PL-TR-96-1126, Pt. 2

PL-TR-96-1126, Pt 2

ADVANCED HARD REAL-TIME OPERATING SYSTEM, THE MARUTI PROJECT

Part 2 of 2

Ashok K. Agrawala
Satish K. Tripathi

Department of Computer Science
University of Maryland
College Park, MD 20742

January 1997

Final Report

PHILLIPS LABORATORY
Space Technology Directorate
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

PL-TR-96-1126

This report has been approved for publication.

[illegible]

JIM RUSSELL, Capt, USAF
Project Manager

FOR THE COMMANDER

[illegible]

NANCY L. CROWLEY, Lt Col, USAF
Chief, Satellite Control and Simulation Division

[illegible]

CHRISTINE M. ANDERSON, SES
Director, Space Technology

DRAFT SF 298

1. Report Date (dd-mm-yy) January 1997	2. Report Type Final	3. Dates covered (from... to) 4/92 to 10/96

4. Title & subtitle Advanced Hard Real-Time Operating System, The Maruti Project	5a. Contract or Grant # DASG-60-92-C-0055
	5b. Program Element # 62301E
6. Author(s) Ashok K. Agrawala Satish K. Tripathi	5c. Project # DRPB
	5d. Task # TB
	5e. Work Unit # AT
7. Performing Organization Name & Address Department of Computer Science University of Maryland College Park, MD 20742	8. Performing Organization Report #
9. Sponsoring/Monitoring Agency Name & Address Phillips Laboratory 3550 Aberdeen Ave. SE Kirtland, AFB, NM 87117-5776	10. Monitor Acronym
	11. Monitor Report # PL-TR-96-1126, Part 2

12. Distribution/Availability Statement
Further dissemination only as directed by the U.S. Army Strategic Defense Command, ATTN: SCCD-IM-PA, P.O. Box 1500, Huntsville, AL 35807-3801, January 1997, or higher DoD authority.

13. Supplementary Notes

14. Abstract System correctness for real-time systems relies on both functional and temporal correctness of the system components. In order to allow creation and deployment of critical applications with hard real-time constraints in a reactive environment, we have developed the Maruti environment, which consists of the Maruti operating system and runtime environment, and an application development and environment that uses the Maruti Programming Language (MPL), an extension of ANSI C; the Maruti Configuration language (MCL), which specifies how MPL modules are to be connected and any environmental constraints; and various analysis and debugging tools. The core of the Maruti runtime system is the Elemental Unit (EU) and calendar. An EU is an atomic entity triggered by incoming data/signals, that produces data/signals. A calendar specifies the execution order and time for each EU. Calendars are static entities created during application design and development, thus allowing temporal debugging of applications before they are executed on the machine. A given application may have more than one calendar to allow contingency or degraded operation.

15. Subject Terms
Real-Time operating systems, fault tolerance, concurrency, embedded systems, environments

Security Classification of					
16. Report Unclassified	17. Abstract Unclassified	18. This Page Unclassified	19. Limitation of Abstract Limited	20. # of Pages 200	21. Responsible Person (Name and Telephone #) Capt Jim Russell (505) 846-8986 ext 352

Optimization in Non-Preemptive Scheduling for Aperiodic Tasks [*]

Shyh-In Hwang Sheng-Tzong Cheng
Ashok K. Agrawala
Institute for Advanced Computer Studies
and
Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Real-time computer systems have become more and more important in many applications, such as robot control, flight control, and other mission-critical jobs. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. We propose a scheduling algorithm based on an analytic model. Our goal is to derive the optimal schedule for a given set of aperiodic tasks such that the number of rejected tasks is minimized, and then the finish time of the schedule is also minimized. The scheduling problem with a nonpreemptive discipline in a uniprocessor system is considered. We first show that if a total ordering is given, this can be done in $O(n^2)$ time by dynamic programming technique, where n is the size of the task set. When the restriction of the total ordering is released, it is known

[*]This work is supported in part by Honeywell under N00014-91-C-0195 and Army/Phillips under DASG-60-92-C-0055. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Honeywell or Army/Phillips.

to be NP-complete [3]. We discuss the super sequence [18] which has been shown to be useful in reducing the search space for testing the feasibility of a task set. By extending the idea and introducing the concept of conformation, the scheduling process can be divided into two phases: computing the pruned search space, and computing the optimal schedule for each sequence in the search space. While the complexity of the algorithm in the worst case remains exponential, our simulation results show that the cost is reasonable for the average case.

1 Introduction

In a hard real-time system, the computer is required to support the execution of applications in which the timing constraints of the tasks are specified. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. Failure to satisfy the timing constraints can incur fatal errors. Once a task is accepted by the system, the system should be able to finish it under the timing constraint of the task. A task T_i can be characterized as a triple of (r_i, c_i, d_i), representing the ready time, the computation time, and the deadline of the task, respectively. A task can not be started before its ready time. Once started, the task must use the processor for a consecutive period of c_i, and be finished by its deadline. The task set is represented as $\Gamma = \{T_1, T_2, ..., T_n\}$. A task set is *feasible* if there exists a *schedule* in which all the tasks in the task set can meet their timing constraints. Scheduling is a process of binding starting times to the tasks such that each task executes according to the schedule. A sequence $S = \langle T_1^S, T_2^S, \ldots, T_k^S \rangle$, where $k \leq n$. T_i^S represents the ith task of the sequence S for any $1 \leq i \leq k$. A sequence specifies the order in which the tasks are executed. Without confusion, a schedule can be represented as a sequence. How to schedule the tasks so that the timing constraints are met is nontrivial. Many scheduling problems are known to be intractable [3] in that finding the optimal schedule requires large amounts of computations to be carried out.

The approaches adopted to date for scheduling algorithms can be generally classified into two categories. One approach is to assign priorities to tasks so that the tasks can be scheduled according to their priorities [1, 7, 8, 10, 12, 15, 14]. This approach is called *priority based scheduling*. The priority can be determined by deadline, execution time, resource requirement, laxity, period, or can be programmer-defined [4]. The other is *time based scheduling* approach [9, 13]. A time based scheduler generates as an output a *calendar* which specifies the time instants at which the tasks start and finish.

Generally speaking, scheduling for aperiodic task sets without preemption is NP-complete [3]. Due to the intractability, several search algorithms [11, 17, 19, 20] are proposed for computing optimal or suboptimal schedules. Analytic techniques may also be used for optimal scheduling. A dominance concept by Erschler *et al* [2] was proposed to reduce the search space for checking the feasibility of task sets. They explored the relations among the tasks and determined the partial orderings of feasible schedules. Yuan and Agrawala [18] proposed decomposition methods to substantially reduce the search space based on the dominance concept. A task set is decomposed into subsets so that each subset can be scheduled independently. A super sequence is constructed to reduce search space further. Saksena and Agrawala [13] investigated the technique of temporal analysis serving as a pre-processing stage for scheduling. The idea is to modify the windows of two partially ordered tasks which are generated by the temporal relations so that more partial orderings of tasks may be generated recursively.

The time based model is employed by several real-time operating systems currently being developed, including MARUTI [5], MARS [6], and Spring [16]. In this paper, we study an analytic approach to optimal scheduling under the time based model. When complicated timing constraints and task interdependency are taken into consideration, the schedulability analysis of priority based scheduling algorithms becomes much more difficult. By analytic approach, we believe that the time based scheduling algorithm and analysis require reasonable amounts of computations to produce a feasible schedule.

The rest of this paper is organized as follows. In section 2, we describe how to compute the optimal schedule for a sequence. In section 3, releasing the restriction of total ordering on a sequence, we present the approach to computing the optimal schedule for a task set. Related theorems are also presented. In section 4, a simulation experiment is conducted to compare the performance of different algorithms. The last section is our conclusions.

2 Scheduling a Sequence

The *size* of a sequence (task set) is the number of tasks in the sequence(task set), and is denoted by $|S|$ ($|\Gamma|$). A sequence S is *feasible* if all tasks in S are executed in the order of the sequence and the timing constraints are satisfied. For convenience, we further define an *instance*, I, to be a sequence such that $|I| = |\Gamma|$. We denote the instance I by

$$I = \langle T_1^I, T_2^I, \ldots, T_n^I \rangle.$$

Notice that $\{\}$ is used to represent a task set, and $\langle\rangle$ a sequence. Let T_i and T_j be two tasks belonging to sequence S. If T_i is located before T_j in the sequence S, we say that

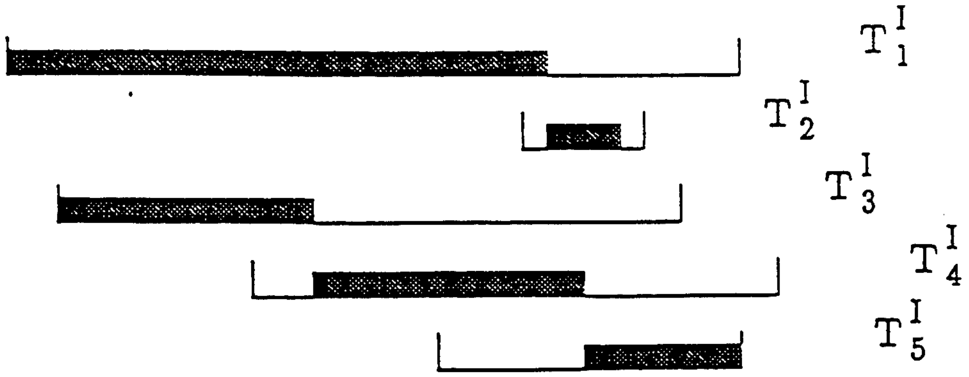


Figure 1: An instance $I = \langle T_1^I, T_2^I, \ldots, T_5^I \rangle$

$\langle T_i, T_j \rangle$ *conforms to* S. A sequence $S1$ *conforms to* a sequence $S2$, if, for any T_i and T_j, $\langle T_i, T_j \rangle$ conforming to $S1$ implies $\langle T_i, T_j \rangle$ conforming to $S2$. We use $\sigma(k)$ to represent the *optimal schedule* of $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ in the sense that for any feasible sequence S conforming to $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$, either

$$|S| < |\sigma(k)|,$$

or

$$|S| = |\sigma(k)| \qquad \textit{and} \qquad f_S \geq f_{\sigma(k)}, \tag{1}$$

where f_S and $f_{\sigma(k)}$ is the finish time of S and $\sigma(k)$ respectively. $\sigma(k)$ is thus the optimal schedule for the first k tasks of I. The optimal schedule for an instance I can thus be represented by $\sigma(n)$. For simplicity, let $u_k = |\sigma(k)|$. In this section, we will discuss the scheduling for an instance. However, the approach is generally applicable to any sequence.

2.1 Preliminary

We assume that $r_i + c_i \leq d_i$ holds for each task T_i in the task set Γ. At the first glance, one may attempt to compute $\sigma(k)$ based on $\sigma(k-1)$. However, with careful examination, we can find that merely computing $\sigma(k-1)$ does not suffice to compute $\sigma(k)$. This is illustrated by the example in Figure 1. From this example, we can obtain

$$\sigma(1) = \langle T_1^I \rangle$$

$$\sigma(2) = \langle T_1^I, T_2^I \rangle.$$

At the next step, $\sigma(2) \oplus \langle T_3^I \rangle$ is not feasible, where the operator $\oplus$ means concatenation of two sequences. One task must be rejected, which is T_3^I in this case. Hence, we got

$$\sigma(3) = \sigma(2) = \langle T_1^I, T_2^I \rangle.$$

A problem arises at the next step. $\sigma(3) \oplus \langle T_4^I \rangle$ is not feasible either. If we try to fix it by taking a task off this sequence, the result is

$$\sigma'(4) = \sigma(3) = \langle T_1^I, T_2^I \rangle.$$

However, the correct result should be

$$\sigma(4) = \langle T_3^I, T_4^I \rangle.$$

Although both $\sigma'(4)$ and $\sigma(4)$ are of the same size, the latter comes with a shorter finish time, which becomes significant at next step. We get

$$\sigma(n) = \sigma(5) = \sigma(4) \oplus \langle T_5^I \rangle = \langle T_3^I, T_4^I, T_5^I \rangle.$$

However, with $\sigma'(4)$, we would have

$$\sigma'(5) = \sigma'(4) = \langle T_1^I, T_2^I \rangle.$$

This example shows that merely computing $\sigma(k-1)$ does not suffice to compute $\sigma(k)$. When $\sigma(k-1)$ is obtained, it can not be predicted as to which tasks would be included in $\sigma(k)$. The approach has to be modified as follows.

2.2 Sequence-Scheduler Algorithm

We denote by $S(k,j)$ the sequence such that $S(k,j)$ conforms to $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ and $|S(k,j)| = j$, where $j \leq |\sigma(k)|$. $S(k,j)$ represents any sequence of j tasks picked up from the first k tasks of S. We further define a sequence, denoted by $\sigma(k,j)$, to be the *optimal schedule with degree* j for $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ in the sense that for any feasible sequence $S(k,j)$, we have

$$f_{\sigma(k,j)} \leq f_{S(k,j)}.$$

Notice that $\sigma(k)$ is an abbreviation of $\sigma(k,\ u_k)$. If a sequence $S(k,j)$ is not feasible, $f_{S(k,j)} = \infty$.

We would like to compute $\sigma(k,j)$ based on $\sigma(k-1,j')$, where $j' \leq j \leq |\sigma(k)|$. The basic idea is as follows. We know $\sigma(k,j)$ either contains T_k^I or not. If so, then the other $j-1$ tasks are picked up from the first $k-1$ tasks, and $\sigma(k-1,j-1)$ is one of the best choices. In this case, $\sigma(k,j) = \sigma(k-1,j-1) \oplus T_k^I$. If $\sigma(k,j)$ does not contain T_k^I, all of the j tasks should be picked up from the first $k-1$ tasks, and $\sigma(k-1,j)$ is one of the best choices. In this case, $\sigma(k,j) = \sigma(k-1,j)$. Whether taking T_k^I or not is determined by comparing which one of the sequences comes with a shorter finish time. Therefore, $\sigma(k,j)$ can be determined by $\sigma(k-1,j-1)$, and $\sigma(k-1,j)$. The computation of $\sigma(k-1,j)$ is in turn based on $\sigma(k-2,j-1)$, and $\sigma(k-2,j)$. In general, at each step k, we need to compute $\sigma(k,j)$ for $j = 1, 2, \ldots, |\sigma(k)|$. The algorithm *Sequence-Scheduler* in Figure 2 formalizes this idea. It is worth mentioning that the condition of the "while" statement in the algorithm is designed to let j increase from 1 through $|\sigma(k)|$. The correctness is verified in the next section.

2.3 Verification of Sequence-Scheduler Algorithm

The proof of the correctness of the algorithm along with some related lemmas are given below.

Lemma 1 Let $S1$ and $S2$ be two sequences such that $f_{S1} \leq f_{S2}$. If $S2 \oplus \langle T_x \rangle$ is feasible, then $f_{S1 \oplus \langle T_x \rangle} \leq f_{S2 \oplus \langle T_x \rangle}$.

Algorithm Sequence-Scheduler:

Input: an instance $I = \langle T_1^I, T_2^I, \ldots, T_n^I \rangle$
Output: the optimal schedule $\sigma(n) \equiv \sigma(n, u_n)$ for I

```
$\sigma(0,0) := \langle\rangle$; $u_0 = 0$
for $k := 1, 2, \ldots, n$
    $j := 1$
    while $(j \leq u_{k-1})$ or $((j = u_{k-1} + 1)$ and $(\sigma(k-1, u_{k-1}) \oplus \langle T_k^I \rangle$ is feasible$))$
        if $f_{\sigma(k-1,j-1)\oplus\langle T_k^I \rangle} < f_{\sigma(k-1,j)}$
            $\sigma(k,j) := \sigma(k-1, j-1) \oplus \langle T_k^I \rangle$
        else
            $\sigma(k,j) := \sigma(k-1, j)$
        endif
        $j := j + 1$
    endwhile
    $u_k := j - 1$
endfor
```

Figure 2: Sequence-Scheduler Algorithm

PROOF: This is straightforward via the following equations.

$$\begin{aligned} f_{S1\oplus\langle T_x\rangle} &= max(f_{S1}, r_{T_x}) + c_{T_x} \\ &\leq max(f_{S2}, r_{T_x}) + c_{T_x} \\ &= f_{S2\oplus\langle T_x\rangle} \end{aligned}$$

□

Corollary 1 Let $S1$ and $S2$ be two sequences such that $f_{S1} \leq f_{S2}$. If $S2 \oplus S3$ is feasible, where $S3$ is another sequence, then $f_{S1\oplus S3} \leq f_{S2\oplus S3}$.

PROOF: This is a direct result of applying Lemma 1 repeatedly through the tasks in $S3$. □

Lemma 2 $u_k = u_{k-1}$ or $u_k = u_{k-1} + 1$.

PROOF: It is obvious that $u_k \geq u_{k-1}$, where both u_k and u_{k-1} are integers. Let us assume that $u_k = u_{k-1} + \alpha$, and $\alpha \geq 2$. We are going to show that this assumption does not hold. We know $\sigma(k, u_k)$ either contains T_k^I or not. If $\sigma(k, u_k)$ contains T_k^I, we can represent $\sigma(k, u_k)$ as $S(k-1, u_k - 1) \oplus \langle T_k^I\rangle$, by picking up a proper sequence $S(k-1, u_k - 1)$. However, from the assumption above, we have $u_{k-1} = u_k - \alpha < u_k - 1 = |S(k-1, u_k - 1)|$. This contradicts the definition of u_{k-1}. On the other hand, if $\sigma(k, u_k)$ does not contain T_k^I, we can represent $\sigma(k, u_k)$ as $S(k-1, u_k)$. We have $u_{k-1} = u_k - \alpha < u_k$, which is a contradiction. The assumption thus does not hold. Therefore, we have $\alpha \leq 1$. □

From this lemma, $\sigma(k, j)$ does exist for $j \leq u_{k-1}$. Furthermore, in the algorithm, $j = u_{k-1} + 1$ is tested to see if $u_k = u_{k-1} + 1$.

Theorem 1 For $k = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, u_k$, if $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$, then $\sigma(k, j) = \sigma(k-1, j-1) \oplus \langle T_k^I\rangle$; otherwise, $\sigma(k, j) = \sigma(k-1, j)$.

PROOF: The proof is by induction on k. When $k = 1$, $I_1 = \langle T_1^I\rangle$. Since $u_1 \leq 1$ and $\langle T_1^I\rangle$ is feasible, $\sigma(1, 1) = \langle T_1^I\rangle$. It is easy to come up with the same result through this theorem. Thus holds the base case. We assume that we can compute $\sigma(k-1, j)$, for $j = 1, 2, \ldots, u_{k-1}$, in the same way, and consider the case of k. Let us first bring forward three basic equations. Since $f_{\sigma(k-1,j-1)} \leq f_{S(k-1,j-1)}$, the following equation holds by Lemma 1

$$f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}. \tag{2}$$

By induction hypothesis on $\sigma(k-1,j)$ such that $f_{\sigma(k-1,j)} \leq f_{S(k-1,j)}$, we have

$$if \quad f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}, \quad then\ f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{S(k-1,j)}. \tag{3}$$

From Equation 2, we have

$$if \quad f_{\sigma(k-1,j)} \leq f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle}, \quad then\ f_{\sigma(k-1,j)} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}. \tag{4}$$

From Lemma 2, we know either $u_k= u_{k-1}+1$ or $u_k= u_{k-1}$. The two cases are discussed below.

Case I: $u_k= u_{k-1}+1$. We first discuss the situation when $j = 1,2,\ldots,u_k - 1$. We know that a feasible sequence $S(k,j)$ is either in the form of $S(k-1,j)$ or $S(k-1,j-1) \oplus \langle T_k^I\rangle$. If $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$, then $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{S(k-1,j)}$ by Equation 3. Also we have $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$ by Equation 2. This means $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k,j)}$ for any feasible sequence $S(k,j)$. Consequently, $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$, which justifies the theorem. On the other hand, if $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \geq f_{\sigma(k-1,j)}$, then $f_{\sigma(k-1,j)} < f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$ by Equation 4. In addition, $f_{\sigma(k-1,j)} \leq f_{S(k-1,j)}$ by induction hypothesis on $\sigma(k-1,j)$. So $f_{\sigma(k-1,j)} \leq f_{S(k,j)}$ for any feasible sequences $S(k,j)$. In this case, $\sigma(k,j) = \sigma(k-1,j)$, which justifies the theorem.

Then we discuss the situation when $j = u_k$. Since $u_k= u_{k-1}+1$, it is clear that $\langle T_k^I\rangle$ belongs to $\sigma(k)$; otherwise, we need to pick up $u_{k-1}+1$ tasks from I_{k-1} to make a feasible sequence, which violates the definition of u_{k-1}. Therefore, $\sigma(k,j)$ can be expressed as $S(k-1, u_{k-1}) \oplus \langle T_k^I\rangle$ by picking up a proper sequence $S(k-1, u_{k-1})$. Note that $u_{k-1}= j-1$ here. By Equation 2, we have $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$, for any sequence $S(k-1,j-1)\oplus\langle T_k^I\rangle$. Thus, $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$. Now let us check the theorem. The sequence $\sigma(k-1,j) = \sigma(k-1, u_{k-1}+1)$ is not feasible; thus its finish time is ∞. The condition $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$ is satisfied. So $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$. This justifies the theorem.

Case II: $u_k= u_{k-1}$. The reasoning follows the discussion of the first part in Case I. □

3 Scheduling a Task Set

In this section we discuss how to schedule a task set by using Sequence-Scheduler. The *optimal schedule, ρ,* of a task set is defined as follows: for any feasible sequence S consisting of tasks in Γ, we have either

$$|S| < |\rho|,$$

or

$$|S| = |\rho| \qquad and \qquad f_S \geq f_\rho.$$

For simplicity, we use optimal schedule to represent the optimal schedule of the task set, when there is no confusion. Note that the optimal schedule of the task set is the *best* one of the optimal schedules of all instances in the task set. Erschler *et al* [2] proposed the dominance concept to reduce the number of permutations that should be examined for the feasibility test of a task set. Yuan and Agrawala [18] proposed the super sequence to further reduce the search space for testing the feasibility of a task set. In this section, we show that for our optimization problem, the super sequence provides a valid and pruned search space. In other words, there exists one optimal schedule which conforms to an instance in the super sequence of the task set. Thus we may use Sequence-Scheduler to schedule for the instances extracted from the super sequence to derive the optimal schedule. There may exist more than one optimal schedule for a task set. Our interest is on how to derive one of them.

3.1 Super Sequence

Temporal relations between two tasks T_i and T_j are summarized in the following. They are illustrated by Figure 3.

- leading : $T_i \prec T_j$, if $r_i \leq r_j, d_i \leq d_j$, but both of the equalities do not hold at the same time.
- matching : $T_i \parallel T_j$, if $r_i = r_j, d_i = d_j$.
- containing : $T_i \sqcup T_j$, if $r_i < r_j, d_i > d_j$.

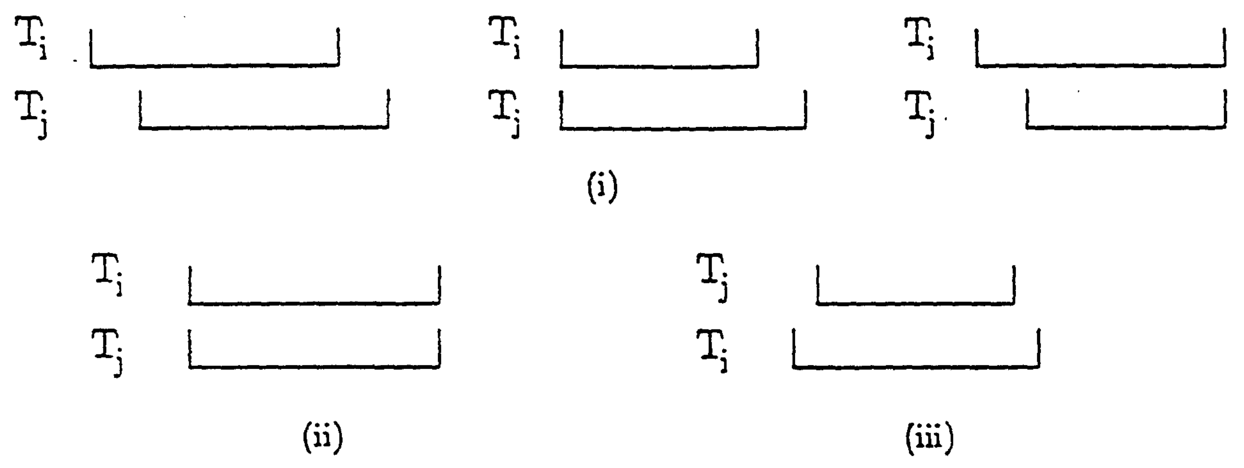


Figure 3: *(i)* $T_i \prec T_j$; *(ii)* $T_i \parallel T_j$; *(iii)* $T_i \sqcup T_j$

A task h is called a *top task* if there is no task contained by h. A task is called a *nontop task* if it contains at least one task. Assume that we have t top tasks in the task set, denoted by $h_1, h_2, \ldots, h_t$ respectively. Denote by M_k the set of tasks that contain the top task h_k, including h_k, and by $\overline{M_k}$ the set of tasks in the task set Γ that do not belong to M_k. We say that T_i is *weakly leading* to T_j, denoted by $T_i \lhd T_j$, if $T_i \prec T_j$ or $T_i \parallel T_j$. If $T_i \lhd T_j$ for all T_j belonging to S, then $T_i \lhd S$.

