Average TB system response time (* t1):        3.01

   Distribution of level 1 response times:
         0 -      0           8 -  24656
         1 -      0           9 -   6976
         2 -      0          10 -    754
         3 -     21          11 -     18
         4 -    673          12 -      0
         5 -   6296          13 -      0
         6 -  23666          14 -      0
         7 -  36940          15 -      0

   Distribution of level 2 response times:
      0.00 -      0        5.60 -  24668
      0.70 -      0        6.30 -   6824
      1.40 -      0        7.00 -    752
      2.10 -     22        7.70 -     31
      2.80 -    662        8.40 -      0
      3.50 -   6213        9.10 -      0
      4.20 -  23628        9.80 -      0
      4.90 -  37200       10.50 -      0

Synchronous Gaussian System -- Simulation Results

Sample set size:                                100000
Processing time of level 2 (times level 1):  0.65
Average processing time (level 1):             0.44
Average processing time (level 2):             0.29
Average DB system response time (* t1):        2.00
Average TB system response time (* t1):        3.01

   Distribution of level 1 response times:
         0 -      0           8 -  24689
         1 -      0           9 -   7066
         2 -      1          10 -    743
         3 -     20          11 -     23
         4 -    649          12 -      0
         5 -   6360          13 -      0
         6 -  23658          14 -      0
         7 -  36791          15 -      0

   Distribution of level 2 response times:
      0.00 -      0        5.20 -  24566
      0.65 -      0        5.85 -   6914
      1.30 -      0        6.50 -    728
      1.95 -     22        7.15 -     28
      2.60 -    667        7.80 -      0
      3.25 -   6335        8.45 -      0
      3.90 -  23650        9.10 -      0
      4.55 -  37090        9.75 -      0
```

```
Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1):  0.60
Average processing time (level 1):                0.44
Average processing time (level 2):                0.26
Average DB system response time (* t1):           2.00
Average TB system response time (* t1):           3.00

   Distribution of level 1 response times:
         0 -       0           8 -  24796
         1 -       0           9 -   6838
         2 -       1          10 -    734
         3 -      23          11 -     27
         4 -     688          12 -      0
         5 -    6431          13 -      0
         6 -   23463          14 -      0
         7 -   36999          15 -      0

   Distribution of level 2 response times:
      0.00 -       0        4.80 -  24517
      0.60 -       0        5.40 -   6912
      1.20 -       0        6.00 -    756
      1.80 -      22        6.60 -     33
      2.40 -     679        7.20 -      0
      3.00 -    6285        7.80 -      0
      3.60 -   23698        8.40 -      0
      4.20 -   37098        9.00 -      0

Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1):  0.55
Average processing time (level 1):                0.44
Average processing time (level 2):                0.24
Average DB system response time (* t1):           2.00
Average TB system response time (* t1):           3.00

   Distribution of level 1 response times:
         0 -       0           8 -  24602
         1 -       0           9 -   6782
         2 -       0          10 -    748
         3 -      25          11 -     34
         4 -     719          12 -      1
         5 -    6350          13 -      0
         6 -   23693          14 -      0
         7 -   37046          15 -      0

   Distribution of level 2 response times:
      0.00 -       0        4.40 -  24589
      0.55 -       0        4.95 -   6904
      1.10 -       0        5.50 -    765
      1.65 -      20        6.05 -     28
      2.20 -     632        6.60 -      0
      2.75 -    6270        7.15 -      0
      3.30 -   23596        7.70 -      0
      3.85 -   37196        8.25 -      0
```

```
Synchronous Gaussian System -- Simulation Results

Sample set size:                                  100000
Processing time of level 2 (times level 1):         0.50
Average processing time (level 1):                  0.44
Average processing time (level 2):                  0.22
Average DB system response time (* t1):             2.00
Average TB system response time (* t1):             3.00

   Distribution of level 1 response times:
         0 -      0            8 -  24718
         1 -      0            9 -   6874
         2 -      1           10 -    745
         3 -     23           11 -     33
         4 -    632           12 -      0
         5 -   6308           13 -      0
         6 -  23844           14 -      0
         7 -  36822           15 -      0

   Distribution of level 2 response times:
      0.00 -      0         4.00 -  24577
      0.50 -      0         4.50 -   7069
      1.00 -      0         5.00 -    777
      1.50 -     17         5.50 -     29
      2.00 -    662         6.00 -      0
      2.50 -   6372         6.50 -      0
      3.00 -  23504         7.00 -      0
      3.50 -  36993         7.50 -      0
```

VITA

VITA

Bradley Warren Smith was born in Euclid, Ohio on November 27, 1958. He was granted a high school diploma from Hawken school in 1976. He received the B.S. degree in 1978, the M.S. degree in 1980, and the Ph.D. degree in 1985, all from Purdue University in West Lafayette, Indiana. As a graduate student, he was a teaching assistant for Purdue's School of Electrical Engineering, developing hardware laboratories and instructing a junior level programming course. He has also been a programmer for the Engineering Computer Network, developing applications software. As a graduate research assistant, his studies included parallel/distributed processing system architecture, architectures for image and speech processing, and models for use in the design of macro-pipelined parallel processors. He is a member of Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Sigma Xi honoraries and the IEEE Computer Society.

END

DTIC

6-86