The dominance concept is originally developed by Erschler *et al* [2] to reduce the search space for testing the feasibility of a task set. The idea is extended with the super sequence proposed by Yuan and Agrawala [18]. An instance I *dominates* an instance I' iff:

$$I' \; feasible \Rightarrow I \; feasible.$$

It can be considered that I is a better candidate as a feasible schedule than I'. A *dominant instance* is an instance such that for each possible instance I of the task set, if I dominates the dominant instance, then the dominant instance dominates I. Thus the dominant instance can be considered as the best candidate of the feasible schedule. A set of instances is said to be a *dominant set*, if I does not belong to the dominant set, then there exists a dominant instance in the dominant set such that the dominant instance dominates I.

A super sequence Δ serves similarly as a dominant set in that there exists a dominant instance in the super sequence; and it is more appropriate for solving our problem. A super sequence is a sequence of tasks, where duplicates of tasks are allowed. The purpose is to extract instances from the super sequence for scheduling. The super sequence is constructed according to the dominant rules [2, 18] described below. Whenever a task satisfies one of the conditions specified by the rules, a duplicate of the task is inserted into the super sequence. Note that duplicates can only be generated for nontop tasks. The top tasks appear once and only once in the super sequence.

Rule R1: Let T_α and T_β be any two top tasks. If $T_\alpha \prec T_\beta$, then T_α is positioned before T_β. If $T_\alpha \parallel T_\beta$, the order of the two top tasks is determined arbitrarily.

A unique order of the top tasks can be thus determined for the super sequence. Let us denote the sequence composed of the top tasks by $H = \langle h_1, h_2, \ldots, h_t \rangle$. The rule implies that if T_α is positioned before T_β in the super sequence, then $T_\alpha \lhd T_\beta$. So $h_1 \lhd h_2 \lhd \ldots \lhd h_t$.

Rule R2:
(1) A nontop task can be positioned before the first top task h_1 only when it contains h_1.
(2) A nontop task can be positioned after the last top task h_t only when it contains h_t.
(3) A nontop task can be positioned between h_k and h_{k+1} only when it contains h_k or h_{k+1}.

The t top tasks delimit the super sequence into $t+1$ regions by rule $R1$. Now we have $t+1$ subsets of nontop tasks separated by the t top tasks by rule $R2$. Generally speaking, a nontop task has more than one possible location. Denote the kth subset by A_k, which is between top tasks h_k and h_{k+1}. From rule $R2$, it can be deduced that

$$A_k = B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}, \text{ where} \tag{5}$$
$$B_{k,\overline{k+1}} = M_k \cap \overline{M_{k+1}},\ B_{k,k+1} = M_k \cap M_{k+1},\ B_{\overline{k},k+1} = \overline{M_k} \cap M_{k+1}.$$

Next rule is to specify the order of the tasks within each subset.

Rule R3: In each subset A_k, for $k = 0, 1, \ldots, n$,
(1) the tasks in $B_{k,\overline{k+1}}$ are ordered according to their deadlines, and tasks with the same deadlines are ordered arbitrarily,
(2) the tasks in $B_{k,k+1}$ are ordered arbitrarily,
(3) the tasks in $B_{\bar{k},k+1}$ are ordered according to their ready times, and tasks with the same ready times are ordered arbitrarily,
(4) the tasks in $B_{k,\overline{k+1}}$ are positioned before those in $B_{k,k+1}$, which in turn are positioned before those in $B_{\bar{k},k+1}$.

Now we are ready to construct the super sequence with these three rules. Top tasks are first picked out and ordered, forming $t + 1$ regions. In each region, there is a subsequence of nontop tasks. An instance extracted out of the super sequence is one that conforms to the super sequence without duplication of tasks. Let q be the number of top tasks that a nontop task contains. The number of possible regions the nontop task can fall into is $q + 1$. The number of instances in the super sequence thus sums up to

$$N = \prod_{q=1}^{q=t} (q + 1)^{n_q},$$

where n_q is the number of nontop tasks which contains q top tasks. Compared with an exhaustive search which takes up to $n!$ instances (permutations) into account, the super sequence generally leads to a smaller set. Notice that it takes $O(n)$ time to check if an instance is feasible. Hence, the time complexity of the feasibility test for the task set is $O(N * n)$.

3.2 Leading Theorem

The super sequence is not only useful in testing the feasibility of a task set; we will show that it is also useful in reducing the number of instances to be examined in order to obtain the optimal schedule of a task set. We will show that there exists at least one optimal schedule which conforms to an instance in the super sequence Δ. Hence, it suffices to check through Δ to obtain the optimal schedule of Γ.

It is worth attention that the top tasks in ρ may *not* be the same top tasks of Γ. This arises because some of the top tasks of Γ may be rejected, introducing new top tasks in ρ. Before proceeding to verify the rules for the super sequence, we will first introduce the *Leading Theorem.* It serves as the base for further analysis in the Dominance Theorem and Conformation Theorem to be described later. The Leading Theorem tells that under certain condition we can adjust the order of tasks to satisfy the Weakly Leading Condition to be defined below and do not introduce a schedule with greater finish time.

Assume that S is a feasible sequence, with L_{pre}, L, and L_{post} subsequences of S such that

$$S = L_{pre} \oplus L \oplus L_{post}.$$

Let us denote L by

$$L = \langle T_\beta, T_{x_1}, T_{x_2}, \ldots, T_{x_\omega}, T_\alpha \rangle,$$

where $\omega \geq 0$. A frame F is defined to be a time interval characterized by a beginning time b_F, and an ending time e_F. We say that F is a frame corresponding to L, if $b_F = s_\beta$, and $e_F = f_\alpha$, where s_β is the starting time of T_β, and f_α is the finish time of T_α.

Theorem 2 (Leading Theorem) Assume that $S = L_{pre} \oplus L \oplus L_{post}$ is a feasible sequence, where $L = \langle T_\beta, T_{x_1}, T_{x_2}, \ldots, T_{x_\omega}, T_\alpha \rangle$. Let F be a frame corresponding to L. If $T_\alpha \prec T_\beta$, and there does not exist a task T_{x_i}, $1 \leq i \leq \omega$, such that $F \sqcup T_{x_i}$, then there exists a sequence $\bar{L}$ which is a permutation of L such that
(i) $\langle T_\alpha, T_\beta \rangle$ conforms to $\bar{L}$, and
(ii) $f_{L_{pre} \oplus \bar{L} \oplus L_{post}} \leq f_{L_{pre} \oplus L \oplus L_{post}}$.

Before we can proceed to prove the theorem, the following definition is useful.

Weakly Leading Condition: a sequence $S = \langle T_1^S, T_2^S, \ldots, T_k^S \rangle$ satisfies Weakly Leading Condition if $T_1^S \lhd T_2^S \lhd \ldots \lhd T_k^S$.

Lemma 3 Let S be a sequence satisfying Weakly Leading Condition. If $\langle T_i, T_j \rangle$ conforms to S and $T_i \parallel T_j$, then all tasks located between T_i and T_j in S must match T_i and T_j.

PROOF: For any task T_x located between T_i and T_j, according to the definition of Weakly Leading Condition, we have $r_i \leq r_x \leq r_j$ and $d_i \leq d_x \leq d_j$. Since $T_i \parallel T_j$, $r_i = r_j$ and $d_i = d_j$. Therefore, we have $r_i = r_x = r_j$ and $d_i = d_x = d_j$. So, T_x matches T_i and T_j. □

To obtain $\tilde{L}$, let us modify the tasks in L in the following way. If the ready time of a task is less than b_F, then its ready time is set to b_F. If the deadline of a task is greater than e_F, then its deadline is set to e_F. The computation times remain unchanged. Let L' be a sequence consisting of the modified tasks with the same order of L, i.e.,

$$L' = \langle T'_\beta, T'_{x_1}, T'_{x_2}, \ldots, T'_{x_w}, T'_\alpha \rangle.$$

Since $T_\alpha \prec T_\beta$, $d_\beta \geq d_\alpha \geq f_\alpha = e_F$. So $d'_\beta = d'_\alpha = e_F$. Also $r_\alpha \leq r_\beta \leq s_\beta = b_F$, so $r'_\alpha = r'_\beta = b_F$. This is illustrated in Fig 4 (ii).

Note that swapping T'_β and T'_α in the sequence does not result in a feasible sequence in this example. It is essential that we adjust the order of the tasks located between them. Let $\tilde{L}'$ be a sequence which is a permutation of L' and satisfies the Weakly Leading Condition, and to which $\langle T'_\alpha, T'_\beta \rangle$ conforms. Furthermore, $\tilde{L}'$ satisfies an even stronger condition. If $T_i^{\tilde{L}'}$ is positioned before $T_j^{\tilde{L}'}$ in $\tilde{L}'$, then $T_i^{\tilde{L}'} \lhd T_j^{\tilde{L}'}$; if, furthermore, $T_i^{\tilde{L}'} \parallel T_j^{\tilde{L}'}$, the corresponding tasks $T_{i'}^{L'}$ and $T_{j'}^{L'}$ satisfies that $T_{i'}^{L'} \lhd T_{j'}^{L'}$. The idea of such arrangement is that when interchanging T'_β and T'_α, we do not produce a new *reversed pair* like them. By reversed pair we mean for example $T_\alpha \prec T_\beta$ but T_β is positioned before T_α in the sequence. So, if $\langle T_i^{\tilde{L}'}, T_j^{\tilde{L}'} \rangle$ conforms to $\tilde{L}'$, the corresponding tasks satisfies the condition that either $T_{i'}^{L'} \prec T_{j'}^{L'}$ or $T_{i'}^{L'} \sqcup T_{j'}^{L'}$ or $T_{j'}^{L'} \sqcup T_{i'}^{L'}$. One possibility of $\tilde{L}'$ is illustrated in Fig 4 (iii), or

$$\tilde{L}' = \langle T'_{x_2}, T'_\alpha, T'_{x_3}, T'_\beta, T'_{x_1}, T'_{x_4} \rangle.$$

The existence of such a sequence is proved later. Finally, $\tilde{L}$ can be a sequence with the same order of $\tilde{L}'$, but the ready times and deadlines of the tasks are recovered to their original settings. This is illustrated in Fig 4 (iv). The figures give the rough idea about how the adjustment of task order can be made to satisfy the conditions described in the Leading Theorem. Here below is the proof of the Leading Theorem.

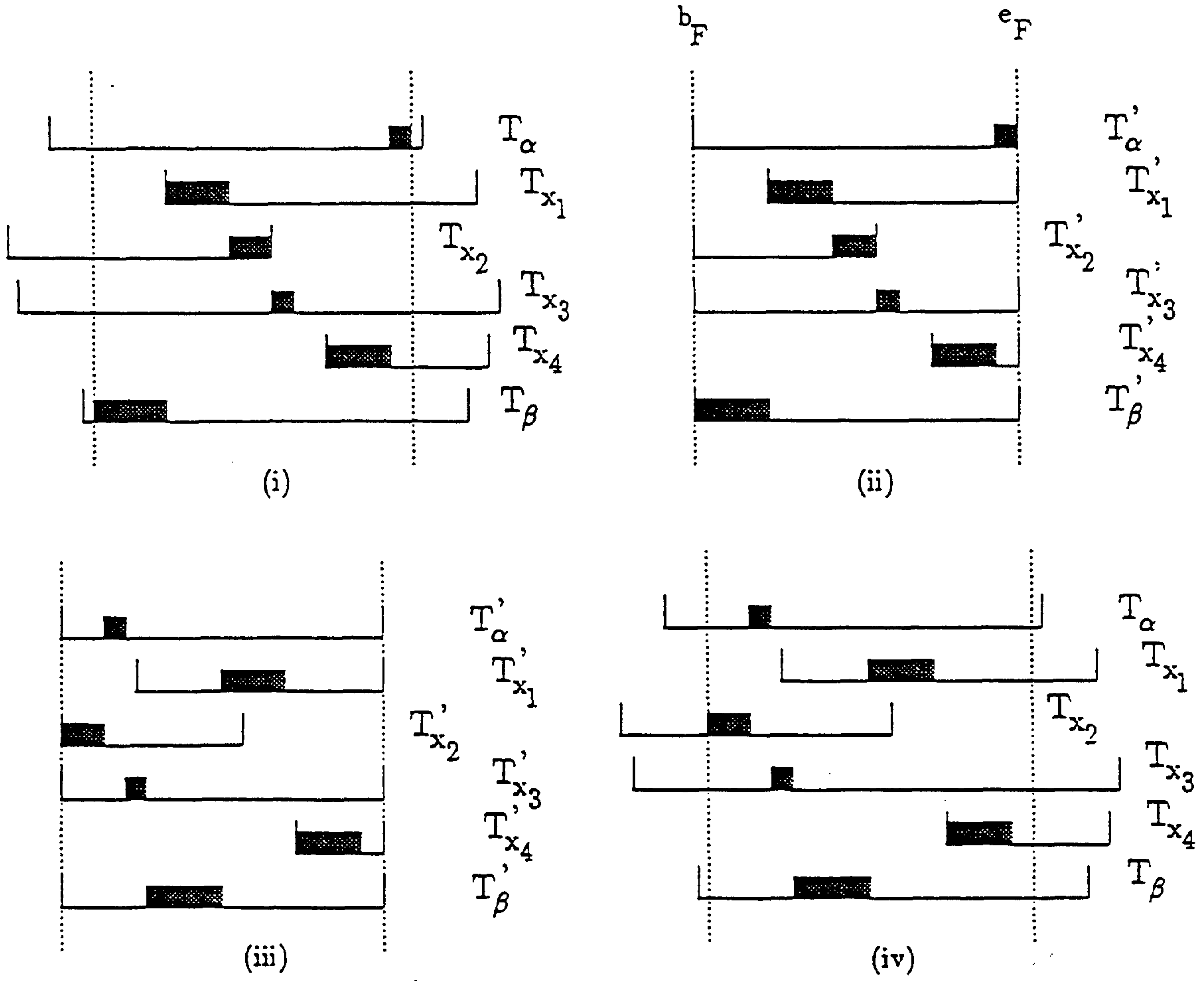


Figure 4: (i) $L = \langle T_\beta, T_{x_1}, T_{x_2}, T_{x_3}, T_{x_4}, T_\alpha \rangle$ (ii) $L' = \langle T'_\beta, T'_{x_1}, T'_{x_2}, T'_{x_3}, T'_{x_4}, T'_\alpha \rangle$ (iii) $\tilde{L}' = \langle T'_{x_2}, T'_\alpha, T'_{x_3}, T'_\beta, T'_{x_1}, T'_{x_4} \rangle$ (iv) $\tilde{L} = \langle T_{x_2}, T_\alpha, T_{x_3}, T_\beta, T_{x_1}, T_{x_4} \rangle$

PROOF (of Leading Theorem): We would first show the existence of $\bar{L}$. The modification of ready times and deadlines of the tasks for L' is done in such a way that their started times are not affected. In addition, their computation times remain the same. It is clear that L' is feasible, and

$$f_{L_{pre} \oplus L'} = f_{L_{pre} \oplus L}.$$

We can obtain $\bar{L}'$ in the following way. At the first step, the first task $T_1^{\bar{L}'}$ of $\bar{L}'$ is the task in L' such that, for any task T_x belonging to L', $T_1^{\bar{L}'} \lhd T_x$. Such a task $T_1^{\bar{L}'}$ exists because there are no containing relations among the modified tasks, and ties can be broken arbitrarily. $T_1^{\bar{L}'}$ is exchanged with the task located just left to it in L'. Continue the exchanging process until $T_1^{\bar{L}'}$ occupies the first location in the sequence. At the second step, the second task $T_2^{\bar{L}'}$ of $\bar{L}'$ is the task in L' such that, for any task T_x belonging to L' except $T_1^{\bar{L}'}$, $T_2^{\bar{L}'} \lhd T_x$. Exchange $T_2^{\bar{L}'}$ with its left neighbor task consecutively until it occupies the second location in the sequence. At the ith step, the ith task $T_i^{\bar{L}'}$ of $\bar{L}'$ is the task in L' such that, for any task T_x belonging to L' except $T_1^{\bar{L}'}$ through $T_{i-1}^{\bar{L}'}$, $T_i^{\bar{L}'} \lhd T_x$. Exchange $T_i^{\bar{L}'}$ with its left neighbor task consecutively until it occupies the ith location in the sequence. We keep performing this operation until we finally obtain $\bar{L}'$. Insertion of $T_i^{\bar{L}'}$, $1 \leq i \leq |\bar{L}'|$, into the ith position of the sequence by consecutive swapping is possible because $T_i^{\bar{L}'} \lhd T_x$ for all T_x not belonging to $\langle T_1^{\bar{L}'} \dots T_{i-1}^{\bar{L}'} \rangle$. In a word, the adjustment is possible because there are no containing relations among the modified tasks, and hence there exists a total ordering of the modified tasks by the Weakly Leading Condition. The resultant $\bar{L}'$ is existent and is a sequence satisfying the Weakly Leading Condition.

There is a chance that $\langle T_\beta, T_\alpha \rangle$ conforms to $\bar{L}'$. By Lemma 3, all tasks located between T_β and T_α must match each other. Hence the order of these tasks does not make any difference. We can thus exchange the position of T_β and T_α, which makes $\langle T_\beta, T_\alpha \rangle$ conform to $\bar{L}'$

During the process of adjusting the position of $T_i^{\bar{L}'}$, $1 \leq i \leq |\bar{L}'|$, $T_i^{\bar{L}'}$ leads to or matches any task in the sequence except $\langle T_1^{\bar{L}'} \dots T_{i-1}^{\bar{L}'} \rangle$. Thus we can apply Lemma 4, to be described next, which assures that the resultant sequence after swapping $T_i^{\bar{L}'}$ to the ith location comes with a shorter or equal finish time. This explains

$$f_{L_{pre} \oplus \bar{L}'} \leq f_{L_{pre} \oplus L'} = f_{L_{pre} \oplus L}.$$

$\tilde{L}$ is a sequence with the same order of $\tilde{L}'$, but the ready times and deadlines of the tasks are recovered to their original values. Each task in $\tilde{L}$ can be started no later than the starting time of the same task in $\tilde{L}'$. Consequently,

$$f_{L_{pre}\oplus\tilde{L}} \leq f_{L_{pre}\oplus L}.$$

By Corollary 1, we have

$$f_{L_{pre}\oplus\tilde{L}\oplus L_{post}} \leq f_{L_{pre}\oplus L\oplus L_{post}}.$$

□

Lemma 4 Assume that $S1 \oplus S2 \oplus \langle T_j \rangle \oplus S3$ is feasible, where $S1, S2$, and $S3$ are sequences. If $T_j \lhd S2$, then $f_{S1\oplus\langle T_j\rangle\oplus S2\oplus S3} \leq f_{S1\oplus S2\oplus\langle T_j\rangle\oplus S3}$.

PROOF: We will prove the theorem by induction on $|S2|$. When $|S2| = 0$, it is vacuously true. Assume that it is true when $|S2| = k$. We would like to show that it is true when $|S2| = k+1$. Let $S2 = \langle T_i \rangle \oplus S2'$, where $|S2'| = k$; i.e.,

$$S1 \oplus S2 \oplus \langle T_j \rangle \oplus S3 = S1 \oplus \langle T_i \rangle \oplus S2' \oplus \langle T_j \rangle \oplus S3.$$

We can view $S1 \oplus \langle T_i \rangle$ as a single sequence, and because $|S2'| = k$, by induction hypothesis, we have

$$f_{S1\oplus\langle T_i\rangle\oplus\langle T_j\rangle\oplus S2'\oplus S3} \leq f_{S1\oplus\langle T_i\rangle\oplus S2'\oplus\langle T_j\rangle\oplus S3}.$$

By definition,

$$\begin{aligned} f_{S1\oplus\langle T_i\rangle\oplus\langle T_j\rangle} &= max(max(f_S, r_i) + c_i, r_j) + c_j \\ &= max(f_S + c_i + c_j, r_i + c_i + c_j, r_j + c_j) \end{aligned}$$

Since $T_j \lhd S2$, which indicates that $T_j \lhd T_i$, we have $r_j \leq r_i$, and $r_j + c_j \leq r_i + c_i + c_j$.

$$f_{S1\oplus\langle T_i\rangle\oplus\langle T_j\rangle} = max(f_S + c_i + c_j, r_i + c_i + c_j)$$

On the other hand,

$$\begin{aligned} f_{S1 \oplus (T_j) \oplus (T_i)} &= max(max(f_S, r_j) + c_j, r_i) + c_i \\ &= max(f_S + c_i + c_j, r_j + c_i + c_j, r_i + c_i) \end{aligned}$$

Because $r_j + c_i + c_j \leq r_i + c_i + c_j$, we have

$$f_{S1 \oplus (T_j) \oplus (T_i)} \leq f_{S1 \oplus (T_i) \oplus (T_j)}.$$

By Corollary 1,

$$f_{S1 \oplus (T_j) \oplus (T_i) \oplus S2' \oplus S3} \leq f_{S1 \oplus (T_i) \oplus (T_j) \oplus S2' \oplus S3}.$$

Therefore,

$$f_{S1 \oplus (T_j) \oplus S2 \oplus S3} \leq f_{S1 \oplus S2 \oplus (T_j) \oplus S3}.$$

□

3.3 Dominance Theorem

The super sequence is constructed for the feasibility test of a task set. If a task set is feasible, we say that there exists a *full schedule* of the task set. There may exist more than one full schedule for a given task set. An *optimal full schedule* is a full schedule whose finish time is shortest among all the possible full schedules. Note that a full schedule is a feasible instance. In this section, we prove that if a task set is feasible, there exists an optimal full schedule conforming to the super sequence [*] Hence, the super sequence provides a valid and pruned search space for deriving the optimal full schedule of a task set.

[*] In [2], Erschler *et al.*'s theorem implied a similar result: if a task set is feasible, there exists a full schedule in the dominant set. Our theorem further shows that there exists such a full schedule, with the minimum finish time among all full schedules, that conforms to the super sequence. We prove the existence of such an optimal full schedule in a more systematic way.

Theorem 3 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let T_α and T_β be two top tasks of ρ such that $T_\alpha \prec T_\beta$. If $\langle T_\beta, T_\alpha \rangle$ conforms to ρ, then there exists another optimal full schedule ρ' such that $\langle T_\alpha, T_\beta \rangle$ conforms to ρ'.

PROOF: T_α and T_β are two top tasks. Let F be a frame such that $b_F = s_\beta$ and $e_F = f_\alpha$. $T_\alpha \prec T_\beta$ means $b_F = s_\beta \geq r_\beta \geq r_\alpha$, and $e_F = f_\alpha \leq d_\alpha$. If there exists a task T_x such that $F \sqcup T_x$, then $T_\alpha \sqcup T_x$ too. This contradicts to the fact that T_α is a top task. Hence F cannot contain any task. By the Leading Theorem, there exists another sequence ρ' such that $\langle T_\alpha, T_\beta \rangle$ conforms to ρ', and both $|\rho'| = |\rho|$ and $f_{\rho'} \leq f_\rho$ hold, which means ρ' is an optimal full schedule too. □

When two tasks match each other, it dose not matter which task is executed first. This gives rise to the following Corollary.

Corollary 2 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Also assume that $\langle h_1, \ldots, T_\beta, T_\alpha, \ldots, h_t \rangle$, the subsequence of the top tasks in ρ, conforms to ρ. If $T_\alpha \lhd T_\beta$, then there exists another optimal full schedule ρ' such that $\langle h_1, \ldots, T_\alpha, T_\beta, \ldots, h_t \rangle$ conforms to ρ'.

PROOF: Theorem 3 holds when $T_\alpha \lhd T_\beta$, because when two tasks match each other, the execution order of the two tasks is arbitrary. Also by looking at the adjustment process of Leading Theorem, we can find that the tasks located before and after T_α and T_β have not been adjusted. This verifies the corollary. □

Corollary 3 Let $H = \langle h_1, h_2, \ldots, h_t \rangle$ be top tasks of the task set Γ such that $h_1 \lhd h_2 \lhd \ldots \lhd h_t$. If Γ is feasible, there exists an optimal full schedule ρ' to which H conforms.

PROOF: Since Γ is feasible, there exists an optimal full schedule ρ. Let $K = \langle k_1, k_2, \ldots, k_t \rangle$ be a sequence which is a permutation of H such that K conforms to ρ. We would like to adjust the order of the tasks in K so that K is transformed successively into H. We locate the corresponding task of h_x in K, where x is chosen in the order of 1 through t, and adjust it to the xth position in K by consecutively swapping h_x with its left neighbor. This leads to the sequence H. During the swapping, h_x always weakly leads to its left neighbor, for $h_1, \ldots, h_{x-1}$ are in positions $1, \ldots, x-1$. By Corollary 2, there always exists an optimal full schedule to which the intermediate resultant sequences conform. Therefore, there exists an optimal full schedule ρ' to which $\langle h_1, h_2, \ldots, h_t \rangle$ conforms. $\square$

Given an optimal full schedule ρ, we can always obtain another optimal full schedule ρ' in which the top tasks are ordered according to the weakly leading relations by Corollary 3. Therefore the rule R1 is verified.

Before we can go further, the following definitions are useful. Let h_k be a top task and T_x a nontop task of a sequence S. We say that (h_k, T_x) is a *disorder pair* of S if $\langle h_k, T_x \rangle$ conforms to S and $T_x \prec h_k$. Similarly, (T_x, h_k) is a disorder pair of S if $\langle T_x, h_k \rangle$ conforms to S and $h_k \prec T_x$. The *disorder degree* of S is defined to be the number of disorder pairs in S.

Theorem 4 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let $h_1 \lhd h_2 \lhd \ldots \lhd h_t$ be top tasks and T_x a nontop task of ρ. Assume that $\rho = L_{pre} \oplus L \oplus L_{post}$ such that

$$\langle h_1, \ldots, h_{k-1} \rangle \textit{ conforms to } L_{pre},$$
$$\langle h_{k+1}, \ldots, h_t \rangle \textit{ conforms to } L_{post}.$$

We have the following properties:
(1) if $T_x \prec h_k$ and $L = \langle h_k, \ldots, T_x \rangle$, then there exists another optimal full schedule $\rho' = L_{pre} \oplus \bar{L} \oplus L_{post}$ such that $\bar{L}$ is a permutation of L, and $\langle T_x, h_k \rangle$ conforms to $\bar{L}$; besides, the disorder degree of ρ' is less than that of ρ
(2) if $h_k \prec T_x$ and $L = \langle T_x, \ldots, h_k \rangle$, then there exists another optimal full schedule $\rho' = L_{pre} \oplus \bar{L} \oplus L_{post}$ such that $\bar{L}$ is a permutation of L, and $\langle h_k, T_x \rangle$ conforms to $\bar{L}$; besides, the disorder degree of ρ' is less than that of ρ

PROOF: We will prove (1) first. Let F be a frame with $b_F = s_{h_k}$ and $e_F = f_x$. Since $T_x \prec h_k$, $e_F = f_x \leq d_x \leq d_{h_k}$. Also $b_F = s_{h_k} \geq r_{h_k}$. If there exists a task T_{w_i} located between h_k and

T_x such that $F \sqcup T_{w_i}$, then $h_k \sqcup T_{w_i}$ too. This contradicts to the fact that h_k is a top task. The condition of the Leading Theorem is satisfied. Hence there exists a sequence $\tilde{L}$ which is a permutation of L such that $\langle T_x, h_k \rangle$ conforms to $\tilde{L}$ and $f_{L_{pre} \oplus \tilde{L} \oplus L_{post}} \leq f_\rho$. Therefore, $L_{pre} \oplus \tilde{L} \oplus L_{post}$ is also an optimal full schedule. Now let us look at Figure 4(iv). This is the schedule after the adjustment process of the Leading Theorem is made. For the tasks whose deadlines are less than e_F, they all lead to h_k. Note that the disorder is a relationship defined between a nontop task and a top task, and h_k is the only top task in the frame F. Therefore, no new disorder pairs with h_k are introduced among these tasks. Similarly, for the tasks whose ready times are greater than b_F, they are all led by h_k. Therefore, no new disorder pairs are introduced. As for the tasks otherwise, including T_x and h_k, whose deadlines are greater than or equal to e_F and ready times less than or equal to b_F, they can be ordered arbitrarily. Hence, we can position T_x before h_k, and remove the disorder pairs, if any, in these tasks by rearranging the proper orders for them. Thus the disorder degree of $\tilde{L}$ is decremented by at least one. So the disorder degree of ρ' is less than that of ρ. Property (2) holds for the same reason. □

Note that T_x does not match h_k or h_{k+1}; otherwise T_x is also a top task, which contradicts our assumption.

Theorem 5 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let $h_1 \lhd h_2 \lhd \ldots \lhd h_t$ be top tasks of ρ. There exists an optimal full schedule ρ' such that $\langle h_1, h_2, \ldots, h_t \rangle$ conforms to ρ', and for any nontop task T_x such that $\langle h_k, T_x, h_{k+1} \rangle$ conforms to ρ', either $T_x \sqcup h_k$ or $T_x \sqcup h_{k+1}$.

Proof: Assume that T_x is a nontop task such that $\langle h_k, T_x, h_{k+1} \rangle$ conforms to ρ'. If T_x does not contain h_k and T_x does not contain h_{k+1}, then either $T_x \prec h_k$ or $h_{k+1} \prec T_x$. Hence, either (h_k, T_x) or (T_x, h_{k+1}) is a disorder pair. We can eliminate it through Theorem 4, and the disorder degree is decremented by at least one. Whenever there is a disorder pair in the schedule, we can always apply Theorem 4 to eliminate it. The disorder degree is decremented in this way until finally reaching zero. Hence, $\langle h_k, T_x, h_{k+1} \rangle$ conforming to ρ' implies that T_x is not leading to h_k and h_{k+1} is not leading to T_x. The only possibilities are either $T_x \sqcup h_k$

or $T_x \sqcup h_{k+1}$. □

Theorem 5 confirms the validity of rules R1 and R2.

Theorem 6 (Dominance Theorem) If a task set Γ is feasible, there exists an optimal full schedule ρ such that ρ conforms to the super sequence of Γ.

PROOF: In Theorem 5, we verify the existence of the optimal full schedule such that the top tasks are ordered according to their weakly leading relations, and the nontop tasks are located in the appropriate subsets between top tasks. The only work left is to order the nontop tasks in each subset. The adjustment process of the Leading Theorem can be applied, and the resultant order is exactly specified by rule R3. So we can conclude that there exists an optimal full schedule ρ which conforms to the super sequence. □

3.4 Conformation Theorem

If there is no task rejected in ρ, there exists an optimal full schedule conforming to the super sequence of Γ. However, if Γ is not feasible, some tasks in Γ should be rejected. The dominant rules are developed based on the assumption that no task is rejected. When tasks are allowed to be rejected, the situation is different. The issue to be raised is whether the decent solution for feasibility test can be applied to our optimization problem. Remember that by optimization we mean that the number of rejected tasks in the schedule is minimized and then the finish time of the schedule is also minimized. When a task set is feasible, the optimal schedule is also the optimal full schedule. The difficulties are addressed in the next section, followed by the approach and proof to solving the difficulties.

3.4.1. Difficulties

We wish to make use of the super sequence as search space in our scheduling problem. The difficulties are twofold.

First, when a task is allowed to be rejected, the dominant rules specifying the relations among containing tasks and contained tasks need to be modified, because the rules are

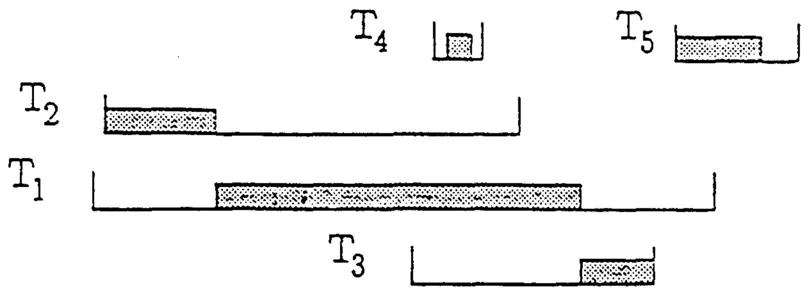


Figure 5: The optimal schedule may not conform to the super sequence.

developed based on the assumption that no task is to be rejected. The new rules can become quite complicated. Let's look at the example depicted in Figure 5. Assume that the task set is

$$\Gamma = \{T_1, T_2, T_3, T_4, T_5\},$$

and the super sequence of the task set is

$$\Delta = \langle T_1, T_2, T_3, \mathbf{T_4}, T_2, T_3, T_1, \mathbf{T_5}\rangle.$$

The top tasks are typed in bold letters for emphasis. Γ is not feasible. We can see that one possibility of the optimal schedule could be

$$\rho_0 = \langle T_2, T_1, T_3, T_5\rangle.$$

Apparently, ρ_0 does not conform to Δ. One may be able to show that another optimal schedule $\langle T_2, T_4, T_3, T_5\rangle$ conforms to Δ. However, given an arbitrary task set, it is not guaranteed that one is always able to do so. In the example, T_4 is rejected. If we recompute the super sequence without T_4, we would get a different super sequence. The new super sequence would be

$$\Delta_0 = \langle T_1, \mathbf{T_2}, T_1, \mathbf{T_3}, T_1, \mathbf{T_5}\rangle,$$

to which ρ_0 conforms. This gives a great difficulty. It seems that we need to check against each task. Construct a super sequence in condition that the task is accepted, and construct another super sequence in condition that the task is rejected. In general, we need to construct 2^n super sequences in this way. This is too formidable to schedule, considering that the number of instances in each individual super sequence can be exponential to the size of the task set.

Secondly, while rejecting a nontop tasks does not affect much, rejecting a top task could affect the duplication and positions of the nontop tasks or might even result in some new top tasks. Thus, the super sequences can be totally different. Look at the same example in Figure 5. The rejection of T_4 results in two more top tasks, i.e., T_2 and T_3. This makes Δ_0 completely different from Δ.

We propose a *swapping and replacing* procedure to overcome the difficulties. The procedure would be described and verified in the following section.

3.4.2 Approach and Proof

The idea of our approach is stated briefly below, followed a formal proof. Let Γ and Δ be the original task set, and the super sequence of the task set respectively. It is clear that there exists an optimal schedule, which is unknown to us, for any task set. Let Γ_0 be the task set which is composed of the tasks of the unknown optimal schedule, and Δ_0 be the super sequence of Γ_0. Γ_0 and Δ_0 are also unknown to us. As mentioned above, Δ_0 might be quite different from Δ. Notice that the unknown optimal schedule of Γ is also an optimal (full) schedule of Γ_0. Since Γ_0 is feasible, by the Dominance Theorem, there exists an optimal full schedule for Γ_0, say ρ_0, such that ρ_0 conforms to Δ_0. Our problem is that we are not able to compute ρ_0 from Δ_0, because Δ_0 is unknown. We are able to compute Δ from Γ by applying the dominant rules. The swapping and replacing procedure exploits the way to adjust the order of tasks in ρ_0 and to replace some tasks if necessary, so as to transform ρ_0 into a new schedule ρ such that ρ is also an optimal schedule and best of all ρ conforms to an instance of Δ. For the sake of simplicity, we will say a schedule conforms to Δ, when the schedule conforms to an instance of Δ. So we can use Δ as a valid search space when scheduling Γ. In the example of Figure 5, we transform ρ_0 into

$$\rho = \langle T_2, T_4, T_3, T_5 \rangle.$$

This example is so simplified that the existence of ρ can be verified by mere intuition. However, the reasoning is far more complicated than it appears at the first glance. We are going to prove in the following theorem that such an optimal schedule ρ that conforms to Δ always exists. The corresponding lemmas are presented in the next section.

Theorem 7 (Conformation Theorem) Given a task set $\Gamma = \{T_1, T_2, ..., T_n\}$, there exists an optimal schedule ρ such that ρ conforms to the super sequence Δ of Γ.

PROOF: Given any task set Γ, there exists at least one optimal schedule, which is unknown to us. Assume that we need to reject w tasks from Γ to make a feasible schedule. Let Γ_0 be the task set which is composed of the tasks in the unknown optimal schedule. Γ_0 is a subset of Γ. The super sequence of Γ_0 is denoted by Δ_0. In addition, we use Γ_j, $0 \leq j \leq w$, to represent a task set derived by adding j tasks into Γ_0, Δ_j the super sequence of Γ_j, and ρ_j an optimal schedule of Γ_j. When we say adding j tasks into Γ_0, we mean that the resultant task set Γ_j is composed of distinct tasks and Γ_j is a subset of Γ. In particular, Γ_w is Γ. We will prove by induction on w to show that there exists an optimal schedule ρ_w for Γ conforming to Δ_w.

Base step $w = 0$: there is no task rejected. $\Gamma = \Gamma_0$. Since Γ is feasible, by the Dominance Theorem, there exists an optimal (full) schedule ρ_0 for Γ such that ρ_0 conforms to Δ_0.

Induction hypothesis: assume that the theorem holds when $w = j$, i.e., $|\rho_0| = n - j$. For the task set Γ_j which is derived by adding j tasks into Γ_0, there exists an optimal schedule ρ_j for Γ such that ρ_j conforms to Δ_j. Notice that $|\rho_j| = |\rho_0|$, and $|\Gamma_j| = |\Gamma_0| + j$.

Now consider the case when $w = j + 1$, i.e., $|\rho_0| = n - (j + 1)$. We need to reject $j + 1$ tasks to make a feasible schedule. There exists an optimal schedule ρ_j for Γ conforming to Δ_j by induction hypothesis. We want to show that, by swapping and replacing the tasks in ρ_j, the resultant sequence ρ_{j+1} conforms to Δ_{j+1}; besides, $|\rho_{j+1}| = |\rho_j|$, and $f_{\rho_{j+1}} \leq f_{\rho_j}$, which implies that ρ_{j+1} is also an optimal schedule for Γ. Let T_x be the task added into Γ_j to make Γ_{j+1}. So, $\Gamma_j \cup \{T_x\} = \Gamma_{j+1}$. There are two possibilities when adding T_x.

If T_x is a nontop task of Γ_{j+1}, adding T_x does not add a top task into Γ_j. The orders of the top tasks in both Δ_{j+1} and Δ_j derived through rule R1 are exactly the same. Rule R2 specifies the relation between a nontop task and a top task. Adding a nontop task T_x

does not affect the relations between the already existent nontop tasks and top tasks. The positions (duplicates) of the already existent nontop tasks in Δ_j are preserved in Δ_{j+1}. Rule R3 specifies how to arrange the order of the nontop tasks within each subset. Again adding a nontop task T_x does not alter the orders of the already existent nontop tasks in each subset in Δ_j. Therefore, if the task being added is a nontop task, Δ_j is a subsequence of Δ_{j+1}. Let us look at the example in Figure 5. Assume that Γ_j and Γ_{j+1} are

$$\Gamma_j = \{T_2, T_3, T_4, T_5\}, \text{ and}$$

$$\Gamma_{j+1} = \{T_1, T_2, T_3, T_4, T_5\},$$

where T_1 is a nontop task. The corresponding super sequences would be

$$\Delta_j = \langle T_2, T_3, \mathrm{T}_4, T_2, T_3, \mathrm{T}_5\rangle, \text{ and}$$

$$\Delta_{j+1} = \langle T_1, T_2, T_3, \mathrm{T}_4, T_2, T_3, T_1, \mathrm{T}_5\rangle.$$

We can see in the example how Δ_j conforms to Δ_{j+1}.

Otherwise, T_x is a top task of Γ_{j+1}. T_x does not contain other tasks in Γ_{j+1}. Two situations are possible.

(i) T_x is not contained by other tasks. The number of top tasks in Γ_{j+1} is one more than that of the top tasks in Γ_j. The order of the top tasks in Δ_j is preserved in Δ_{j+1}, since the relations of the top tasks are not altered by adding T_x. Furthermore, T_x does not alter any existent orders among the nontop tasks and top tasks, or among the orders between the nontop tasks and nontop tasks, specified by rules R2 and R3, respectively. Therefore, Δ_j is a subsequence of Δ_{j+1}. Let us look at the example in Figure 6. Assume that Γ_j and Γ_{j+1} are

$$\Gamma_j = \{T_1, T_2, T_4, T_5\}, \text{ and}$$

$$\Gamma_{j+1} = \{T_1, T_2, T_3, T_4, T_5\},$$

where T_3 is a top task not contained by other tasks. The corresponding super sequences would be

$$\Delta_j = \langle T_1, \mathrm{T}_2, T_1, T_4, \mathrm{T}_5, T_4\rangle, \text{ and}$$

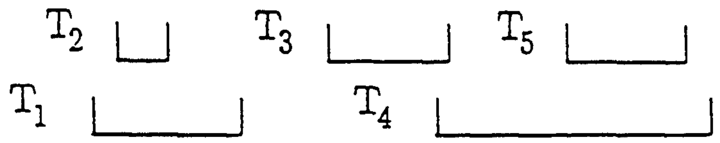


Figure 6: The added top task T_3 is not contained by other tasks.

$\Delta_{j+1} = \langle T_1, \mathrm{T}_2, T_1, T_3, T_4, \mathrm{T}_5, T_4 \rangle$.

We can see in the example how Δ_j conforms to Δ_{j+1}.

(ii) T_x is contained by some top tasks and/or nontop tasks of Γ_j. Let the top tasks of Γ_j containing T_x be $g_1, \ldots, g_m$, indexed in the weakly leading order. This situation is more complicated, because $g_i, i = 1, \ldots, m$, turn out to be nontop tasks in Γ_{j+1}. There exists a total ordering of them by weakly leading relations, because there is no containing relations among g_i. By rule R1, the super sequence of Γ_{j+1} can be expressed as

$$\Delta_{j+1} = \langle \ldots, h_{k-1}, \ldots, g_1, \ldots, g_m, \ldots, h_k, \ldots, g_1, \ldots, g_m, \ldots, h_{k+1}, \ldots \rangle,$$

where $h_1, \ldots, h_{k-1}, h_k, h_{k+1}, \ldots$ are the top tasks in Γ_{j+1}, and in particular, h_k represents T_x. By rule R3, the super sequence of Γ_{j+1} can also be expressed as

$$\Delta_{j+1} = \langle \ldots, h_{k-1}, \underbrace{B_{k-1,\overline{k}}, \underbrace{B_{k-1,k}, B_{\overline{k-1},k}, h_k, B_{k,\overline{k+1}}, B_{k,k+1}}_{\Omega'_{j+1}}, B_{\overline{k},k+1}}_{\Omega_{j+1}}, h_{k+1}, \ldots \rangle, \tag{6}$$

where Ω_{j+1} represents the subsequence of Δ_{j+1} between h_{k-1} and h_{k+1}, excluding h_{k-1} and h_{k+1}, as depicted above, and $\Omega_{j+1} = B_{k-1,\overline{k}} \oplus \Omega'_{j+1} \oplus B_{\overline{k},k+1}$, where $\oplus$ means concatenation of sequences. Remember that $g_1, \ldots, g_m$ are top tasks of Γ_j. All the top tasks in Γ_j are in the order of $h_1, \ldots, h_{k-1}, g_1, \ldots, g_m, h_{k+1}, \ldots$ by the weakly leading relations. By rule R1, the super sequence of Γ_j can be expressed as

$$\Delta_j = \langle \ldots, h_{k-1}, \underbrace{\ldots, g_1, \ldots, g_m, \ldots}_{\Omega_j}, h_{k+1}, \ldots \rangle, \tag{7}$$

where Ω_j represents the subsequence of Δ_j between h_{k-1} and h_{k+1}, excluding h_{k-1} and h_{k+1}, as depicted above. Notice that in Equations 6 and 7, the subsequences before h_{k-1} of both Δ_{j+1} and Δ_j are exactly the same, because the addition of the top task h_k, or T_x, affects only the subsequence between h_{k-1} and h_{k+1}. Similarly, the subsequences after h_{k+1} of both Δ_{j+1} and Δ_j are exactly the same too. Hence an instance of Δ_{j+1} will differ from an instance of Δ_j only in the the subsequences of Ω_{j+1} and Ω_j.

Now we would like to check what tasks in Ω_j should follow immediately after h_{k-1}. By Lemma 7, all the tasks in Ω_j can be found in $\Omega_{j+1} - h_k$. So we only need to check the tasks in $\Omega_{j+1} - h_k$. If a task contains h_{k-1} but not h_k, the task must not contain g_1. Because g_1 contains h_k, any task which contains g_1 should also contain h_k. When constructing the subsequence of Δ_j between h_{k-1} and g_1, by rule R2, all the tasks which appear in $B_{k-1,\bar{k}}$ of Δ_{j+1} should follow immediately after h_{k-1}; and by rule R3, the order of the subsequence is exactly the same as $B_{k-1,\bar{k}}$.

One may observe that some tasks in $B_{k-1,k}$ contain h_{k-1} but do not contain g_1, so they would also be positioned between h_{k-1} and g_1. These tasks would follow after the tasks of $B_{k-1,\bar{k}}$. This is because they do contain h_k and hence have greater deadlines than those tasks in $B_{k-1,\bar{k}}$. For the same reason, all the tasks which appear in $B_{\bar{k},k+1}$ of Δ_{j+1} should be located immediately before h_{k+1} when constructing Δ_j, and the order is the same. Hence, the Δ_j can be further expressed as

$$\Delta_j = \langle \ldots, h_{k-1}, B_{k-1,\bar{k}}, \underbrace{\ldots, g_1, \ldots, g_m, \ldots}_{\Omega'_j}, B_{\bar{k},k+1}, h_{k+1}, \ldots \rangle, \tag{8}$$

where Ω'_j represents the subsequence between $B_{k-1,\bar{k}}$ and $B_{\bar{k},k+1}$ of Δ_j, excluding $B_{k-1,\bar{k}}$ and $B_{\bar{k},k+1}$. We have $\Omega_j = B_{k-1,\bar{k}} \ominus \Omega'_j \ominus B_{\bar{k},k+1}$. By Lemma 9, all the tasks in Ω'_j are either in $B_{k-1,k}$ or in $B_{\overline{k-1},k}$.

Let us look at an example in Figure 7. The task set in the figure is Γ_{j+1}. And $\Gamma_{j+1} - h_k$ would be Γ_j. g_1 and g_2 contain only h_k. So g_1 and g_2 are classified as nontop tasks in Γ_{j+1},

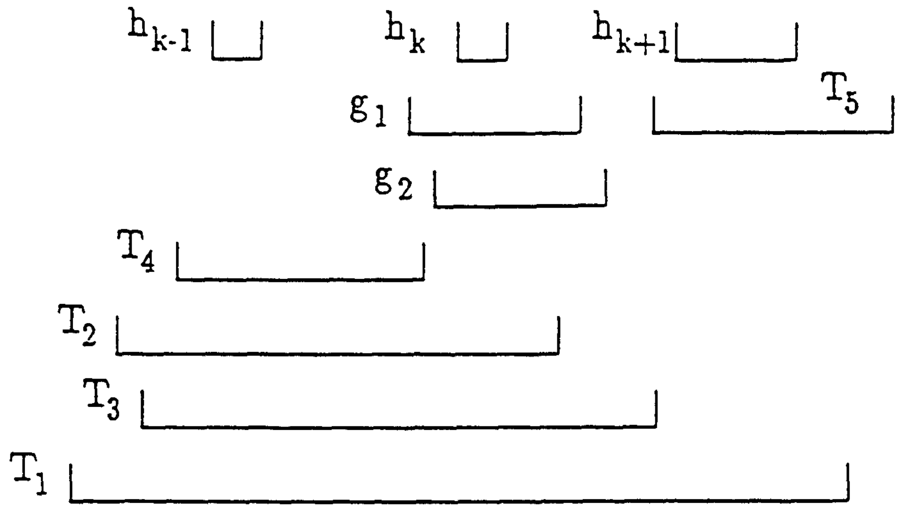


Figure 7: The added top task h_k is contained by other tasks.

and as top tasks in Γ_j. We can compute the super sequences as follows.

$$\Delta_{j+1} = \langle T_1, T_2, T_3, T_4, \mathrm{h}_{k-1}, \underbrace{\underbrace{T_4}_{B_{k-1,\bar{k}}}, \underbrace{\underbrace{T_2, T_3, T_1}_{B_{k-1,k}}, \underbrace{g_1, g_2}_{B_{\overline{k-1},k}}, \mathrm{h}_k, \underbrace{T_2, g_1, g_2, T_3}_{B_{k,\overline{k+1}}}, \underbrace{T_1}_{B_{k,k+1}}}_{\Omega'_{j+1}}, \underbrace{T_5}_{B_{\bar{k},k+1}}}_{\Omega_{j+1}}, \mathrm{h}_{k+1}, T_1, T_5 \rangle,$$

$$\Delta_j = \langle T_1, T_2, T_3, T_4, \mathrm{h}_{k-1}, \underbrace{\underbrace{T_4}_{B_{k-1,\bar{k}}}, \underbrace{T_2, T_3, T_1, \mathrm{g}_1, T_3, T_1, \mathrm{g}_2, T_3, T_1}_{\Omega'_j}, \underbrace{T_5}_{B_{\bar{k},k+1}}}_{\Omega_j}, \mathrm{h}_{k+1}, T_1, T_5 \rangle.$$

Now going back to examine Equations 6 and 8, we can find that Δ_j and Δ_{j+1} only differ in the middle subsequences represented by Ω'_j and Ω'_{j+1}. This can also be seen in the example in Figure 7. The instances extracted from Δ_j would conform to Δ_{j+1} except the corresponding middle subsequence mentioned above. Remember that ρ_j conforms to Δ_j. We would try to adjust the order of the tasks of the subsequence in ρ_j which correspond to Ω'_j for the purpose that the resultant schedule ρ_{j+1} conforms to Δ_{j+1} and ρ_{j+1} is also an optimal schedule of Γ. The adjustment procedure, called the *swapping and replacing* method, applied to ρ_j is described below:

C1: for all tasks $T_y \in \Omega'_j$ such that $f_y \le d_{h_k}$, they are sorted by their ready times r_y.

C2: for all tasks $T_y \in \Omega'_j$ such that $s_y \ge r_{h_k}$, they are sorted by their deadlines d_y.

C3: a task can be sorted by C1 or C2 described above if the task satisfies both conditions.

C4: if there exists a task $T_y \in \Omega'_j$ such that $s_y < r_{h_k}$ and $f_y > d_{h_k}$, T_y is replaced by h_k.

We would like to show that the adjustment does make ρ_{j+1} conform to Δ_{j+1}. Remember that Δ_j and Δ_{j+1} only differ in the middle subsequences represented by Ω'_j and Ω'_{j+1}. We only swap and replace the tasks of ρ_j located in Ω'_j to derive ρ_{j+1}. Since ρ_j conforms to Δ_j, the head and the tail of ρ_{j+1} also conforms to Δ_{j+1}. So we only need to check the middle subsequence of ρ_{j+1} to see if the whole sequence of ρ_{j+1} conforms to Δ_{j+1}. The tasks adjusted by the swapping and replacing procedure are either in $B_{k-1,k}$ or in $B_{\overline{k-1},k}$ by Lemma 9. Let us first check the adjustment of C1. In Δ_{j+1}, the order of the tasks in $B_{k-1,k}$ can be determined arbitrarily according to rule R3, so it does not matter which task is located before which. And in Δ_{j+1}, the order of the tasks in $B_{\overline{k-1},k}$ is determined by their ready times. During the adjustment of C1, all the qualifying tasks are ordered according to their ready times. We know that the ready times of the tasks in $B_{k-1,k}$ are less than the ready times of the tasks in $B_{\overline{k-1},k}$. So, in the resultant schedule ρ_{j+1}, the tasks of $B_{k-1,k}$ are positioned before the tasks of $B_{\overline{k-1},k}$. This indicates that the order of these tasks in ρ_{j+1} after the adjustment of C1 conforms to Δ_{j+1}. For the same reason, the adjustment of C2 makes the order of the swapped tasks conform to Δ_{j+1}. In condition C4, if such a T_y exists, replacing T_y by h_k also conforms to Δ_{j+1}, which can be seen in Equation 6. Each task $T_y \in \Omega'_j$ satisfies one of the conditions by Lemma 11. Hence, all the tasks in the middle subsequence of ρ_{j+1} are adjusted in such a way that the order conforms to Δ_{j+1}. So ρ_{j+1} conforms to Δ_{j+1}.

Now we would like to show that ρ_{j+1}, in addition to conforming to Δ_{j+1}, can also be finished no later than ρ_j. We can view the tasks satisfying condition C1 as having the same virtual deadlines of d_{h_k}, because they all finish before this time instant. Hence, there is a total ordering among the tasks with virtual deadlines by weakly leading relations, which is achieved by sorting their ready times. By Lemma 10, the finish time of the resultant schedule after the adjustment of C1 would not be greater than that of the original schedule. Similarly, the tasks satisfying condition C2 can be viewed as having the same virtual ready times of r_{h_k}, because they all start after this time instant. For the same reason, the finish time of the resultant schedule after the adjustment of C2 would not be greater than that of the original schedule. In condition C3, the qualifying tasks can be sorted in either way and does not affect the result. In condition C4, if there exists a task T_y whose computation time covers the whole window of the rejected task h_k, we may as well replace T_y by h_k, and the finish time of the resultant schedule after the adjustment of C4 would not be greater than that of the original schedule. Each task $T_y \in \Omega'_j$ satisfies one of the conditions by Lemma 11. Hence, all the tasks in the middle subsequence of ρ_{j+1} are adjusted in such a way that ρ_{j+1} would be finished no later than ρ_j. Therefore, $|\rho_{j+1}| = |\rho_j|$, and $f_{\rho_{j+1}} \leq f_{\rho_j}$. Since ρ_j is an optimal schedule of Γ, ρ_{j+1} is also an optimal schedule of Γ.

How the adjustment procedure makes the finish time shorter is illustrated by Figures 8. In Figure 8(i), both T_1 and T_3 satisfy condition C3, and T_2 satisfies condition C1. The procedure of C1 is applied to all these three tasks and makes the finish time shorter. The dotted task window frame in the figure indicates that h_k is a rejected task. In Figure 8(ii), C1 is applied to the qualifying tasks T_1 and T_2. And by C4, T_3 is replaced by h_k. This makes the finish time shorter. While it is h_k that is rejected before the adjustment, it turns out that T_3, whose computation time covers the whole window of h_k, is rejected after the adjustment.

So far, we have shown that ρ_{j+1} conforms to Δ_{j+1}, and that ρ_{j+1} is also an optimal schedule of Γ. The theorem is thus verified by the induction. It deserves our attention that we do not really apply the swapping and replacing procedure to any schedule. We just want to show the existence of the optimal schedule which is ρ_{j+1} in the context. To make it clear, the structure of the theorem is illustrated in Figure 9. □

3.4.3 Corresponding Lemmas

The lemmas used by the conformation theorem are demonstrated as follows

Lemma 5 $B_{k-1,\bar{k}} \cap B_{k-1,k} = B_{k-1,k} \cap B_{\overline{k-1},k} = B_{k-1,\bar{k}} \cap B_{\overline{k-1},k} = \emptyset$

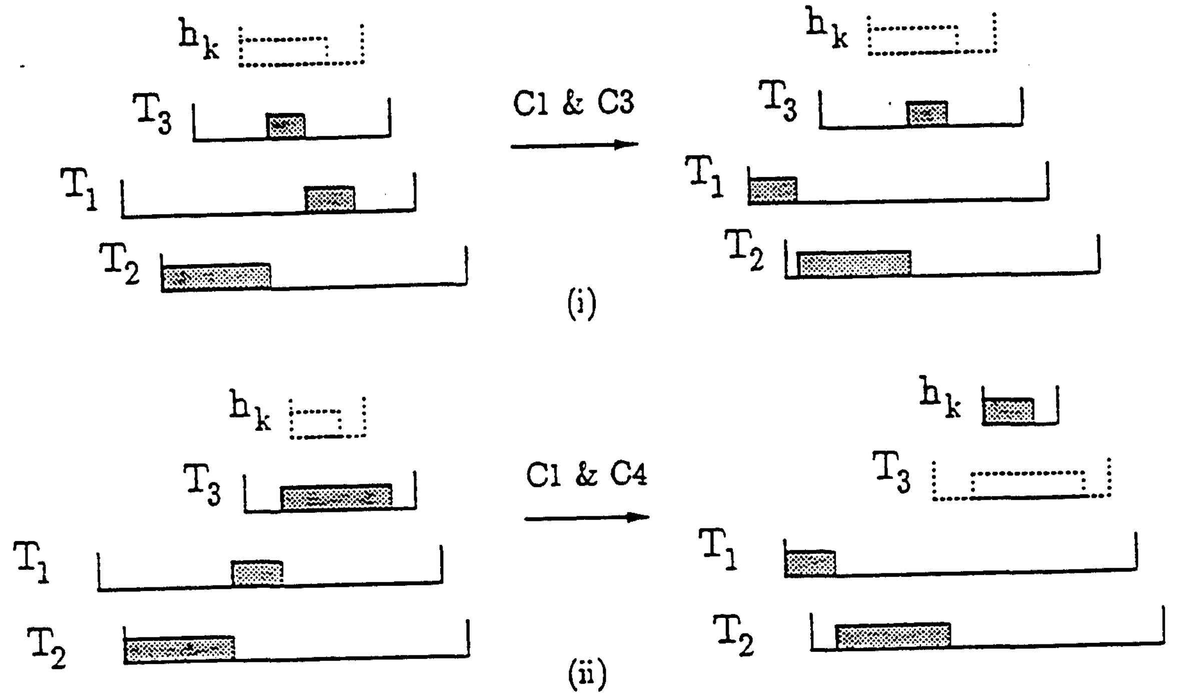


Figure 8: The swapping and replacing procedure C1, C3 and C4.

$$B_{k,\overline{k+1}} \cap B_{k,k+1} = B_{k,k+1} \cap B_{\overline{k},k+1} = B_{k,\overline{k+1}} \cap B_{\overline{k},k+1} = \emptyset.$$

PROOF: We first show that $B_{k-1,\overline{k}} \cap B_{k-1,k} = \emptyset$. Given any task $T_y \in B_{k-1,\overline{k}}$, T_y does not contain h_k by definition. Hence, T_y does not belong to $B_{k-1,k}$. So $B_{k-1,\overline{k}} \cap B_{k-1,k} = \emptyset$. The others can be proved similarly. □

Lemma 6 $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$.

PROOF: We first prove that if a task $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$, then $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$. Because T_y contains h_k by definition, T_y must have a location after h_k too by rule R2. $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}$. T_y does not belong to $B_{\overline{k},k+1}$, so $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$. We can prove similarly that if a task $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$, then $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$. So $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. □

$$\begin{tikzcd}
 & & \exists\text{ an optimal schedule} \arrow[d] \\
\rho_0 \arrow[d, "\text{Swapping \& Replacing}"'] & \Delta_0 \arrow[l, "\text{Dominance Theorem}"'] & \Gamma_0 \arrow[l, "\text{Dominant Rules}"'] \arrow[d, dotted, "\text{Adding a Task}"] \\
\vdots \arrow[d] & & \vdots \arrow[d] \\
\rho_j \arrow[r, dashed, "\textit{conforming to}", "\text{Induction Hypothesis}"'] \arrow[d, "\text{Swapping \& Replacing}"'] & \Delta_j & \Gamma_j \arrow[l, "\text{Dominant Rules}"'] \arrow[d, "\text{Adding a Task}"] \\
\rho_{j+1} \arrow[r, dashed, "\textit{conforming to}", "\text{induction for } j+1"'] & \Delta_{j+1} & \Gamma_{j+1} \arrow[l, "\text{Dominant Rules}"']
\end{tikzcd}$$

$p \xrightarrow{r} q$ `q is derived from p by method r`

$p \overset{r}{\dashrightarrow} q$ `p is related to q by relation r`

Figure 9: The Structure of the Conformation Theorem

Lemma 7 If and only if $T_y \in B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\bar{k},k+1}$ then $T_y \in \Omega_j$.

PROOF: We will first prove the "if" part. By Lemma 6, $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. So we only need to check against $B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{\bar{k},k+1}$. If $T_y \in B_{k-1,\bar{k}}$ or $T_y \in B_{k-1,k}$, then T_y has a location between the top tasks h_{k-1} and g_1 in Δ_j. This is because T_y contains h_{k-1}, T_y has a location after h_{k-1} by rule R2. So $T_y \in \Omega_j$. Then consider $T_y \in B_{\overline{k-1},k}$. T_y is either a top task or a nontop task in Δ_j. If T_y is a top task in Δ_j, T_y must be one of the g_i, $i = 1, \ldots, m$ by the definition of g_i. If T_y is a nontop task in Δ_j, it must contain at least one top task, which is a top task among $g_1, \ldots, g_m$, or h_{k+1} by referring to Equation 7. Notice that T_y must not contain h_{k-1} since $T_y \in B_{\overline{k-1},k}$. No matter whether T_y is a top or nontop task in Δ_j, T_y has a location in Ω_j by rule R2. If $T_y \in B_{\bar{k},k+1}$, then T_y has a location between the top tasks g_m and h_{k+1} in Δ_j by rule R2. So $T_y \in \Omega_j$.

Now we prove the "only if" part. Let T_y be a task located in Ω_j. If T_y is one of the top tasks of $g_1, \ldots, g_m$, T_y contains h_k. Either $T_y \in B_{k-1,k}$ or $T_y \in B_{\overline{k-1},k}$. Otherwise, T_y is a nontop task of Δ_j. By rule R2, T_y contains at least one of the top tasks of $h_{k-1}, g_1, \ldots, g_m, h_{k+1}$. If T_y contains one of $g_1, \ldots, g_m$, then T_y also contains h_k. So T_y contains either h_{k-1}, or h_k, or h_{k+1}. By rule R2, T_y should fall between h_{k-1} and h_k, and/or between h_k and h_{k+1}. So $T_y \in B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\bar{k},k+1}$. □

This lemma means that the tasks in $\Omega_{j+1} - h_k$ are exactly the same tasks which are in Ω_j.

Lemma 8 If $T_y \in \Omega'_j$, T_y contains h_k.

PROOF: We would like to show that if T_y does not contain h_k, then T_y does not belong to Ω'_j. Since g_i, $i = 1, \ldots, m$, contains h_k, that T_y does not contain h_k means that T_y does not contain g_i either. Hence T_y can not have a location in the subsequence between g_1 and g_m in Ω'_j. The only possible locations of T_y to fall in Ω'_j are either between h_{k-1} and g_1, or between g_m and h_{k+1}. If T_y falls between h_{k-1} and g_1, that T_y does not contain g_1 implies that T_y contains h_{k-1} by rule R2. That is $T_y \in B_{k-1,\bar{k}}$. A nontop task cannot have duplicate positions in the same region between two adjacent top tasks. $B_{k-1,\bar{k}}$ is located between h_{k-1} and g_1 by Equation 8. T_y does not have a location in the head of Ω'_j before g_1. For the same reason, T_y does not have a location in the tail of Ω'_j after g_m. So T_y does not belong to Ω'_j. Therefore, if $T_y \in \Omega'_j$, T_y contains h_k. □

Lemma 9 If $T_y \in \Omega'_j, T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$.

PROOF: If $T_y \in \Omega'_j$, then $T_y \in \Omega_j$. By Lemma 7, $T_y \in B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\bar{k},k+1}$ We know that T_y contains h_k by Lemma 8, so $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1}$. Also by Lemma 6, we have $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. □

Lemma 10 Assume that $S = L_{pre} \oplus L \oplus L_{post}$ is a feasible sequence, where $L = \langle T_{x_1}, T_{x_2}, \ldots, T_{x_l} \rangle$. If there exists a sequence $\bar{L} = \langle T_{y_1}, T_{y_2}, \ldots, T_{y_l} \rangle$ such that $\bar{L}$ is a permutation of L and the tasks of $\bar{L}$ are ordered by the weakly leading relation. We have $f_{L_{pre} \oplus \bar{L} \oplus L_{post}} \leq f_{L_{pre} \oplus L \oplus L_{post}}$.

PROOF: We bubble sort the tasks of L in weakly leading order. The swapping only occurs between two adjacent tasks. For each swapping, we apply the Leading Theorem to the adjacent tasks, which correspond to T_β and T_α respectively in the theorem. No other tasks lie in between the two tasks during each individual swapping. So the finish time of the resultant schedule is not greater than that of the original schedule according to the Leading Theorem. $\Box$

Lemma 11 A task $T_y \in \Omega'_j$ should satisfy one of the conditions C1, C2, C3 or C4.

PROOF: If $T_y \in \Omega'_j$, then $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$ by Lemma 9, which implies that $T_y \sqcup h_k$. We have $r_y < r_{h_k}$ and $d_y > d_{h_k}$. There are four possibilities.

(i) $s_y < r_{h_k}$ and $f_y > d_{h_k}$: C4 is satisfied.
(ii) $s_y \geq r_{h_k}$ and $f_y \leq d_{h_k}$: C3 is satisfied.
(iii) $s_y < r_{h_k}$ and $f_y \leq d_{h_k}$: C1 is satisfied.
(iv) $s_y \geq r_{h_k}$ and $f_y > d_{h_k}$: C2 is satisfied.

$\Box$

3.5 Set-Scheduler Algorithm

By Conformation Theorem, we have shown that there exists an optimal schedule which conforms to the super sequence Δ. Hence, we can use Sequence-Scheduler to schedule for each instance in the super sequence, and pick up the best one. Since Sequence-Scheduler obtains the optimal schedule for each instance, we end up with the optimal schedule for the task set. The algorithm for scheduling a task set is given in Figure 10. The Sequence-Scheduler takes $O(n^2)$ time for each instance, while there are

$$N = \prod_{q=1}^{q=t} (q+1)^{n_q}$$

instances to check in the super sequence as illustrated in the previous section. The time complexity of Set-Scheduler algorithm is thus $O(N * n^2)$.

```
Algorithm Set-Scheduler:

Input: a task set $\Gamma = \{T_1, T_2, ..., T_n\}$
Output: the optimal schedule $\rho$ for $\Gamma$
compute the super sequence $\Delta$ for $\Gamma$
$\rho := ()$
for each instance $I$ in the super sequence $\Delta$
    invoke Sequence-Scheduler to compute the optimal schedule $\sigma(n)$ of $I$
    if $(|\rho| < |\sigma(n)|)$ or
      $(|\rho| = |\sigma(n)|$ and $f_\rho > f_{\sigma(n)})$
      $\rho := \sigma(n)$
    endif
endfor
```

Figure 10: Set-Scheduler Algorithm

4 Evaluation

Experiments are conducted to compare the performance of Set-Scheduler with those of the well-known Earliest-Deadline-First and Least-Laxity-First heuristic algorithms. The relations among the tasks are important for the schedulability of the tasks. To study the differences between different cases, we allow the variation of the computation times, and the interarrival times, which are the time intervals between the ready times of two consecutive tasks. Tasks in a task set are generated in non-descending order by their ready times. The parameters of the experiments are random variables with truncated normal distribution, as shown in Figure 11. If the computation time of a task is greater than its window length, the computation time is truncated to its window length. Such a truncation is not applied to the interarrival times.

The mean of Window is fixed. Computation time ratio is the ratio of the computation time to the window length. The mean of Interarrival time ranges from 10% to 100% of the mean of Window. The standard deviation of these three random variables are set to be

parameters	*mean*
Window length	10.0
Computation time ratio	0.25 0.5 0.75
Interarrival time	1.0, 2.0, ..., 10.0

Figure 11: Parameters of the experiments

20%, 50%, and 80% of their means. For simplicity, the ratios of the three random variables are set to be the same for each individual experiment. For each experiment with different parameters, 100 task sets, each with 12 tasks, are generated for scheduling.

We compare the performance of these algorithms by (1) Percentage of accepted tasks: the number of accepted tasks by the algorithm over the number of the tasks of the optimal schedule by exhaustive search; (2) Success ratio: the number of times that the algorithm comes up with an optimal schedule in the 100 task sets; and (3) Comparisons per task set: the number of comparisons per task set that each algorithm takes. When interarrival times are small, more containing relations among tasks are likely to happen. Figure 12 shows that the heuristic algorithms perform worse under this condition and tend to reject more tasks, especially when the computation time ratio is larger. Set-Scheduler always reaches 100% acceptance rate since it is an optimal scheduler. In the figure, because the characteristics of the data with different standard deviation ratios are similar, only the data with standard deviation ratio equal to 0.8 are depicted. When success ratio is concerned, which can be seen in Figure 4, the heuristic algorithms performs even worse. Generally speaking, the heuristic algorithms can usually produce suboptimal schedules, but fail to produce the optimal ones most of the time. The search space is shown in Figure 14. Set-Scheduler performs well at the expense of the complexity, which may become very large when the interarrival times are small. The cost is more reasonable while the interarrival times between tasks are not too small.

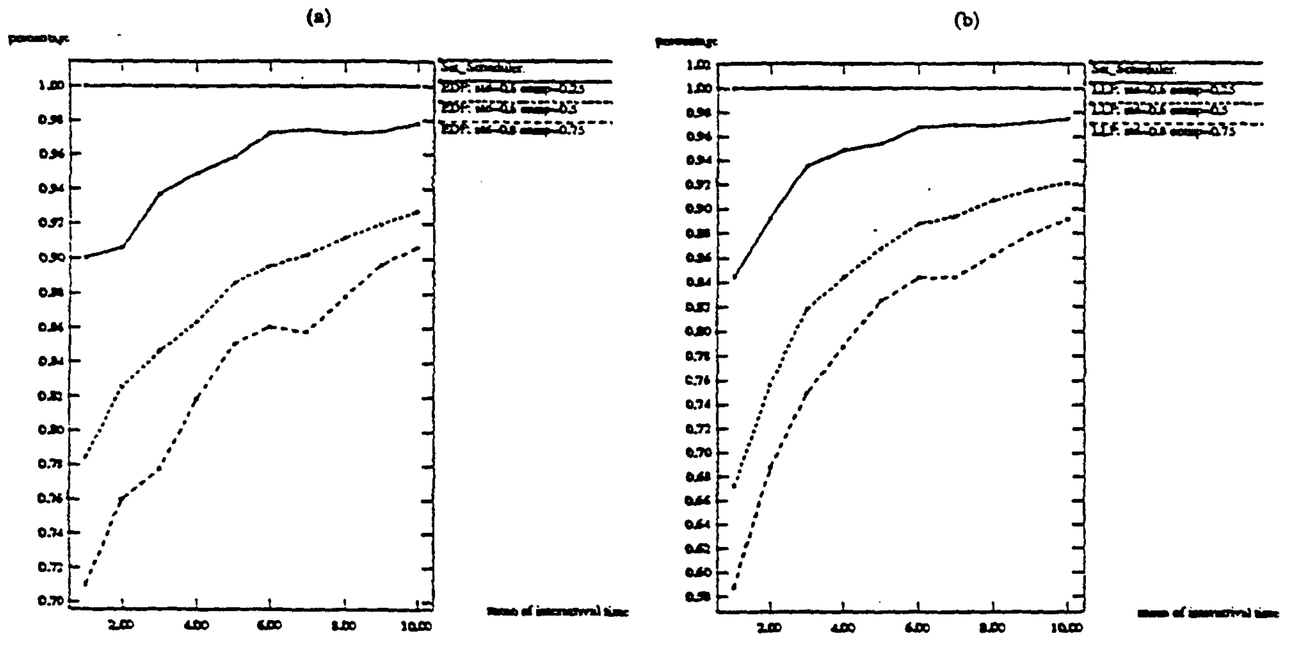


Figure 12: Percentage of accepted tasks (a) EDF (b) LLF compared with Set-Scheduler

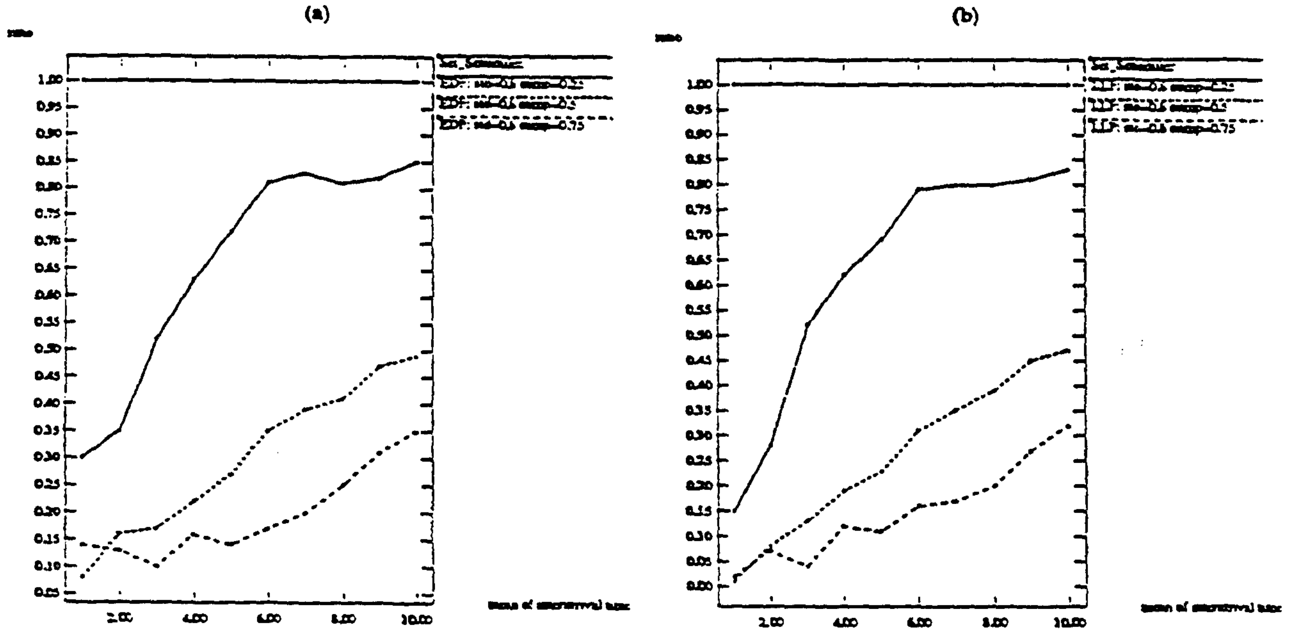


Figure 13: Success Ratio (a) EDF (b) LLF compared with Set-Scheduler

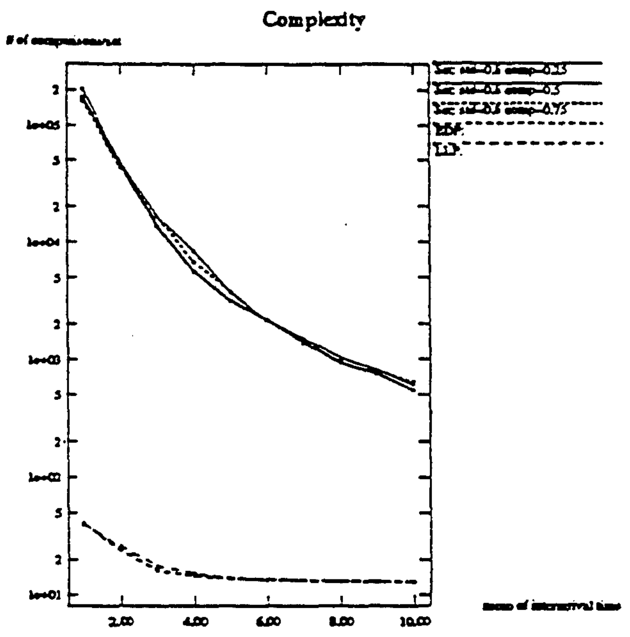


Figure 14: Number of comparisons per task set

5 Conclusion Remarks and Future Work

In this paper, we discuss the optimization techniques in real-time scheduling for aperiodic tasks in a uniprocessor system with the non-preemptive discipline. We first propose a Sequence-Scheduler algorithm to compute the optimal schedule for a sequence in $O(n^2)$ time. Then a Set-Scheduler algorithm is proposed based on the super sequence and Sequence-Scheduler algorithm. The complexity of our Set-Scheduler algorithm is $O(N * n^2)$, compared to $O(N * n)$ for the feasibility test by Erschler *et al.*, where N might be as large as exponential in the worst case. However, our simulation results show that the cost is reasonable for the average case. We explore the temporal properties concerning the optimization issues, and present several theorems to formalize the results. The study of temporal properties on a uniprocessor may serve as a base for the more complex cases in multiprocessor systems.

For the future work, we propose to incorporate the decomposition technique [18] into our scheduling algorithm. Under this approach a task set can be decomposed into subsets, which results in backtracking points to reduce the search space. This has been shown to be useful in reducing the search space substantially when the task set is well decomposable.

References

[1] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[2] J. Erschler, G. Fontan, C. Merce, and F. Roubellat. A new dominance concept in scheduling n jobs on a single machine with ready times and due dates. *Operations Research*, 31(1):114–127, Jan. 1983.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.

[4] D. W. Gillies and J. W-S. Liu. Greed in resource scheduling. In *IEEE Real-Time Systems Symposium*, pages 285–294, Dec. 1989.

[5] O. Gudmundsson, D. Mosse, K.T. Ko, A.K. Agrawala, and S.K. Tripathi. Maruti: A platform for hard real-time applications. In *Workshop on Operating Systems for Mission Critical Computing*, pages C1–C14, Sep. 1989.

[6] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[7] J. P. Lehoczky. Fixed priority scheduling of periodic tasks with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–209, Dec. 1990.

[8] J.Y. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[9] S.T. Levi, S.K. Tripathi, S.D. Carson, and A.K. Agrawala. The maruti hard real-time operating system. *ACM SIGOPS, Operating Systems Review*, 23:90–106, July 1989.

[10] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[11] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, May 1975.

[12] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, MIT Laboratory for Computer Science, May 1983.

[13] Manas Saksena and Ashok Agrawala. Temporal analysis for static hard-real time scheduling. In *Proceedings 12th International Phoenix Conference on Computers and Communications*, pages 538–544, March 1993.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[15] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Department of Computer Science, Carnegie-Mellon University, 1987.

[16] J.A. Stankovic and K. Ramamritham. The spring kernel: Operating system support for critical, hard real-time systems. In *Workshop on Operating Systems for Mission Critical Computing*, pages A1–A9, Sep. 1989.

[17] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[18] X. Yuan and A. K. Agrawala. A decomposition approach to nonpreemptive scheduling in hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 1989.

[19] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, Aug. 1987.

[20] W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling tasks with resource requirements in a hard real-time system. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.

REPORT DOCUMENTATION PAGE	Form approved OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE January 1994	3. REPORT TYPE END DATES COVERED Technical Report

4. TITLE AND SUBTITLE Optimization in Non-Preemptive Scheduling for Aperiodic Tasks	5. FUNDING NUMNBERS N00014-91-C-0195 DSAG-60-92-C-0055
6. AUTHOR(S) Shyh-In Hwang, Sheng-Tzong Cheng and Ashok K. Agrawala	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science A.V. Williams Building College Park, MD 20742	8. PERFORMING ORGANIZATION REPORT NUMBER CS-TR-3216 UMIACS-TR-94-14
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) Honeywell Inc. 3600 Technology Drive Minneapolis, MN 55148 — Phillips Laboratory Directorate of Contracting 3651 Lowry Avenue, SE Kirtland AFB, NM 87117-5777	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12.a. DISTRIBUTION/ AVAILABILITY STATEMENT	12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Real-time computer systems have become more and more important in many applications, such as robot control, flight control, and other mission-critical jobs. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. We propose a scheduling algorithm based on an analytic model. Our goal is to derive the optimal schedule for a given set of aperiodic tasks such that the number of rejected tasks is minimized, and then the finish time of the schedule is also minimized. The scheduling problem with a nonpreemptive discipline in a uniprocessor system is considered. We first show that if a total ordering is given, this can be done in *O(n2)* time by dynamic programming technique, were *n* is the size to the task set. When the restriction of the total ordering is released, it is known to be NP-complete [3]. We discuss the super sequence [18] which has been shown to be useful in reducing the search space for testing the feasibility of a task set. By extendng the idea and introducing the concept of conformation, the scheduling process can be divided into two phases: computing the pruned search space and computing the optimal schedule for each sequence in the search space. While the complexity of the algorithm in the worst case remains exponential, our simulation results show that the cost is reasonable for the average case.

14. SUBJECT TERMS Operating Systems Process, Process Management Analysis of Algorithms and Problem Complexity, Nonnumerical Algorithms and Problems	15. NUMBER OF PAGES 44
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlisted

UMIACS-TR-89-109 November, 1989
CS-TR -2345

A Decomposition Approach to Nonpreemptive Real-Time Scheduling[*]

Xiaoping (George) Yuan[†] and Ashok K. Agrawala
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Let us consider the problem of scheduling a set of n tasks on a single processor such that a feasible schedule which satisfies the time constraints of each task is generated. It is recognized that an exhaustive search may be required to generate such a feasible schedule or to assure that there does not exists one. In that case the computational complexity of the search is of the order $n!$.

We propose to generate the feasible schedule in two steps. In the first step we decompose the set of tasks into m subsets by analyzing their ready times and deadlines. An ordering of these subsets is also specified such that in a feasible schedule all tasks in an earlier subset in the ordering appears before tasks in a later subset. With no simplification of scheduling of tasks in a subset, the scheduling complexity is $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset.

The improvement of this approach towards reducing the scheduling complexity depends on the the number and the size of subsets generated. Experimental results indicates that significant improvement can be expected in most situations.

[*]This work is supported in part by contract DSAG60-87-0066 from the U.S. Army Strategic Defense Command to the Department of Computer Science, University of Maryland at College Park. The views, opinions and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision, unless so designated by other official documentation.

[†]System Design and Analysis Group and Department of Computer Science, University of Maryland, College Park, MD.

Contents

I Introduction

Consider the problem of nonpreemptive scheduling of n tasks on a single CPU of a hard real-time system. For task T_i, identified as i, the scheduling request consists of a triple $< c_i, r_i, d_i >$ where c_i is the computation time, r_i the ready time before which task i can not start, and d_i the deadline before which the computation must be completed. Time interval $[r_i,\ d_i]$ is called the *time window* denoted by w_i. The window length $|w_i|$ is $d_i - r_i$. In a hard real-time system, a schedule is called *feasible* if all tasks are processed within their individual windows.

The result of the scheduling process is a schedule in which for any task i, a start time s_i and a finish time f_i is identified, where $f_i = s_i + c_i$. Clearly, a schedule is feasible, if for every task i,

$$r_i \leq s_i \leq d_i - c_i. \tag{1}$$

The scheduling process is not preemptive only if for any two tasks i and j,

$$s_i \leq s_j \Rightarrow s_i + c_i \leq s_j. \tag{2}$$

In other words, when task i is scheduled, a span of nonpreemptable processing time, c_i, is allocated for it. No other task may be in execution during that time span. Thus the scheduling problem is to find a mapping from a task set $\{i\}$ to a start time set $\{s_i\}$, such that constraints in (1) and (2) are met. Note that for a given set of tasks $\{i\}$, there may be none, one or many feasible schedules.

In general the nonpreemptive real-time scheduling problem is known to be NP-complete [Gare79]. To find a feasible schedule, the number of schedules to be examined is $O(n!)$, which we count as the *scheduling complexity*. Heuristic techniques can be used [Ma84, McMa75, Mok83, Zhao87] to reduce the complexity. This reduction, however, is achieved at the cost of obtaining a potentially sub-optimal solution. That is, when looking for feasible schedules, heuristic techniques may not yield a feasible schedule, even though one exists. Schedules based on the earliest-deadline-first, or minimum-laxity-first rules are examples of such heuristics used in scheduling.

An alternate approach is to develop analytical methods for scheduling [Ersc83, Liu73]. This approach analyzes the relationships among real-time tasks and schedules. The purpose is to precisely determine optimal task schedules, or narrow the search scope from the original search space.

The objective of this research is to develop *correct* and efficient algorithms for nonpreemptive real-time scheduling. We call a scheduling algorithm *correct*, if whenever a feasible schedule exists, the algorithm can find it.

In this paper, we present an analytical decomposition approach for real-time scheduling. The strategy is to divide a set of tasks into a sequence of subsets, such that the search for feasible schedules is only performed within each subset. The decomposition technique used for generating the sequence of subsets assures that in a feasible schedule all tasks in a subset earlier in the sequence are scheduled before any task in a later subset. Backtracking in the search is bounded within each subset, which significantly reduces the scheduling complexity.

There are several different strategies which can be used to subset tasks. The decomposition strategy discussed in this paper is to use a relation called the *leading relation* which depends on the tasks' relative window positions.

We performed an experiment which examined the number and size of subsets with regard to the number of tasks, task arrival rate, and window length. We found that, in general, the number of tasks in any subset is independent of the total number of tasks to be scheduled, if the task window lengths are bounded. The decomposition scheduling is a polynomial computation. As a consequence, the decomposition method is very practical for the implementation.

In section II we present some basic notions used in the paper. In section III we discuss a case where all the tasks have the leading relation with each other. Our approach of decomposition scheduling is introduced in section IV along with concepts of the single schedule subset and decomposed leading schedule sequence. We present our experiment results in section V. Our conclusion and future research in section VII.

II Background

If we consider any two tasks i and j, they must have one of these three relations:

1. *leading* - $i \prec j$ (or $j \prec i$), where if $r_i \leq r_j$, $d_i \leq d_j$ and $w_i \neq w_j$.
2. *matching* - $i \| j$, if $r_i = r_j$ and $d_i = d_j$.
3. *containing* - $i \sqcup j$ (or $j \sqcup i$), if $r_i < r_j$ and $d_j < d_i$.

These three relations are shown in Fig. 1. It is easy to see that the leading, matching and containing relations are all transitive. Additionally, if $i \| j$ or $i \sqcup j$, we say that i and j are *concurrent*.

A *length* is associated with a schedule which is the finish time of the last task in the schedule. One example is shown in Fig. 2.

The concept of *dominance* was introduced in [Ersc83], and we will use it later in the discussion.

Definition 1 For two schedules S_1 and S_2, S_1 *dominates* S_2 if and only if:

$$S_2 \ feasible \Rightarrow S_1 \ feasible.$$

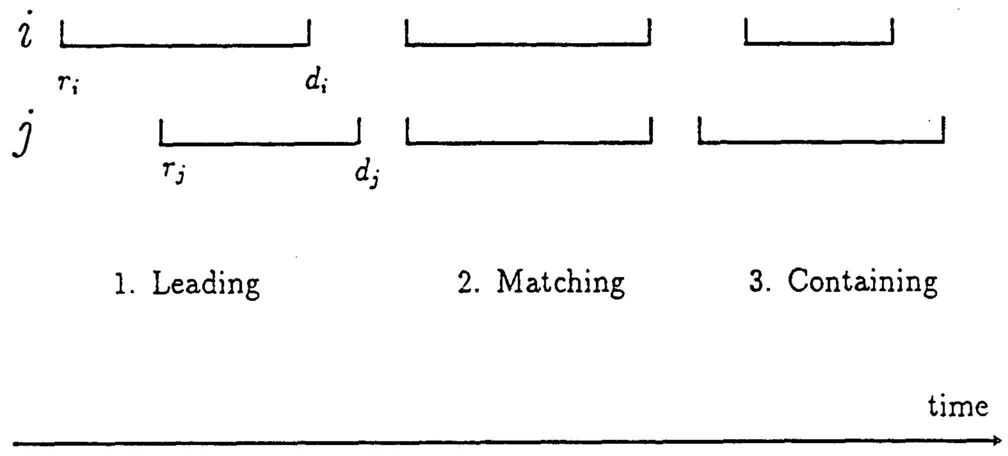


Figure 1: Task window relations

Definition 2 A set of schedules, S, is *dominant* if $\forall S_2 \notin S, \exists S_1 \in S$ such that S_1 dominates S_2.

A schedule is *dominant* if it dominates all other schedules.

III The Leading Schedule Sequence

Let us consider the case where for a set of task $\{i\}$, every pair of tasks in this set has a leading relation, i.e. $i \prec j$ or $j \prec i$, for every $i, j, i \neq j$.

Based on the leading relation we can define a total order of tasks for the set. We define the *leading schedule sequence* (LSS) to be a sequence of tasks in which tasks are in order according to the leading relation, that is, for any i and j, $i \dot{\rightarrow} j \iff i \prec j$, where $i \dot{\rightarrow} j$ means that i is scheduled in front of j.

Theorem 1 For a set of tasks all of which have a pairwise leading relation, the schedule where tasks are sequenced in order of the leading schedule sequence is a dominant one.

Proof: We prove this theorem by construction.

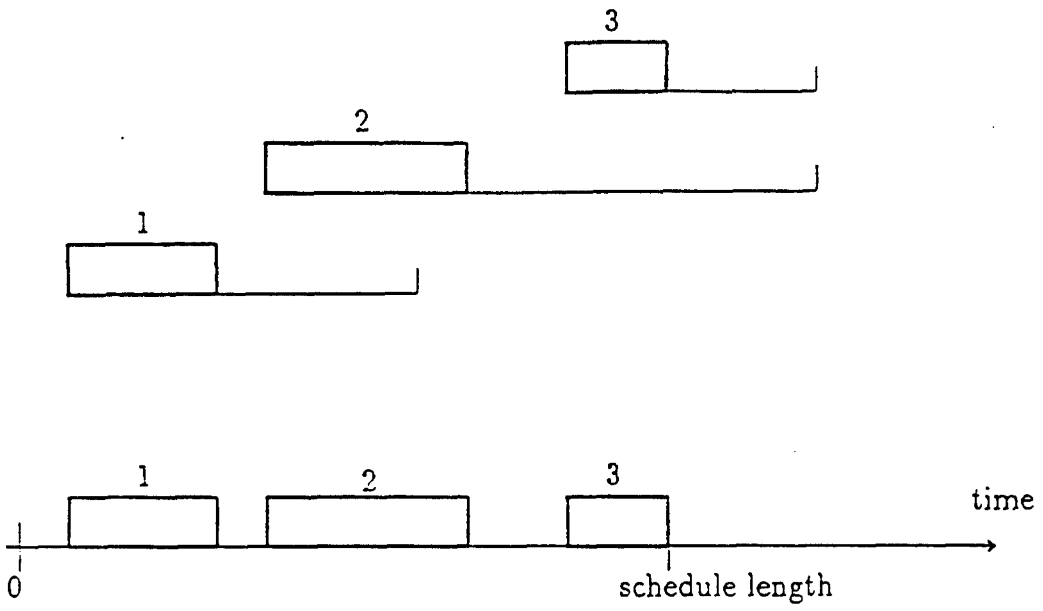


Figure 2: An example of one schedule

Suppose this set of tasks has a feasible schedule S in which tasks occur in a different sequence than the leading schedule sequence. When we examine this schedule, let i and j be the first pair of tasks that are not ordered by the leading relation, i.e. $i \prec j$, but $j \doteq i$. From the leading relation we know that $r_i \leq r_j$ and $d_i \leq d_j$.

Since j and i are the first such pair, deadlines of all tasks between j and i in S must be greater than or equal to d_j as well as d_i. In S, let us construct another schedule S' by moving i from the current position in S to the position just in front of j. The start time and finish time of tasks between j and i including j will be increased by no more than c_i. And so, no task between j and i including j will finish later than d_i. Meanwhile, the rest of this schedule is unchanged. Thus if S is feasible, the new schedule S' will be feasible too.

By repeating the process of constructing S' from S, we obtain a schedule which has all tasks ordered according to the leading relation, such that if the original schedule is feasible, so is the constructed one. □

Thus if there exists a set of feasible schedules, the set must contain schedules that are conforming to the order of the leading schedule sequence. The result can be generalized to the situation where there exist matching windows. The generalization is to combine the tasks with the same window into one task whose computation time is the sum of the computation times of all these tasks.

We will see that the above leading schedule sequence is a special case of the decomposed leading schedule sequence introduced in the next section.

IV Task Decomposition

A. Philosophy

To solve the general real-time scheduling problem with n tasks, the number of schedules to be examined can be as much as $O(n!)$. However, taking a closer look, we find that every task has an important property called the *locality* of a task, that is, a task is time-bounded by its time window. Furthermore, if any two task windows are not overlapping, there is only one possible order for them. The above facts motivate us to separate the tasks into subsets according to their different time localities.

The *decomposition scheduling* can be divided into two steps: decomposition and scheduling.

First, a set of n tasks is decomposed into a sequence of m subsets such that the orders of subsets are fixed. The order of a task is determined only relative to the other tasks within its own subset. The sequence of the subsets is called the *decomposed schedule sequence*. The decomposition should be so developed that the schedulability of tasks is not damaged at all. The decomposition by using the leading relation introduced in this paper shows this property.

The second step is to schedule the subsets in the sequence order. It always selects a schedule for each subset with the shortest length, so that when a subset is scheduled, the time span available for it is maximized. In this way, the total number of schedules to be examined is only $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset ($\sum_{i=1}^{m} n_i = n$).

The only remaining problem is how to decompose a set of tasks into a sequence of subsets of tasks such that a feasible schedule is guaranteed to be found if one exists. In the rest of this paper, we outline how to use the leading relation as a means to divide the task set.

B. Decomposition Scheme

A set of tasks is called the *single schedule subset* (*sss*), represented as τ, if

$$\forall i \in \tau\ \exists j \in \tau\ (i \sqcup j) \vee (j \sqcup i) \vee (i \| j).$$

In other words, each task window is contained in the window of another task, contains the window of another task, or matches the window of another task in the subset.

Given a set of tasks $\{i\}$, we can decompose it into a sequence of single schedule subsets $\tau^1, \tau^2, \cdots, \tau^k$ such that all the tasks in τ^i are leading to all the tasks in τ^{i+1}.

The *decomposed leading schedule sequence* ($DLSS$) is defined to be a sequence of single schedule subsets, denoted as:

$$DLSS = \tau^1 \circ \tau^2 \circ \cdots \circ \tau^m,$$

such that $\forall k^i \in \tau^i \, \forall k^j \in \tau^j \; k^i \prec k^j$, for $1 \leq i < j \leq m$, (denoted as $\tau^i \prec \tau^j$), and τ^i can not be further decomposed, for $i = 1, \cdots, m$. Symbol $\circ$ represents a concatenating operation.

Note that if a task in τ^i does not lead another task in τ^j for $i < j$, they must have a matching or containing relation. If this happens, τ^i and τ^j can not be different single schedule subsets. Clearly, all n tasks may belong to a single schedule subset.

Theorem 2 The set of schedules conforming to the decomposed leading schedule sequence is dominant.

Proof: Assume that if there are two tasks $k^i \in \tau^i$ and $k^j \in \tau^j$, where $\tau^i \prec \tau^j$. There is no common concurrent task with both k^i and k^j. k^j is positioned in front of k^i in a feasible schedule (S). Specifically, $S = (\cdots) \circ (k^j \circ \cdots k^i) \circ (\cdots)$. Let us define $S' = (k^j \circ \cdots k^i)$ for abbreviation ($S = (\cdots) \circ S' \circ (\cdots)$). The new schedule created by exchanging k^i's position with k^j's is still feasible.

Without loss of generality, suppose that k^i and k^j are the first such pair in S. Tasks between k^j and k^i are led by k^j, or concurrent with k^j, but not leading to and not concurrent with k^i. Since $k^i \prec k^j$ (i.e. $r_i \leq r_j$), switching k^i's and k^j's positions will not increase the finish time of S', which is defined as the finish time of the last task in S'. All the tasks between k^i and k^j, including k^j, are led by k^i, i.e. having deadlines greater than or equal to d_{k^i}. If S is feasible with k^i as the last task in S', it will be still feasible after the switching. □

Note that if the set of schedules that are following the decomposed leading schedule sequence is *empty*, there is no feasible schedule available for the tasks to be scheduled.

C. Decomposition Algorithm

Decomposing a set of tasks into single schedule subsets, the algorithm starts with the tasks having been sorted by their ready times (using their deadlines if their ready times are the same).

The algorithm uses one single loop to determine which single schedule subset the current task should belong to. The loop consists of two parts. The first part is a *while* loop which merges single schedule subsets into one, if the current task is contained by them. The second part decides whether the current task can form a new single schedule subset, or join with another single schedule subset.

The Leading-relation Decomposition Algorithm

```
begin
        /* Initialization. */
        $k = 1$; $\tau^1 = \{1\}$;
        $r_{\tau^1} = r_1$; $d_{\tau^1} = d_1$;
        for $i = 2$ to $n$ do /* Go over the task list. */
            $l = k - 1$;      /* $l$ is the index of single schedule subsets. */
            continue = TRUE;
            while $(l > 0) \wedge (continue)$ do
                /* Merge single schedule subsets if the current task is concurrent
                   with tasks in different subsets. */
                if $(d_{\tau^l} > d_i)$
                   $\tau^l = \tau^k \cup \tau^l$;
                   $d_{\tau^l} = d_{\tau^k}$;
                   $k = l$;
                else
                   continue = FALSE;
                $l = l - 1$;
            od
            if $(r_{\tau^k} \le r_i) \wedge (d_i < d_{\tau^k})$
                /* The current task is concurrent with tasks in the current subset.*/
                $\tau^k = \tau^k \cup \{i\}$;
            else if $(r_{\tau^k} \le r_i) \wedge (d_{\tau^k} \le d_i)$
                /* The current task is led by all the tasks in the current subset.
                   A new single schedule subset is created only containing the current task.*/
                $k = k + 1$;
                $\tau^k = \{i\}$;
                $r_{\tau^k} = r_i$;
                $d_{\tau^k} = d_i$.
        od
end
```

In this algorithm, the outer loop is executed n times. The *while* loop is executed no more than the number of time proportional to n in total, since no more than n subsets can be merged during the whole execution of the algorithm with n tasks. Thus the complexity of this algorithm is only $O(n)$. If we count in the sorting complexity, the decomposition will cost no more than $O(n \log n)$.

D. Scheduling Scheme

After tasks has been decomposed into a sequence of subsets, scheduling should be performed on each subset in the sequence order, such that the schedule on each subset is of the shortest length. A brute force method is to give an exhaustive search whose computational complexity amounts to $O(n_i!)$, where n_i is the number of tasks in the ith subset.

In [Yuan89b], other scheme is explored for scheduling a subset. The method is to first build a super-sequence where tasks may have several occurrences. The occurrence of a task is decided by its relative window position in the subset. Selecting one occurrence for every task in the super-sequence forms a schedule. A complete search costs $O(n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})})$ in the worst case. When we made a few calculation samples of $n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})}$ with n_i less than 100, $n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})}$ is a much smaller number than $n_i!$, as shown in the cited paper.

Since the set of schedules following the decomposed leading schedule sequence is dominant, and since the subsets are scheduled in the sequence order with their shortest length, it is proved that the decomposition scheduling with the leading relation is correct [Yuan89b].

V Empirical Study

A. Experiment

In order to observe the behavior of the number of tasks in a single schedule subset and number of the subsets to be created with regard to the number of tasks to be scheduled, task arrival rate, and task window length, we conduct an experiment as an example to see the feasibility of our approach for practical implementation.

The outputs we are interested in are:

1. the number of single schedule subsets (*sss*),

2. the number of window concurrences,

3. the maximum number of tasks in single schedule subsets,

4. the minimum number of tasks in single schedule subsets, and

5. the average number of tasks in single schedule subsets.

One window concurrence is counted for any two tasks i and j if i and j have a concurrent relation. We call the number of tasks in a single schedule subset as the *size* of the subset.

Meanwhile, we change the following parameters independently to watch the changes in the outputs,

1. the number of total tasks,
2. task arrival rate, and
3. window length.

The data is shown in Table 1-4[1] in the end of this paper. Following are basic rules in the experiment.

1. The computation time is uniformly distributed over $(0, \alpha]$.
2. The task interarrival is uniformly distributed over $[0, \beta)$. The arrival rate is $2/\beta$.
3. The window length is also randomly created by controlling the laxity for each task. The laxity of a task is the difference between its window length and its computation time. The laxity is uniformly distributed $[0, \gamma)$. The distribution guarantees the window length greater than the computation time for the task.

We notice that the arrival rate should be less than or equal to the service rate, otherwise, there are congestions in the system, which will result in deadline-missing. In other words, $2/\beta \leq 2/\alpha$. That is,

$$\alpha \leq \beta.$$

The random numbers are provided by function *drand*() in the UNIX operating system. The numbers are uniformly distributed over $[0, 1)$ [Stev86].

In the experiment, we found that the minimum size of single schedule subsets is always one.

B. The Result Explanation and Observation

From the experiment results , we found that when the average window length increases (γ increases), the number of single schedule subsets reduces and the maximum size of single schedule subsets slightly increases. The result is expected, since the larger some task

[1]In the tables, *number* is represented by num. *Window* by W. *Concurrences* by concurr. *Average* by avg. The *Single schedule subset* by sss.

windows are, the more tasks may be concurrent with them. These tasks may be in the same single schedule subset.

When β increases, that is, the arrival rate decreases, the number of single schedule subsets increases, and maximum size of single schedule subsets decreases. The result is also expected, since when the arrival rate decreases, the opportunity of tasks concurrent with each other decreases too. Most tasks have the leading relation with each other.

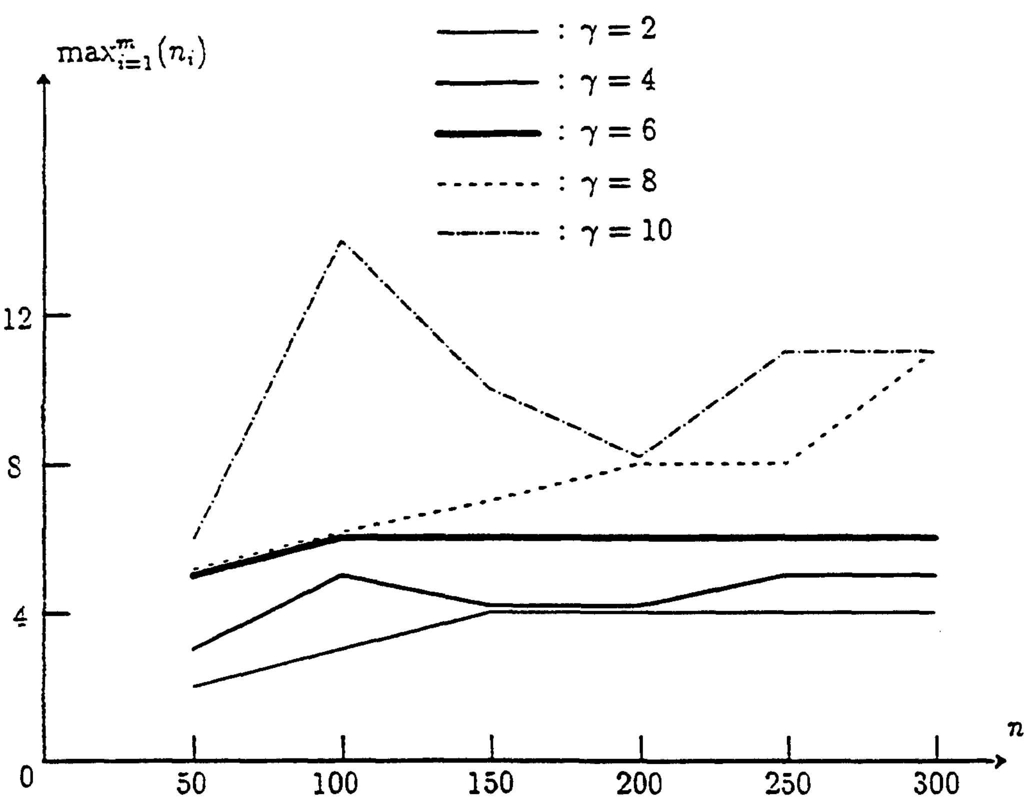


Figure 3: The relationship between the size of single schedule subsets and the number of tasks with regard to the laxity parameter γ, where $\alpha = 4,\ \beta = 4$.

Fig. 3 shows the relationship between the maximum size of single schedule subsets and the number of tasks to be scheduled. From the experiment, we found that the size of a single schedule set never exceeds 14 even when there are 300 tasks being scheduled. The observation indicates that for most cases $max_{i=1}^{m}(n_i)$ is a constant.

We show the relationship between the number of subsets (m) and number of tasks to be scheduled in Fig. 4 and Fig. 5 with regard to different window length and arrival rate distributions.

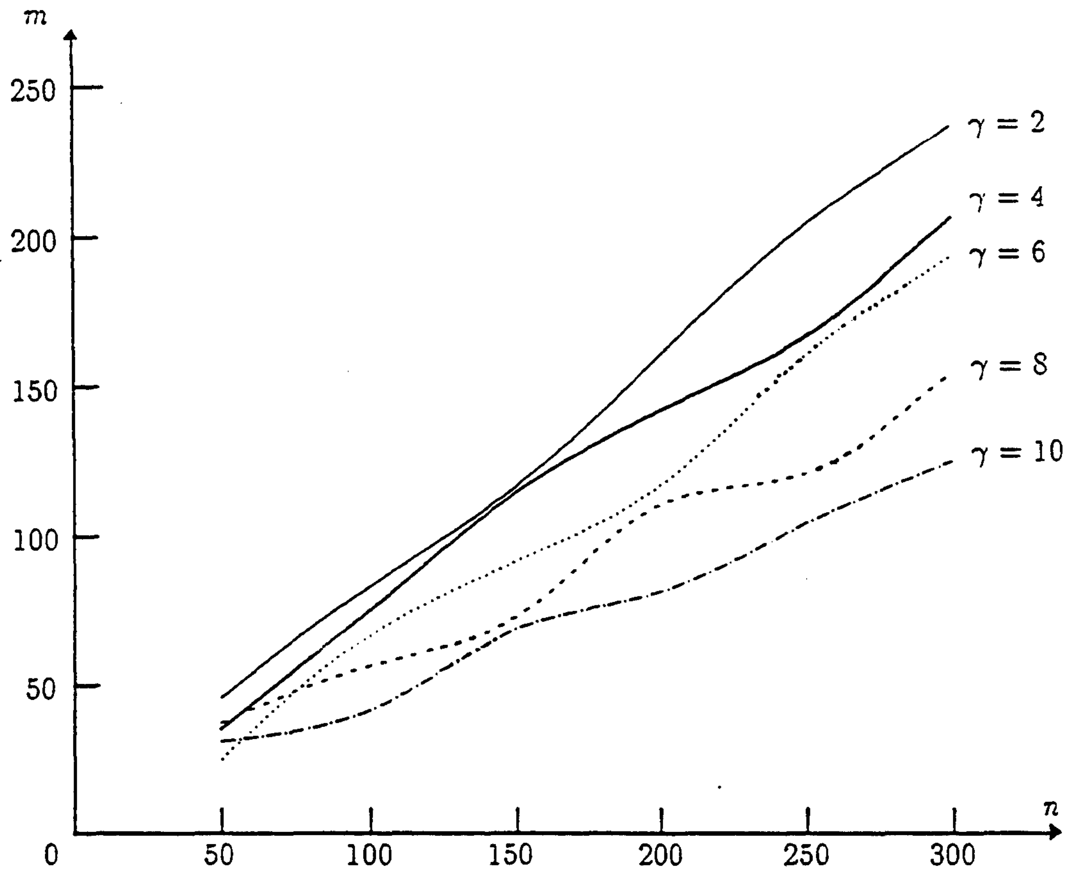


Figure 4: The relationship between the number of single schedule subsets and the number of tasks with regard to the laxity parameter γ, where $\alpha = 4$, $\beta = 4$.

VI Final Remarks

In this paper, we examine the problem of nonpreemptive scheduling of n tasks on a single CPU in hard real-time systems. We propose a correct decomposition strategy for the scheduling. The strategy significantly reduces the scheduling complexity for most cases.

In this paper we have examined a decomposition technique based only on the windows of tasks. By taking into account the computation time requirements, the decomposition can be made stronger [Yuan89a]. The decomposition approach may also be extended to consider precedence and other dependences among tasks. This aspect of decomposition technique needs further study.

Acknowledgements

Thanks to S. Mukherjee for his help in coding the decomposition algorithm for our

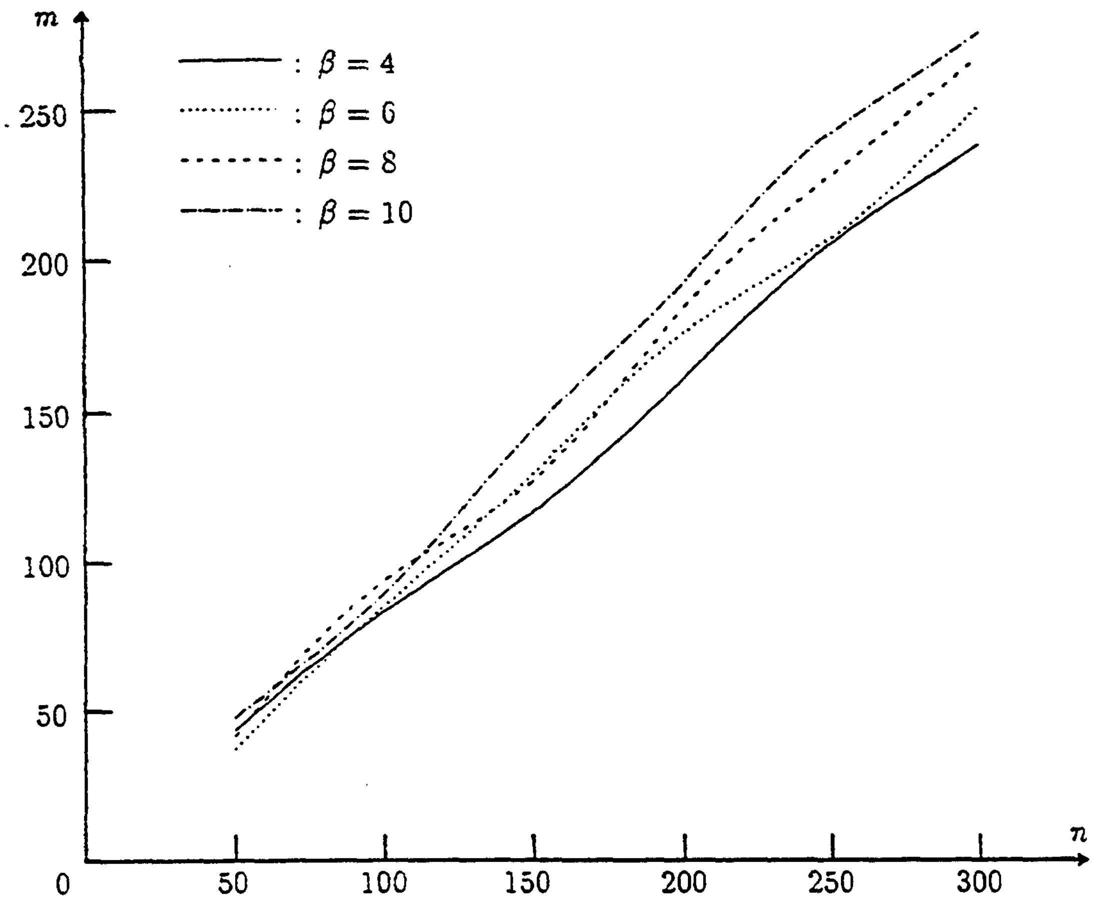


Figure 5: The relationship between the number of single schedule subsets and the number of tasks with regard to the arrival rate parameter β, where $\alpha = 4$, $\gamma = 2$.

experiment.

References

[Ersc83] Erschler, J., Fontan, G., Merce, C., and Roubellat, F., "A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates", *Operations Research*, Vol. 31, No. 1, pp. 114–127, Jan. 1983.

[Gare79] Garey, M. R. and Johnson, D. S., *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[Liu73] Liu, C. L. and Layland, J., "Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*., Vol. 20, pp. 46–61, Jan.

1973.

[Ma84] Ma, P. R., "A Model to Solve Timing-Critical Application Problems in Distributed Computing Systems", *IEEE Computer*, Vol. 17, pp. 62–68, Jan. 1984.

[McMa75] McMahon, G. and Florian, M., "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness", *Operations Research*, Vol. 23, No. 3, pp. 475–482, May 1975.

[Mok83] Mok, A. K., "Fundamental Design Problems for the Hard Real-time Environments", May 1983, MIT Ph.D. Dissertation.

[Stev86] Steven V. Earhart, Ed., *UNIX Programmer's Manual: System Calls and Library Routines*, Volume Volume 2, CBS College Publishing, 1986.

[Yuan89a] Yuan, X. and Agrawala, A. K., *Decomposition with a Strongly-Leading Relation for Hard Real-Time Scheduling*, Technical Report to be Published, Dept. of Computer Science, Univ. of Maryland, Coll. Pk., MD 20742. Mar. 1989.

[Yuan89b] Yuan, X. and Agrawala, A. K., *Scheduling Real-Time Task in Single Schedule Subsets*, Technical Report to be Published, Dept. of Computer Science, Univ. of Maryland, Coll. Pk., MD 20742, Mar. 1989.

[Zhao87] Zhao, W., Ramamritham, K., and Stankovic, J. A., "Scheduling Tasks with Resource requirements in a Hard Real-Time System", *IEEE Trans. on Soft. Eng.*, Vol. SE-13, No. 5, pp. 564–577, May 1987.

Table 1: $\alpha = 4$, $\beta = 4$

num of tasks	γ	avg W. length	num of concurr.	num of *sss*	*sss* size max	*sss* size avg
50	2	3.02	5	45	2	1.11
	4	4.33	18	35	3	1.43
	6	5.08	27	25	5	2.00
	8	6.53	14	36	5	1.39
	10	7.03	26	30	6	1.67
100	2	3.11	18	82	3	1.22
	4	4.18	28	74	5	1.35
	6	4.34	37	68	6	1.47
	8	5.84	56	55	6	1.82
	10	7.01	77	41	14	2.44
150	2	3.09	32	118	4	1.27
	4	3.78	40	114	4	1.32
	6	4.93	68	92	6	1.63
	8	5.85	91	73	7	2.05
	10	6.56	111	68	10	2.21
200	2	3.11	41	159	4	1.26
	4	3.67	64	144	4	1.39
	6	4.94	98	116	6	1.72
	8	5.94	102	113	8	1.77
	10	7.27	161	81	8	2.47
250	2	3.02	46	204	4	1.23
	4	3.82	93	167	5	1.50
	6	4.83	107	160	6	1.56
	8	6.33	162	121	8	2.07
	10	7.29	186	103	11	2.43
300	2	2.98	66	237	4	1.27
	4	3.90	105	205	5	1.46
	6	5.09	125	194	6	1.55
	8	6.21	179	153	11	1.96
	10	6.92	228	125	11	2.40

Table 2: $\alpha = 4$, $\beta = 6$

num of tasks	γ	avg W. length	num of concurr.	num of *sss*	*sss* size	
					max	avg
50	2	2.79	13	38	3	1.32
	4	4.22	7	43	3	1.16
	6	4.45	16	36	3	1.39
	8	6.21	21	31	4	1.61
	10	6.82	23	30	5	1.67
100	2	2.96	16	85	3	1.18
	4	3.93	16	85	3	1.18
	6	3.93	16	85	3	1.18
	8	6.32	31	73	4	1.37
	10	7.21	52	58	7	1.72
150	2	3.29	20	130	4	1.15
	4	3.98	29	125	5	1.20
	6	5.33	42	111	4	1.35
	8	5.90	55	100	5	1.50
	10	6.96	77	83	8	1.81
200	2	3.04	25	175	4	1.14
	4	3.90	48	153	3	1.31
	6	5.18	55	148	9	1.35
	8	5.95	87	128	8	1.56
	10	7.18	84	128	6	1.56
250	2	2.93	45	206	3	1.21
	4	4.08	55	200	6	1.25
	6	5.10	55	197	5	1.27
	8	6.15	78	175	8	1.43
	10	6.78	98	162	7	1.54
300	2	3.00	54	250	4	1.20
	4	4.13	47	254	5	1.18
	6	4.97	67	236	4	1.27
	8	5.82	96	214	5	1.40
	10	6.97	161	159	8	1.89

Table 3: $\alpha = 4,\ \beta = 8$

num of tasks	γ	avg W. length	num of concurr.	num of *sss*	*sss* size max	*sss* size avg
50	2	3.24	9	42	3	1.19
	4	3.56	5	45	3	1.11
	6	4.67	11	40	3	1.25
	8	5.95	3	47	2	1.06
	10	6.51	20	32	5	1.56
100	2	2.99	6	94	2	1.06
	4	4.20	13	87	3	1.15
	6	5.04	26	75	3	1.33
	8	5.77	29	73	5	1.37
	10	6.91	33	70	5	1.43
150	2	2.99	23	127	4	1.18
	4	4.01	28	126	4	1.19
	6	5.27	28	122	3	1.23
	8	6.00	32	120	5	1.25
	10	7.22	39	112	6	1.34
200	2	3.09	19	182	3	1.10
	4	3.94	18	182	3	1.10
	6	4.87	31	171	4	1.17
	8	5.91	51	154	5	1.30
	10	6.96	78	130	6	1.54
250	2	2.91	23	227	3	1.10
	4	4.01	48	206	4	1.21
	6	5.01	52	204	5	1.23
	8	5.84	56	195	5	1.28
	10	7.48	60	191	4	1.31
300	2	2.99	35	267	4	1.12
	4	4.05	30	270	3	1.11
	6	5.07	60	241	4	1.24
	8	5.99	63	240	4	1.25
	10	7.13	110	199	6	1.51

Table 4: $\alpha = 4$, $\beta = 10$

num of tasks	γ	avg W. length	num of concurr.	num of *sss*	*sss* size max	*sss* size avg
50	2	3.44	2	48	2	1.04
	4	4.11	5	45	3	1.11
	6	5.50	4	46	3	1.09
	8	6.69	11	41	4	1.22
	10	6.61	15	38	5	1.32
100	2	3.02	[illegible]	89	4	1.12
	4	4.01	14	87	4	1.15
	6	4.95	21	80	4	1.25
	8	6.59	18	82	3	1.22
	10	6.68	25	76	5	1.32
150	2	3.00	6	144	2	1.04
	4	3.98	11	139	3	1.08
	6	4.72	11	139	2	1.08
	8	5.91	37	117	5	1.28
	10	7.10	43	109	6	1.38
200	2	3.00	18	182	3	1.10
	4	3.97	27	174	5	1.15
	6	5.19	23	177	3	1.13
	8	6.16	34	167	4	1.20
	10	6.90	52	153	4	1.31
250	2	2.94	24	228	3	1.10
	4	4.12	32	218	3	1.15
	6	5.10	43	213	4	1.17
	8	6.07	45	206	3	1.21
	10	6.96	51	200	4	1.25
300	2	3.05	27	274	3	1.09
	4	4.07	21	280	3	1.07
	6	5.04	54	251	6	1.20
	8	6.18	46	254	3	1.18
	10	6.61	61	243	8	1.23

REPORT DOCUMENTATION PAGE			Form approved OMB No 074-0188
1. AGENCY USE ONLY (leave blank)	2. REPORT DATE Novemebr 1989	3. REPORT TYPE END DATES COVERED Technical Report	
4. TITLE AND SUBTITLE A Decomposition Approach to Nonpreemptive Real-Time Scheduling			5. FUNDING NUMNBERS DASG60-87-X-0066
6. AUTHOR(S) Xiaoping (George) Yuan, Ashok K. Agrawala, and Manas C. Saksena			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science A.V. Williams Building College Park, MD 20742			8. PERFORMING ORGANIZATION REPORT NUMBER CS-TR-89-109 UMIACS-TR-2345
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Army Strategic Defense Command Contr. & Acq. Mgt. Office CSSD-H-CRS, P.O. Box 1500 Huntsville, AL 35807-3801			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12.a. DISTRIBUTION/ AVAILABILITY STATEMENT			12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Consider the problem of scheduling a set of n tasks on a uniprocessor such that a feasible schedule that satisfies each task's time constraints is generated. Traditionally, researchers have looked at all the tasks as a group and applied heuristic or enumeration search to it. We propose a new approach called the *decomposition scheduling* where tasks are decomposed into a sequence of subsets. Tasks are scheduled in each subset independently in the order of the sequence. It is proved that a feasible schedule can be generated as long as one exists for the tasks. In addition, the overall scheduling cost is reduced to the sum of the scheduling costs of the tasks in each subset.

Simulation experiments were conducted to analyze the performance of decomposition scheduling approach. The results show that in many cases decomposition scheduling performs better than the traditional branch-and-bound algorithms in terms of scheduling cost, and heuristic algorithms in terms of percentage of finding feasible schedules over randomly-generated task sets.

14. SUBJECT TERMS			15. NUMBER OF PAGES 17
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlisted

MSN 7540-01 280-5500 Standard Form 298 (Rev 2-89)

Let us consider the problem of scheduling a set of n tasks on a single processor such that a feasible schedule which satisfies the time constraints of each task is generated. It is recognized that an exhaustive search may be required to generate such a feasible schedule or to assure that there does not exists one. In that case the computational complexity of the search is of the order $n!$.

We propose to generate the feasible schedule in two steps. In the first step we decompose the set of tasks into m subsets by analyzing their ready times and deadlines. An ordering of these subsets is also specified such that in a feasible schedule all tasks in an earlier subset in the ordering appears before tasks in a later subset. With no simplification of scheduling of tasks in a subset, the scheduling complexity is $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset.

The improvement of this approach towards reducing the scheduling complexity depends on the the number and the size of subsets generated. Experimental results indicates that significant improvement can be expected in most situations.

Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation*

Cengiz Alaettinoğlu[†] A. Udaya Shankar

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

October 1993

Abstract

A simple approach to inter-domain routing is domain-level source routing with link-state approach where each node maintains a domain-level view of the internetwork. This does not scale up to large internetworks. The usual scaling technique of aggregating domains into superdomains loses ToS and policy detail.

We present a new viewserver hierarchy and associated protocols that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and (3) scales well to large number of domains without losing detail. Domain-level views are maintained by special nodes called viewservers. Each viewserver maintains a domain-level view of a surrounding precinct. Viewservers are organized hierarchically. To obtain domain-level source routes, the views of one or more viewservers are merged (upto a maximum of twice the levels in the hierarchy).

We also present a model for evaluating inter-domain routing protocols, and apply this model to compare our viewserver hierarchy against the simple approach. Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*packet networks; store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.m [Routing Protocols]; F.2.m [Computer Network Routing Protocols].

* This work is supported in part by ARPA and Philips Labs under contract DASG60-92-0055 to Department of Computer Science, University of Maryland, and by National Science Foundation Grant No. NCR 89-04590. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, or the U.S. Government.

[†]The author is also supported by University of Maryland Graduate School Fellowship and Washington DC Chapter of the ACM Samuel Alexander Fellowship.

1 Introduction

A computer internetwork, such as the Internet, is an interconnection of backbone networks, regional networks, metropolitan area networks, and stub networks (campus networks, office networks and other small networks)[1]. Stub networks are the producers and consumers of the internetwork traffic, while backbones, regionals, and MANs are transit networks. (Most of the networks in an internetwork are stub networks.) Each network consists of nodes (hosts, routers) and links. Two networks are *neighbors* when there is one or more links between nodes in the two networks (see Figure 1).

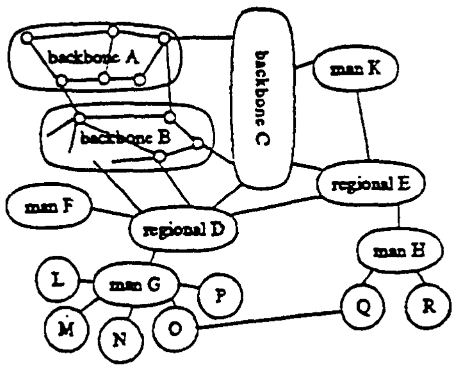


Figure 1: A portion of an internetwork. (Circles represent stub networks.)

An internetwork is organized into *domains*[2]. A domain is a set of networks (possibly consisting of only one network) administered by the same agency. Within each domain, an *intra-domain routing protocol* is executed that provides routes between source and destination nodes in the domain. This protocol can be any of the typical ones, i.e., next-hop or source routes computed using distance-vector or link-state algorithms.

Across all domains, an *inter-domain routing protocol* is executed that provides routes between source and destination nodes in different domains. This protocol must satisfy various constraints:

(1) It must satisfy *policy constraints*, which are administrative restrictions on the inter-domain traffic [8, 12, 9, 5]. Policy constraints are of two types: *transit policies* and *source policies*. The transit policies of a domain A specify how other domains can use the resources of A (e.g. \$0.01 per packet, no traffic from domain B). The source policies of a domain A specify

[1] For example, NSFNET, MILNET are backbones and Suranet, CerfNet are regionals.
[2] also referred to as *routing domains*

constraints on traffic originating from A (e.g. domains to avoid/prefer, acceptable connection cost). Transit policies of a domain are public (i.e. available to other domains), whereas source policies are usually private.

(2) An inter-domain routing protocol must also satisfy *type-of-service* (ToS) constraints of applications (e.g. low delay, high throughput, high reliability, minimum monetary cost). To do this, it must keep track of the types of services offered by each domain [5].

(3) Inter-domain routing protocols must scale up to very large internetworks, i.e. with a very large number of domains. Practically this means that processing, memory and communication requirements should be much less than linear in the number of domains.

(4) Inter-domain routing protocols must automatically adapt to link cost changes, node/link failures and repairs including failures that partition domains [15]. It must also handle non-hierarchical domain interconnections at any level [9] (e.g. we do not want to hand-configure special routes as "back-doors").

A *simple* (or straightforward) approach to inter-domain routing is domain-level source routing with link-state approach [8, 5]. In this approach, each router[3] maintains a *domain-level view* of the internetwork, i.e., a graph with a vertex for every domain and an edge between every two neighbor domains. Policy and ToS information is attached to the vertices and the edges of the view.

When a source node needs to reach a destination node, it (or a router[4] in the source's domain) first examines this view and determines a *domain-level source route* satisfying ToS and policy constraints, i.e., a sequence of domain ids starting from the source's domain and ending with the destination's domain. Then, the packets are routed to the destination using this domain-level source route and the intra-domain routing protocols of the domains crossed.

The disadvantage of this simple scheme is that it does not scale up for large internetworks. The storage at each router is proportional to $N_D \times E_D$, where N_D is the number of domains and E_D is the average number of neighbor domains to a domain. The communication cost is proportional to $N_R \times E_R$, where N_R is the number of routers in the internetwork and E_R is the average router neighbors of a router (topology changes are flooded to all routers in the internetwork).

To achieve scaling, several approaches based on aggregating domains into superdomains have

[3] Not all nodes maintain routing tables. A router is a node that maintains a routing table.

[4] referred to as the policy server in [8]

been proposed [13, 16, 6]. This approaches have drawbacks because the aggregation results in loss of detail (discussed in Section 2).

Our protocol

In this paper, we present an inter-domain routing protocol that we have proposed recently[3]. It combines domain-level views with a novel hierarchical scheme. It scales well to large internetworks, and does not suffer from the problems of superdomains.

In our scheme, domain-level views are not maintained by every router but by special nodes called *viewservers*. For each viewserver, there is a subset of domains around it, referred to as the viewserver's *precinct*. The viewserver maintains the domain-level view of its precinct. This solves the scaling problem for storage requirement.

A viewserver can provide domain-level source routes between source and destination nodes in its precinct. Obtaining a domain-level source route between a source and a destination that are not in any single view, involves accumulating the views of a sequence of viewservers. To make this process efficient, viewservers are organized hierarchically in levels, and an associated addressing structure is used. Each node has a set of addresses. Each *address* is a sequence of viewserver ids of decreasing levels, starting at the top level and going towards the node. The idea is that when the views of the viewservers in an address are merged, the merged view contains domain-level routes to the node from the top level viewservers. (Addresses are obtained from name servers in the same way as is currently done in the Internet.)

We handle dynamic topology changes such as node/link failures and repairs, link cost changes, and domain partitions. Gateways[5] detect domain-level topology changes affecting its domain and neighbor domains. For each domain, there is a *reporting gateway* that communicates these changes by flooding to the viewservers in a specified subset of domains; this subset is referred to as its *flood area*. Hence, the number of packets used during flooding is proportional to the size of the flood area. This solves the scaling problem for the communication requirement.

Thus our inter-domain routing protocol consists of two subprotocols: a **view-query protocol** between routers and viewservers for obtaining merged views; and a **view-update protocol** between gateways and viewservers for updating domain-level views.

[5] A node is called a gateway if it has a link to another domain.

Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements) for inter-domain routing protocols. None of these protocols have been evaluated in a way that they can be compared against each other or the simple approach.

In this paper, we present such a model, and use it to compare our viewserver hierarchy to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to construct a path, and the number of valid paths found (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork. We use three internetwork topologies each of size 11,110 domains (roughly the current size of the Internet). Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Organization of the paper

In Section 2, we survey recent approaches to inter-domain routing. In Section 3, we present the view-query protocol for static network conditions, that is, assuming all links and nodes of the network remain operational. In Section 4, we present the view-update protocol to handle topology changes (this section is not needed for the evaluation part). In Section 5, we present our evaluation model and results from its application to the viewserver hierarchy. In Section 6, we conclude and describe how to add fault-tolerance and cacheing schemes to improve performance.

2 Related Work

In this section, we survey recently proposed inter-domain routing protocols that support ToS and Policy routing for large internetworks [14, 16, 13, 10, 6, 20, 2, 19, 18, 7].

Several inter-domain routing protocols (e.g. BGP [14], IDRP [16], NR [10]) are based on *path-vector* approach [17]. Here, for each destination domain a router maintains a set of paths, one through each of its neighbor routers. ToS and policy information is attached to these paths. Each router requires $O(N_D \times N_D \times E_R)$ space. For each destination, a router exchanges its best valid path[6] with its neighbor routers. However, a path-vector algorithm may not find a valid path from a source to the destination even if such a route exists [13][7]. By exchanging k paths to each destination, the probability of detecting a valid path for each source can be increased.

The most common approach to solve the scaling problem is to use *superdomains*[8] (e.g. IDPR [13], IDRP [16], Nimrod [6]). Superdomains extend the idea of *area hierarchy* [11]. Here, domains are grouped hierarchically into superdomains: "close" domains are grouped into level 1 superdomains, "close" level 1 superdomains are grouped into level 2 superdomains, and so on. Each domain A is addressed by concatenating the superdomain ids starting from a top level superdomain and going down towards A. A router maintains a view that contains the domains in the same level 1 superdomain, the level 1 superdomains in the same level 2 superdomain, and so on. Thus a router maintains a smaller view than it would in the absence of hierarchy. Each superdomain has its own ToS and policy constraints derived from that of the subdomains.

There are several major problems with using superdomains. One problem is that if there are domains with different (possibly contradictory) constraints in a superdomain, then there is no good way of deriving the ToS and policy constraints of the superdomain. The usual techniques are to take either the union or the intersection of the constraints of the subdomains [13]. Both techniques have problems[9]. Other problems are described in [6, 2]. Some of the problems can be relaxed by having overlapping superdomains, but this increases the storage requirements drastically.

Nimrod [6] and IDPR [13] use the link-state approach, domain-level source routing, and superdomains (non-overlapping superdomains for Nimrod). IDRP [16] uses path-vector approach and superdomains.

Reference [10] combines the benefits of path-vector approach and link-state approach by having two modes: An NR mode, which is an extension of IDRP and is used for the most common ToS and policy constraints; and a SDR mode, which is like IDPR and is used for less frequent ToS and

[6] A valid path is a path that satisfies the ToS and policy constraints of the domains in the path.

[7] For example, suppose a router u has two paths $P1$ and $P2$ to the destination. Let u have a router neighbor v, which is in another domain. u chooses and informs v of one of the paths, say $P1$. But $P1$ may violate source policies of v's domain, and $P2$ may be a valid path for v.

[8] also referred to as routing domain confederations

[9] For example, if the union is taken, then a subdomain A can be forced to obey constraints of other subdomains; this may eliminate a path through A which is otherwise valid. If the intersection is taken, then a subdomain A can be forced to accept traffic it would otherwise not accept.

policy requests. This study does not address the scalability of the SDR mode.

In [2], we proposed another protocol based on superdomains. It always finds a valid path if one exists. Both union and intersection policy and ToS constraints are maintained for each visible superdomain. If the union policy constraints of superdomains on a path are satisfied, then the path is valid. If the intersection policy constraints of a superdomain are satisfied but the union policy constraints are not, the source uses a query protocol to obtain a more detailed "internal" view of the superdomain, and searches again for a valid path. The protocol uses a link-state view update protocol to handle topology changes, including failures that partition superdomains at any level.

The landmark hierarchy [19, 18] is another approach for solving the scaling problem. Here, each router is a landmark with a radius, and routers which are within a radius away from the landmark maintain a route to it. Landmarks are organized hierarchically, such that the radius of a landmark increases with its level, and the radii of top level landmarks include all routers. Addressing and packet forwarding schemes are introduced. Link-state algorithms can not be used with the landmark hierarchy, and a thorough study of enforcing ToS and policy constraints with this hierarchy has not been done.

The landmark hierarchy may look similar to our viewserver hierarchy, but in fact they are quite opposite. In the landmark hierarchy, nodes within the radius of the landmark maintain a route to the landmark, and the landmark may not have a route to these nodes. In the viewserver hierarchy, viewserver maintains routes (i.e. a view) to the nodes in its precinct.

Route fragments [7] is an addressing scheme. A destination route fragment, called a *route suffix*, is a sequence of domain ids from a backbone to the destination domain. A source route fragment, called a *route prefix*, is the reverse of a route suffix of that domain. There are also *route middles*, which are from transit domains to transit domains. These addresses are static (i.e. they are not updated with topology changes) and stored at the name servers. A source queries a name server and obtains destination route suffixes. It then chooses an appropriate route suffix for the destination and concatenates it with its own route prefix (and uses routes middles if route suffix and route prefix do not intersect). This scheme can not handle topology changes and does not address handling policy and ToS constraints.

3 Viewserver Hierarchy Query Protocol

In this section, we present our scheme for static network conditions, that is, all links and nodes remain operational. The dynamic case is presented in Section 4.

Conventions: Each domain has a unique id. `DomainIds` denotes the set of domain-ids. Each node has an id which is unique in its domain. `NodeIds` denotes the set of node-ids. Thus, a node is *totally* identified by the combination of its domain's id and its node-id. `TotalIds` denotes the set of total node-ids. For a node u, we use *domainid*(u) to denote the domain-id of u's domain. We use *nodeid*(u) and *totalid*(u) to denote the node-id and total-id of u respectively. For a domain A, we use *domainid*(A) to denote the domain-id of A. *NodeNeighbors*(u) denotes the set of node-ids of the neighbors of u. *DomainNeighbors*(A) denotes the set of domain-ids of the domain neighbors of A. We use the term gateway-id (or viewserver-id) to mean the total-id of a gateway node (or a viewserver node).

In our protocol, a node u uses two kinds of sends. The first kind has the form "Send(m) to v", where m is the message being sent and v is the total-id of the destination. Here, nodes u and v are neighbors, and the message is sent over the physical link (u, v). If the link is down, we assume that the packet is dropped.

The second kind of send has the form "Send(m) to v using *dlsr*", where m and v are as above and *dlsr* is a domain-level source route between u and v. Here, the message is sent using the intra-domain routing protocols of the domains in *dlsr* to reach v[10]. We assume that as long as there is a sequence of up links connecting the domains in *dlsr*, the message is delivered to v[11]. If the u and v are in the same domain, *dlsr* equals $\langle\rangle$.

Views and Viewservers

Domain-level views are maintained by special nodes called *viewservers*. Each viewserver has a *precinct*, which is a set of domains around the viewserver, and a *static view*, which is a domain-level view of the precinct and outgoing edges. The static view includes the ToS and policy constraints

[10] Recall that given a domain-level source route to a destination, using the intra-domain routing protocols we can reach the destination.

[11] This involves time-outs, retransmissions, etc. It requires a transport protocol support such as TCP.

of domains in the precinct and of domain-level edges[12]. Formally, a viewserver x maintains the following:

$Precinct_x$. ($\subseteq$ DomainIds). Domain-ids whose view is maintained.

$SView_x$. Static view of x.

$$= \{\langle A,\ policy\&tos(A),\ \{\langle B,\ edge\_policy\&tos(A,B)\rangle : B \in \text{ subset of } DomainNeighbors(A)\}\rangle : A \in Precinct_x\}$$

$SView_x$ can be implemented as adjacency list representation of graphs [1]. The intention of $SView_x$ is to obtain domain-level source routes between nodes in $Precinct_x$. Hence, the choice of domains to include in $Precinct_x$ and the choice of neighbors of domains to include in $SView_x$ is not arbitrary. $Precinct_x$ and $SView_x$ must be connected; that is, between any two domains in $Precinct_x$, there should be a path in $SView_x$ that lies in $Precinct_x$. Note that $SView_x$ can contain edges to domains outside $Precinct_x$. We say that a domain A is *in the view* of a viewserver x, if either A is in the precinct of x or $SView_x$ has an edge from a domain in precinct to A. Note that the precincts and views of different view servers can be overlapping, identical or disjoint.

If there is a viewserver x whose view contains both the source and the destination domains, then x's view can be used to obtain the required domain-level source route to reach the destination. The source needs to reach x to obtain its view. If the source and x are in the same domain, x can be reached using the intra-domain routing protocol. If x is in another domain, then the source needs to have a domain-level source route to it[13]. In this case, we assume that source has a set of fixed domain-level source routes to x.

Viewserver Hierarchy

For scaling reasons, we cannot have one large view. Thus, obtaining a domain-level source route between a source and a destination which are far away, involves accumulating views of a sequence of viewservers. To keep this process efficient, we organize viewservers hierarchically. More precisely, each viewserver is assigned a hierarchy level from $0, 1, \ldots$, with 0 being the top level in the hierarchy. A parent/child relationship between viewservers is defined as follows:

[12] Not all the domain-level edges need to be included. This is because some domains may have many neighbors causing a big storage requirement.

[13] We cannot obtain this domain-level source route from x, i.e. chicken-egg problem.

1. Every level i viewserver, $i > 0$, has a parent viewserver whose level is less than i.
2. If viewserver x is a parent of viewserver y then x's view contains y's domain and y's view contains x's domain[14].
3. The view of a top level viewserver contains the domains of all other top level viewservers. (typically, top level viewservers are placed in backbones).

Note that the second constraint does not mean that all top level viewservers have the same view. In the hierarchy, a parent can have many children and a child can have many parents. We extend the range of the parent-child relationship to ordinary nodes; that is if the *Precinct*$_x$ contains the domain of node u, we say that u is a child of x, and x is a parent of u (note that an ordinary node does not have a child). We assume that there is at least one parent viewserver for each node.

For a node u, an address is defined to be a sequence $\langle x_0, x_1, \ldots, x_t \rangle$ such that x_i for $i < t$ is a viewserver-id, x_0 is a top level viewserver-id, x_t is the total-id of u, and x_i is a parent of x_{i+1}. Note that a node may have many addresses since the parent-child relationship is many-to-many. If a source wants a domain-level source route to a destination, it first queries the name servers[15] to obtain a set of addresses for the destination. Then, it queries viewservers to obtain an accumulated view containing both its domain and the destination's domain.

Querying the name servers can be done the same way it is done currently in the Internet. It requires nodes to have a set of fixed addresses to name servers. This is also sufficient in our case. However, we can improve the performance by having a set of fixed domain-level source routes instead.

View-Query Protocol: Obtaining Domain-Level Source Routes

We now describe how a domain-level source route is obtained (regardless of whether the source and the destination are in a common view or not).

We want a sequence of viewservers whose merged views contains both the source and the destination domains. Addresses provide a way to obtain such a sequence, by first going up in the viewserver hierarchy starting from the source node and then going down in the viewserver hierarchy towards the destination node. More precisely, let $\langle s_0, \ldots, s_t \rangle$ be an address of the source,

[14] Note that x and y do not have to be in each other's precinct.

[15] In fact, name servers are called *domain name servers*. However, domain names and the domains used in this paper are different. We use "name servers" to avoid confusion.

and $\langle d_0, \ldots, d_l \rangle$ be an address of the destination. Then, the sequence $\langle s_{t-1}, \ldots, s_0, d_0, \ldots, d_{l-1} \rangle$ meets our requirements.[16] In fact, going up all the way in the hierarchy to top level viewservers may not be necessary. We can stop going up at a viewserver s_i if there is a viewserver $d_j, j < l$ such that the domain of d_j is in the view of s_i (one special case is where $s_i = d_j$).

The view-query protocol uses two message types:

- (`RequestView`, *s_address*, *d_address*)
  where *s_address* and *d_address* are the addresses for the source and the destination respectively. A `RequestView` message is sent by a source to obtain an accumulated view containing both the source and the destination domains. When a viewserver receives a `RequestView` message, it either sends back its view or forwards this request to another viewserver.
- (`ReplyView`, *s_address*, *d_address*, *accumview*)
  where *s_address* and *d_address* are as above and *accumview* is the accumulated view. A `ReplyView` message is sent by a viewserver to the source or to another viewserver closer to the source. The *accumview* field in a `ReplyView` message equals the union of the views of the viewservers the message has visited.

We now describe the events of a source node (see Figure 2). The source node[17] sends a `RequestView` packet containing a source and a destination address to its parent in the source address (using a fixed domain-level source route). When the source receives a `ReplyView` packet, it chooses a valid path using the *accumview* in the packet. If it does not find a valid path, it can try again using a different source and/or destination address. Note that, the source does not have to throw away the previous accumulated views, but merge all accumulated views into a richer accumulated view. In fact, it is easy to change the protocol so that source can also obtain views of individual viewservers to make the accumulated view even richer.

The events of a viewserver x are specified in Figure 3. Upon receiving a `RequestView` packet, x checks if the destination domain is in its precinct[18]. If it is, x sends back its view in a `ReplyView` packet. If it is not, x forwards the request packet to another viewserver as follows: x checks if the domain of any viewserver in the destination address is in its view or not. If there is such a domain,

[16] This is similiar to matching route fragments[7]. However, in our case the sequence is computed in a distributed fashion (these is needed to handle topology changes).

[17] or the policy server in the source's domain

[18] Even though destination can be in the view of x, its policies and ToS's are not in the view if it is not in the precinct of x.

```
Constants
  $FixedRoutes_u(x)$, for every viewserver-id $x$ such that $x$ is a parent of $u$,
      $= \begin{cases} \{\langle\rangle\} & \text{if } domainid(u) = domainid(x) \\ \{\langle d_1, \ldots, d_n \rangle : d_i \in \texttt{DomainIds}\}. \text{ Set of domain-level routes to } x & \text{otherwise} \end{cases}$
Events
  $RequestView_u(s\_address,\ d\_address)$        {Executed when $u$ wants a valid domain-level source route}
      Let $s\_address$ be $\langle s_0, \ldots, s_{t-1}, s_t \rangle$, and $dlsr \in FixedRoutes_u(s_{t-1})$;
      Send(RequestView, $s\_address$, $d\_address$) to $s_{t-1}$ using $dlsr$

  $Receive_u$(ReplyView, $s\_address$, $d\_address$, $accumview$)
      Choose a valid domain-level source route using $accumview$;
      If a valid route is not found
          Execute $RequestView_u$ again with another source address and/or destination address
```

Figure 2: View-query protocol: Events and state of a source u.

```
Constants
  $Precinct_x$. Precinct of $x$.
  $SView_x$. Static view of $x$.
Events
  $Receive_x$(RequestView, $s\_address$, $d\_address$)
      Let $d\_address$ be $\langle d_0, \ldots, d_t \rangle$;
      if $domainid(d_t) \notin Precinct_x$ then
          $forward_x$(RequestView, $s\_address$, $d\_address$, $\{\}$);
      else $forward_x$(ReplyView, $d\_address$, $s\_address$, $SView_x$);        {addresses are switched}
      endif

  $Receive_x$(ReplyView, $s\_address$, $d\_address$, $view$)
      $forward_x$(ReplyView, $s\_address$, $d\_address$, $view \cup SView_x$)

  where procedure $forward_x(type, s\_address, d\_address, view)$
      Let $s\_address$ be $\langle s_0, \ldots, s_t \rangle$, $d\_address$ be $\langle d_0, \ldots, d_l \rangle$;
      if $\exists i : domainid(d_i)$ in $SView_x$ then
          Let $i = \max\{j : domainid(d_j) \text{ in } SView_x\}$;
          $target := d_i$;
      else $target := s_i$ such that $s_{i+1} = totalid(x)$;
      endif;
      $dlsr :=$ choose a route to $domainid(target)$ from $domainid(x)$ using $SView_x$;
      if $type =$ RequestView then
          Send(RequestView, $s\_address$, $d\_address$) to $target$ using $dlsr$;
      else Send(ReplyView, $s\_address$, $d\_address$, $view$) to $target$ using $dlsr$;
      endif
```

Figure 3: View-query protocol: Events and state of a viewserver x.

x sends the RequestView packet to the last such one. Otherwise x is a viewserver in the source address and sends the packet to its parent in the source address. (Note that if x is a viewserver in the destination address, its child in the destination address is definitely in its view.)

When a viewserver x receives a ReplyView packet, it merges its view to the accumulated view in the packet. Then it sends the ReplyView packet towards the source node same way it would send a RequestView packet towards the destination node (i.e. the role of the source address and the destination address are changed).

Above we have described one possible way of obtaining the accumulated views. There are various other possibilities, for example: (1) restricting the ReplyView packet to take the reverse of the path that the RequestView packet took; (2) having ReplyView packets go all the way up in the viewserver-hierarchy for a richer accumulated view; (3) source polling the viewservers directly instead of viewservers forwarding request/reply messages to each other; (4) not including the non-transit stub domains other than the source and the destination domains in the *accumview*; (5) including some source policy constraints and ToS requirements in the RequestView packet, and having the viewservers filter out some domains.

4 Update Protocol for Dynamic Network Conditions

In this section, we first examine how topology changes such as link/node failures, repairs, and cost changes, map into domain-level topology changes. Second, we describe how domain-level topology changes are detected and communicated to viewservers, i.e. view-update protocol. Third, we modify the view-query protocol appropriately.

Mapping Topology Changes to Domain-Level Topology Changes

Costs are associated with domain-level edges. The cost of the domain-level edge (A, B) equals a vector of values if the link is up; each cost value indicates how expensive it is to cross domain A to reach domain B according to some criteria such as delay, throughput, reliability, etc. The cost equals ∞ if all links from A to B are down[19]. Each cost value of a domain-level edge (A, B) can be derived from the cost values of the intra-domain routes in A and links from A to B [4][20].

[19] Note that if a gateway connecting A to B is down, its links are also considered to be down.

[20] For example, the delay of a domain-level edge (A, B) can be calculated as the maximum/average delay of the routes from any gateway in A to first gateway in B.

Link cost changes and link/node failures and repairs correspond to cost changes, failures and repairs of domain-level edges. Link/node failures can also partition a domain into cells[15]. A *cell* is a maximal subset of nodes of a domain that can reach each other without leaving the domain. With partitioning, some nodes as well as some neighbor domains may not be accessible by all cells. In the same way, link/node repairs may merge cells into bigger cells. We identify a cell with the minimum node-id of the gateways in the cell. [21] In this paper, for uniformity we treat an unpartitioned domain as a domain with one cell; we do not consider cells that do not isolate gateways since such cells do not affect inter-domain routes.

If a domain gets partitioned, its vertex in the domain-level views should be split into as many pieces as there are cells. And when the cells merge, the corresponding vertices should be merged as well.

Since a domain can be partitioned into many cells, domain-level source routes now include cell-ids as well. Hence, the intra-domain routing protocol of a domain should include a route to each reachable neighbor domain cell.[22]

View-Update Protocol: Updating Domain-Level Views

Viewservers do not communicate with each other to maintain their views. Gateways detect and communicate domain-level topology changes to viewservers. Each gateway periodically (and optionally after a change in the intra-domain routing table) inspects its intra-domain routing table and determines the cell it belongs. For each cell, only the gateway whose node-id is the cell-id (i.e. the gateway with the minimum node-id) is responsible for communicating domain-level topology changes. We refer to this gateway as the *reporting gateway*. Reporting gateways compute the domain-level edge costs for each neighbor domain cell, and report them to parent viewservers. They are also responsible for informing the viewservers of the creation and deletion of cells.

The communication between a reporting gateway and viewservers is done by flooding over a set of domains. This set is referred to as the flood area[23]. The topology of a flood area must

[21] Our cells are like the domain components of IDPR[13].

[22] This involves the following changes in the intra-domain routing protocol: (1) Whenever the cell-id of a gateway changes, it reports its new cell-id to adjacent gateways in neighbor domains. When they receive this information, they update their intra-domain routes to include the new cell-id. (2) Usually when a node recovers from a failure, it queries its neighbors in its domain for their intra-domain routes. When a gateway recovers, it should also query adjacent gateways in neighbor domains for their cell-ids.

[23] For efficiency, the flood area can be implemented by a radius and some forwarding limits (e.g. do not flood

be a connected graph. Due to the nature of flooding, a viewserver can receive information out of order for a domain cell. In order to avoid old information replacing new information, each gateway includes successively increasing time stamps in the messages it sends.

Due to node and link failures, communication between a reporting gateway and a viewserver can fail, resulting in the viewserver having out-of-date information. To eliminate such information, a viewserver deletes any information about a domain cell if it is older than a *time-to-die* period. We assume that gateways send messages more often than the time-to-die value (to avoid false removal).

When a viewserver learns of a new domain cell, it adds it to its view. To avoid adding a domain cell which was just deleted[24], when a viewserver receives a delete domain cell request, it only marks the domain cell as deleted (and removes the entry after the time-to-die period).

The view-update protocol uses two types of messages as follows:

- (`UpdateCell`, *domainid*, *cellid*, *timestamp*, *floodarea*, *ncostset*)
  is sent by the reporting gateway to inform the viewservers about current domain-level edge costs of its cell. Here, *domainid*, *cellid*, and *timestamp* indicate the domain, the cell and the time stamp of the reporting gateway, *ncostset* contains a cost for each neighbor domain cell, and *floodarea* is the set of domains that this message is to be sent over.
- (`DeleteCell`, *domainid*, *cellid*, *timestamp*, *floodarea*)
  where the parameters are as in the `UpdateCell` message. It is sent by a reporting gateway when it becomes non-reporting (because its cell expanded to include a gateway with lower id), to inform viewservers to delete the gateway's old cell.

The state maintained by a gateway g is listed in Figure 4. Note that $LocalViewservers_g$ and $LocalGateways_g$ can be empty. $IntraDomainRT_g$ contains a route (next-hop or source) for every reachable node of the domain and for every reachable neighbor domain cell[25]. We assume that consecutive reads of $Clock_g$ returns increasing values.

The state maintained by a viewserver x is listed in Figure 5. $DView_x$ is the dynamic part of x's view. For each domain cell[26] known to x, $DView_x$ stores a *timestamp* field which equals the

beyond backbones) instead of a set.

[24] If the domain cell was removed, the timestamp for that domain cell is also lost.

[25] $IntraDomainRT_g$ is a view in case of a link-state routing protocol or a distance table in case of a distance-vector routing protocol.

[26] We use $A{:}g$ to denote the cell g of domain A.

Constants:

$LocalViewservers_g$. ($\subseteq$ `TotalIds`). Set of viewservers in g's domain.

$LocalGateways_g$. ($\subseteq$ `TotalIds`). Set of gateways in g's domain excluding g.

$AdjForeignGateways_g$. ($\subseteq$ `TotalIds`). Set of adjacent gateways in other domains.

$FloodArea_g$. ($\subseteq$ `DomainIds`). The flood area of the domain (includes domain of g).

Variables:

$IntraDomainRT_g$. Intra-domain routing table of g. Initially contains no entries.

$CellId_g$: `NodeIds`. The id of g's cell. Initially $= \infty$

$Clock_g$: Integer. Clock of g.

Figure 4: State of a gateway g.

Constants:

$Precinct_x$. Precinct of x.

$SView_x$. Static view of x.

$TimeToDie_x$: Integer. Time-to-die value.

Variables:

$DView_x$. Dynamic view of x.

$= \{ \langle A{:}g,\ timestamp,\ expirytime,\ deleted,$
$\quad \{ \langle B{:}h,\ cost \rangle : B \in DomainNeighbors(A) \wedge h \in \texttt{NodeIds} \cup \{*\} \} \rangle :$
$\quad A \in Precinct_x \wedge g \in \texttt{NodeIds} \}$

$Clock_x$: Integer. Clock of x.

Figure 5: State of a viewserver x.

largest timestamp received for this domain cell, an *expirytime* field which equals the end of the time-to-die period for this domain cell, a *deleted* field which marks whether or not the domain cell is deleted, and a cost set which indicates a cost for every neighbor domain cell whose domain is in $SView_x$. The cell-id of a neighbor domain equals $*$ if no cell of the neighbor domain is reachable.

The events of gateway g and a viewserver x are specified in Appendix A.

Changes to View-Query Protocol

We now enumerate the changes needed to adapt the view-query protocol to the dynamic case (the formal specification is omitted for space reasons).

Due to link and node failures, `RequestView` and `ReplyView` packets can get lost. Hence, the source may never receive a ReplyView packet after it initiates a request. Thus, the source should try again after a time-out period.

When a viewserver receives a RequestView message, in the static case it replies with its view if the destination domain is in its precinct. Now, because domain-level edges can fail, it must also check its dynamic view and reply with its views only if its dynamic view contains a path to the destination. Similarly during forwarding of RequestView and ReplyView packets, a viewserver, while checking whether a domain is in its view, should also check if its dynamic view contains a path to it.

Finally, when a viewserver sends a message to a node whose domain is partitioned, it should send a copy of the message to each cell of the domain. This is because a viewserver does not know which cell contains the node.

5 Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements) for inter-domain routing protocols.

In this section, we first present such a model, and then use the model to evaluate our viewserver hierarchy and compare it to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to construct a path, and the number of paths found out of the total number of valid paths.

Even though the model described here can be applied to other inter-domain routing protocols, we have not done so, and hence have not compared them against our viewserver hierarchy. This is because of lack of time, and because precise definitions of the hierarchies in these protocols is not available. For example, to do a fair evaluation of IDPR[13], we need precise guidelines for how to group domains into super-domains, and how to choose between the union and intersection methods when defining policy/ToS constraints of super-domains. In fact, these protocols have not been evaluated in a way that we can compare them to the viewserver hierarchy. To the best of our knowledge, this paper is the first to evaluate a hierarchical inter-domain routing protocol against explicitly stated policy constraints.

5.1 Evaluation Model

We first describe our method of generating topologies and policy/ToS constraints. We then describe the evaluation measures.

Generating Internetwork Topologies

For our purposes, an internetwork *topology* is a directed graph where the nodes correspond to domains and the edges correspond to domain-level connections. However, an arbitrary graph will not do. The topology should have the characteristics of a real internetwork, like the Internet. That is, it should have backbones, regionals, MANS, LANS, etc.; these should be connected hierarchically (e.g. regionals to backbones), but "non-hierarchical" connections (e.g. "back-doors") should also be present.

For brevity, we refer to backbones as class 0 domains, regionals as class 1 domains, metropolitan-area domains and providers as class 2 domains, and campus and local-area domains as class 3 domains. A (strictly) hierarchical interconnection of domains means that class 0 domains are connected to each other, and for $i > 0$, class i domains are connected to class $i-1$ domains. As mentioned above, we also want some "non-hierarchical" connections, i.e., domain-level edges between domains irrespective of their classes (e.g. from a campus domain to another campus domain or to a backbone domain).

In reality, domains span geographical regions and domain-level edges are usually between domains that are geographically close (e.g. University of Maryland campus domain is connected to SURANET regional domain which is in the east cost). A class i domain usually spans a larger geographical region than a class $i+1$ domain. To generate such interconnections, we associate a "region" attribute to each domain. The intention is that two domains with the same region are geographically close.

The *region* of a class i domain has the form $x_0.x_1.\cdots.x_i$, where the x_j's are integers. For example, the region of a class 3 domain can be `1.2.3.4`. For brevity, we refer to the region of a class i domain as a class i region.

Note that regions have their own hierarchy. Class 0 regions are the top level regions. We say that a class i region $\texttt{r}_0.\texttt{r}_1.\cdots.\texttt{r}_i$ is *contained* in the class $i-1$ region $\texttt{r}_0.\texttt{r}_1.\cdots.\texttt{r}_{i-1}$ (where $i > 0$). Containment is transitive. Thus region `1.2.3.4` is contained in regions `1.2.3`, `1.2` and `1`.

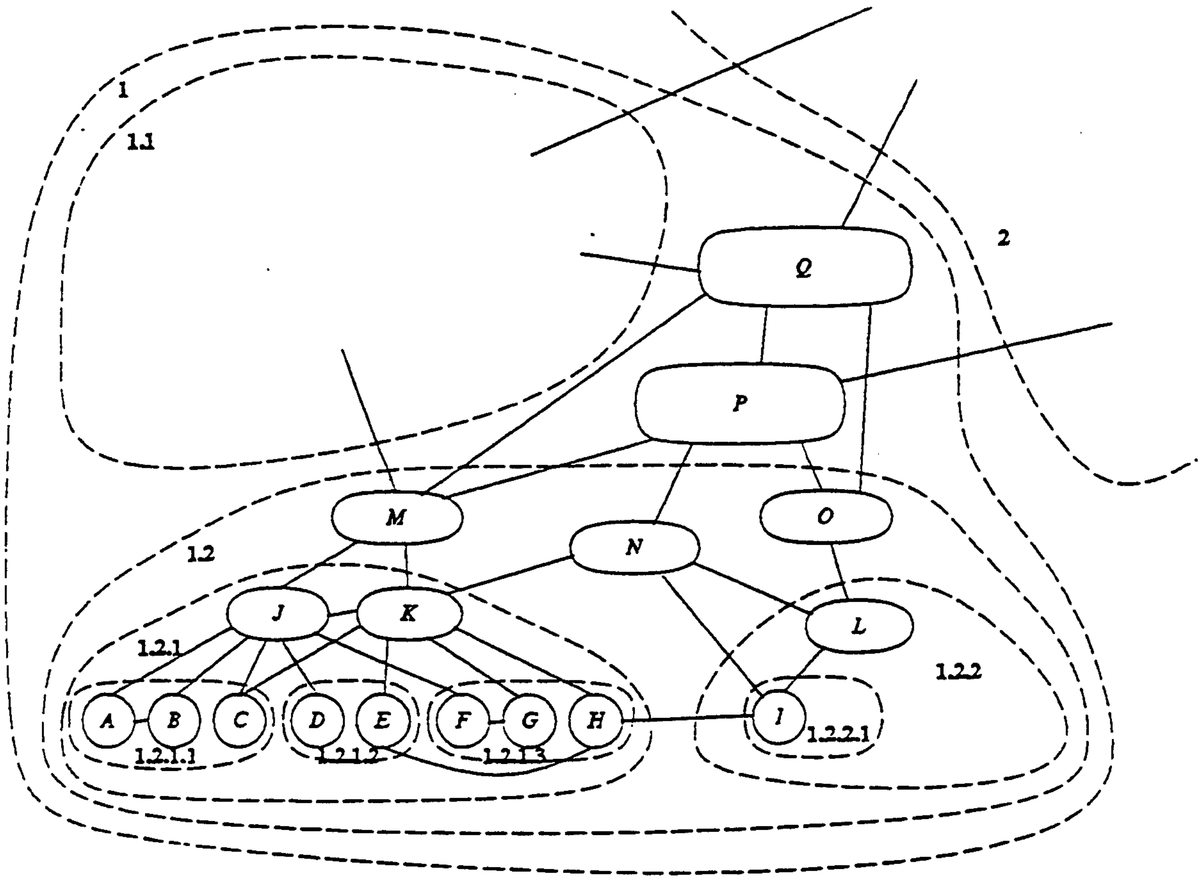


Figure 6: Regions

Given any pair of domains, we classify them as local, remote or far, based on their regions. Let X be a class i domain and Y a class j domain, and (without loss of generality) let $i \leq j$. X and Y are *local* if they are in the same class i region. For example in Figure 6, A is local to B, C, J, K, M, N, O, P, and Q. X and Y are *remote* if they are not in the same class i region but they are in the same class $i-1$ region, or if $i = 0$. For example in Figure 6, some of the domains A is remote to are D, E, F, and L. X and Y are *far* if they are not local or remote. For example in Figure 6, A is far to I.

We refer to a domain-level edge as *local* (*remote*, or *far*) if the two domains it connects are local (remote, or far).

We use the following procedure to generate internetwork topologies:

- We first specify the number of domain classes, and the number of domains in each class.
- We next specify the regions. Note that the number of region classes equals the number of domain classes. We specify the number of class 0 regions. For each class $i > 0$, we specify a *branching factor*, which creates that many class i regions in each class $i - 1$ region. (That is, if there are two class 0 regions and the class 1 branching factor equals three, then there are six class 1 regions.)
- For each class i, we randomly map the class i domains into the class i regions. Note that several domains can be mapped to the same region, and some regions may have no domain mapped into them.
- For every class i and every class j, $j \geq i$, we specify the number of local, remote and far edges to be introduced between class i domains and class j domains. The end points of the edges are chosen randomly (within the specified constraints).

  We ensure that the internetwork topology is connected by ensuring that the subgraph of class 0 domains is connected, and each class i domain, for $i > 0$, is connected to a local class $i - 1$ domain.

Choosing Policy/ToS Constraints

We chose a simple scheme to model Policy/ToS constraints. Each domain is assigned a color: *green* or *red*. For each domain class, we specify the percentage of green domains in that class, and then randomly choose a color for each domain in that class.

A *valid route* from a source to a destination is one that does not visit any red intermediate domains; the source and destination are allowed to be red. Notice that this models transit policy/ToS constraints. We are working on extending this model to source policy/ToS constraints.

Computing Evaluation Measures

The evaluation measures of most interest for an inter-domain routing protocol are its memory and time requirements, and the number of valid paths it finds (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork (e.g. could it find the shortest valid path in the internetwork).

The only analysis method we have at present is to numerically compute the evaluation measures for a variety of source-destination pairs. Because we use internetwork topologies of large sizes, it is not feasible to compute for all possible source-destination pairs. We randomly choose a set of source-destination pairs that satisfy the following conditions: (1) the source and destination domains are different, and (2) there exists a valid path from the source domain to the destination domain in the internetwork topology. (Note that the simple scheme would always find such a path.)

For a source-destination pair, we refer to the length of the shortest valid path in the internetwork topology as the *shortest-path length*. Since the number of paths between a source-destination pair is potentially very large (factorial in the number of domains), and we are not interested in the paths that are too long, we only count the number of paths whose lengths are not more than the shortest-path-length plus 2.

The evaluation measures described above are protocol independent. However, there are also important evaluation measures that are protocol dependent (e.g. number of levels traversed in some particular hierarchy). Because of this we postpone the precise definitions of the evaluation measures to the next subsection (their definition is dependent of viewserver hierarchy).

5.2 Application to Viewserver Protocol

We have used the above model to evaluate our viewserver protocol for several different viewserver hierarchies and query methods. We first describe the different viewserver schemes evaluated. Please refer to Figure 6 in the following discussion.

The first viewserver scheme is referred to as **base**. It has exactly one viewserver in each domain. Each viewserver is identified by its domain-id. The domains in a viewserver's precinct consist of its domain and the neighboring domains. The edges in the viewserver's view consist of the edges between the domains in the precinct, and edges outgoing from domains in the precinct to domains not in the precinct. For example, the precinct of viewserver A (i.e. the viewserver in domain A) consists of domains A, B, J; the edges in the view of viewserver A consists of domain-level edges $(A,B), (A,J), (B,J), (J,M), (J,K), (J,F)$, and (J,D).

As for the viewserver hierarchy, a viewserver's level is defined to be the class of its domain. That is, a viewserver in a class i domain is a level i viewserver. For each level i viewserver, $i > 0$, its parent viewserver is chosen randomly from the level $i-1$ viewservers in the parent region such that there is a domain-level edge between the viewserver's domain and the parent viewserver's domain. For example, for viewserver C, we can pick viewserver J or K; suppose we pick J. For viewserver J, we have no choice but to pick M (N and O are not connected to J). For M, we pick P (out of P and Q).

We use only one address for each domain. The viewserver-address of a stub domain is concatenation of four viewserver (i.e. domain) ids. Thus, the address of A is $P.M.J.A$. Similarly, the address of H is $P.M.K.H$. To obtain a route between A and H, it suffices to obtain views of viewservers A, J, K, H.

The second viewserver scheme is referred to as **base-QT** (where the QT stands for "query upto top"). It is identical to *base* except that during the query protocol all the viewservers in the source and the destination addresses are queried. That is, to obtain a route between A and H, the views of A, J, M, P, K, H are obtained.

The third viewserver scheme is referred to as **locals**. It is identical to *base* except that now a viewserver's precinct also contains domains that have the same region as the viewserver's domain. That is, the precinct of viewserver A has the domains A, B, J, C. Note that in this scheme a viewserver's view is not necessarily connected. For example, if the edge (C, J) is removed, the view of viewserver A is no longer connected. (In Section 3, we said that the view of a viewserver should be connected. Here we have relaxed this condition to simplify testing.)

The fourth viewserver scheme is referred to as **locals-QT**. It is identical to *locals* except that during the query protocol all the viewservers in the source and the destination addresses are queried.

The fifth viewserver scheme is referred to as **vertex-extension**. It is identical to *base* except that viewserver precincts are extended as follows: Let P denote the precinct of a viewserver in the *base* scheme. For each domain X in P, if there is an edge from domain X to domain Y and Y is not in P, domain Y is added to the precinct; among Y's edges, only the ones to domains in P are added to the view. In the example, domains M, K, F, D are added to the precinct of A, but outgoing edges of these domains to other domains are not included (e.g. (F, G) is not included). The advantage of this scheme is that even though it increases the precinct size by a factor which is potentially greater than 2, it increases the number of edges stored in the view by a factor less than 2. (In fact, if the same edge cost and edge policies are used for both directions of domain-level edges, then the only other information that needs to be stored by the viewservers is the policy constraints of the newly added domains.)

The sixth viewserver scheme is referred to as full-QT. It is constructed in the same way as *vertex-extension* except that the *locals* scheme is used instead of *base* scheme to define the P in the construction. In full-QT, during the query protocol all the viewservers in the source and the destination addresses are queried.

In all the above viewserver schemes, we have used the same hierarchy for both domain classes and viewservers. In practice, not all domains need to have a viewserver, and a viewserver hierarchy different from the domain class hierarchy can be deployed. However, there is an advantage of having a viewserver in each domain; that is, source nodes do not require fixed domain-level source routes to their parent viewservers (in the view-query protocol). This reduces the amount of hand configuration required. In fact, the *base* scheme does not require any hand configuration, viewservers can decide their precincts from the intra-domain routing tables, and nodes can use intra-domain routes to reach parent viewservers.

Results for Internetwork 1

The parameters of the first internetwork topology, referred to as Internetwork 1, are shown in Table 1.

Our evaluation measures were computed for a (randomly chosen but fixed) set of 1000 source-destination pairs. For brevity, we use *spl* to refer to the *shortest-path length* (i.e. the length of the shortest valid path in the internetwork topology). The minimum *spl* of these pairs was 2, the maximum *spl* was 13, and the average *spl* was 6.8. Table 2 lists for each viewserver scheme (1) the minimum, average and maximum precinct sizes, (2) the minimum, average and maximum merged view sizes, and (3) the minimum, average and maximum number of viewservers queried.

The precinct size indicates the memory requirement at a viewserver. More precisely, the memory requirement at a viewserver is $O(\text{precinct size} \times d)$ where d is the average number of neighbor domains of a domain, except for the *vertex-extension* and *full-QT* schemes. In these schemes, the memory requirement is increased by a factor less than two. Hence the *vertex-extension* scheme has the same order of viewserver memory requirement as the *base* scheme and the *full-QT* scheme has

[27] Branching factor is 4 for all region classes.

Class i	No. of Domains	No. of Regions[27]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.80	0	8	6	0
1	100	16	0.75	0	190	20	0
				1	26	5	0
2	1000	64	0.70	0	100	0	0
				1	1060	40	0
				2	200	40	0
3	10000	256	0.20	0	100	0	0
				1	100	0	0
				2	10100	50	0
				3	50	50	50

Table 1: Parameters of Internetwork 1.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 68	7 / 71.03 / 101	3 / 7.51 / 8
base-QT	2 / 3.2 / 68	30 / 76.01 / 101	8 / 8.00 / 8
locals	2 / 52.0 / 103	3 / 95.40 / 143	2 / 7.42 / 8
locals-QT	2 / 52.0 / 103	43 / 101.86 / 143	8 / 8.00 / 8
vertex-extension	3 / 19.2 / 796	23 / 362.15 / 486	3 / 7.51 / 8
full-QT	11 / 102.9 / 796	228 / 396.80 / 519	8 / 8.00 / 8

Table 2: Precinct sizes, merged view sizes, and number of viewservers queried for Internetwork 1.

the same order of viewserver memory requirement as the *locals* scheme.

The merged view size indicates the memory requirement at a source; i.e. the memory requirement at a source is O(merged view size $\times$ d) except for the *vertex-extension* and *full-QT* schemes. Note that the source does not need to store information about red and non-transit domains. The numbers in Table 2 take advantage of this.

The number of viewservers queried indicates the communication time required to obtain the merged view at the source. Because the average *spl* is 6.8, the "real-time" communication time required to obtain the merged view at a source is slightly more than one round-trip time between the source and the destination.

As is apparent from Table 2, using a *QT* scheme increases the merged view size and the number of viewservers queried only by about 5%. Using a *locals* scheme increases the merged view size by about 30%. Using the *vertex-extension* scheme increases the merged view size by 5 times (note that the amount of actual memory needed increases only by a factor less than 2). The number of viewservers queried in the *locals* scheme is less than the number of viewservers queried in the *base* scheme. This is because the viewservers in the *locals* scheme have bigger precincts, and a path from the source to the destination can be found using fewer views.

Table 3 shows the average number of *spl*, *spl* + 1, *spl* + 2 length paths found for a source-destination pair by the simple approach and by the viewserver schemes. All the viewserver schemes are very close to the simple approach. The *vertex-extension* and *full-QT* schemes are especially close (they found 98% of all paths). Table 3 also shows the number of pairs for which the viewserver schemes did not find a path (ranging from 1.4% to 5.9% of the source-destination pairs), and the number of pairs for which the viewserver schemes found longer paths. For these pairs, more viewserver addresses need to be tried. Note that the *locals* and *vertex-extension* schemes decrease the number of these pairs substantially (adding *QT* yields further improvement). Our policy constraints are source and destination domain independent. Hence, even a class 2 domain, if it is red, can not carry traffic to a class 3 domain to which it is connected. We believe that these figures would improve with policies that are dependent on source and destination domains.

As is apparent from Table 3 and Table 2, the *locals* scheme does not find many more extra paths than the *base* scheme even though it has larger precinct and merged view sizes. Hence it is not recommended. The *vertex-extension* scheme is the best, but even *base* is adequate since it finds many paths.

We have repeated the above evaluations for two other internetworks and obtained similar conclusions. The results are in Appendix B.

6 Concluding Remarks

We presented hierarchical inter-domain routing protocol that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and

Scheme	Number of paths found			No. of pairs	No. of pairs
	spl	*spl* + 1	*spl* + 2	with no path	with longer paths
simple	2.51	18.48	131.01	N/A	N/A
base	2.41	15.84	99.42	59	3 by 1.33 hops
base-QT	2.41	15.86	100.16	54	3 by 1.33 hops
locals	2.41	16.17	103.54	29	3 by 1 hop
locals-QT	2.41	16.29	105.02	20	3 by 1 hop
vertex-extension	2.51	18.38	128.19	22	0 by 0 hops
full-QT	2.50	18.40	128.90	14	0 by 0 hops

Table 3: Number of paths found for Internetwork 1.

(3) scales well to large number of domains.

Our protocol uses partial domain-level views to achieve scaling in space requirement. It floods domain-level topological changes over a flood area to achieve scaling in communication requirement.

It does not abstract domains into superdomains; hence it does not lose any domain-level detail in ToS and policy information. It merges a sequence of partial views to obtain domain-level source routes between nodes which are far away. The number of views that need to be merged is bounded by twice the number of levels in the hierarchy.

To evaluate and compare inter-domain routing protocols against each other and against simple approach, we presented a model in which one can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures. We applied this model to evaluate our viewserver hierarchy and compared it to the simple approach. Our results indicate that viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two order of magnitude.

Our protocol recovers from fail-stop failures of viewservers and gateways. When a viewserver fails, an address which includes the viewserver's id becomes useless. This deficiency can be overcome by replicating each viewserver at different nodes of the domain (in this case a viewserver fails only if all nodes implementing it fail). This replication scheme requires viewserver ids to be independent of node ids, which can be easily accomplished[28].

[28] For example, if node-ids of nodes implementing a viewserver share a prefix, this prefix can be used as the

The only drawback of our protocol is that to obtain a domain-level source route, views are merged at (or prior to) the connection (or flow) setup, thereby increasing the setup time. This drawback is not unique to our scheme [8, 13, 6, 10].

There are several ways to reduce the setup overhead. First, domain-level source routes to frequently used destinations can be cached. The cacheing period would depend on the ToS requirement of the applications and the frequency of domain-level topology changes. For example, the period can be long for electronic mail since it does not require shortest paths.

Second, views of frequently queried viewservers can be replicated at "mirror" viewservers in the source domain. A viewserver would periodically update the views of its mirror viewservers.

Third, connection setup also involves traversing the name server hierarchy (to obtain destination addresses from its names). By integrating the name server hierarchy with the viewserver hierarchy, we may be able to do both operations simultaneously. This requires further investigation.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ulman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] C. Alaettinoğlu and A. U. Shankar. Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Poicy Resolution. In Proc. *IEEE International Conference on Networking Protocols '93*, San Fransisco, California, October 1993.

[3] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A Scalable and Adaptive Inter-Domain Routing Protocol. Technical Report UMIACS-TR-93-13, CS-TR-3033, Department of Computer Science, University of Maryland, College Park, February 1993.

[4] A. Bar-Noy and M. Gopal. Topology Distribution Cost vs. Efficient Routing in Large Networks. In Proc. *ACM SIGCOMM '90*, pages 242–252, Philadelphia, Pennsylvania, September 1990.

[5] L. Breslau and D. Estrin. Design of Inter-Administrative Domain Routing Protocols. In Proc. *ACM SIGCOMM '90*, pages 231–241, Philadelphia, Pennsylvania, September 1990.

[6] J. N. Chiappa. A New IP Routing and Addressing Architecture. Big-Internet mailing list., 1992. Available by anonymous ftp from `munnari.oz.au:big-internet/list-archive`.

[7] D. Clark. Route Fragments, A Routing Proposal. Big-Internet mailing list., July 1992. Available by anonymous ftp from `munnari.oz.au:big-internet/list-archive`.

[8] D.D. Clark. Policy routing in Internet protocols. Request for Comment RFC-1102, Network Information Center, May 1989.

[9] D. Estrin. Policy requirements for inter Administrative Domain routing. Request for Comment RFC-1125, Network Information Center, November 1989.

[10] D. Estrin, Y. Rekhter, and S. Hotz. Scalable Inter-Domain Routing Architecture. In Proc. *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

viewserver id. Intra-domain routing would forward a packet destined to a viewserver to any operational node with this prefix.

[11] F. Kamoun and L. Kleinrock. Stochastic Performance Evaluation of Hierarchical Routing for Large Networks. *Computer Networks and ISDN Systems*, 1979.

[12] B.M. Leiner. Policy issues in interconnecting networks. Request for Comment RFC-1124, Network Information Center, September 1989.

[13] M. Lepp and M. Steenstrup. An Architecture for Inter-Domain Policy Routing. Internet Draft. Available from the authors., June 1992.

[14] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). Request for Comment RFC-1105, Network Information Center, June 1989.

[15] R. Perlman. Hierarchical Networks and Subnetwork Partition Problem. *Computer Networks and ISDN Systems*, 9:297–303, 1985.

[16] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). Available from the author., 1992. T.J. Watson Research Center, IBM Corp.

[17] K. G. Shin and M. Chen. Performance Analysis of Distributed Routing Strategies Free or Ping-Pong-Type Looping. *IEEE Transactions on Computers*, 1987.

[18] P. F. Tsuchiya. The Landmark Hierarchy: Description and Analysis, The Landmark Routing: Architecture Algorithms and Issues. Technical Report MTR-87W00152, MTR-87W00174, The MITRE Corporation, McLean, Virginia, 1987.

[19] P. F. Tsuchiya. The Landmark Hierarchy:A New Hierarchy For Routing In Very Large Networks. In Proc. *ACM SIGCOMM '88*, August 1988.

[20] P. F. Tsuchiya. Efficient and Robust Policy Routing Using Multiple Hierarchical Addresses. In Proc. *ACM SIGCOMM '91*, pages 53–65, Zurich, Switzerland, September 1991.

A View-Update Protocol Event Specifications

The events of gateway g are specified in Figure 7. When a gateway g recovers, $CellId_g$ is set to $nodeid(g)$. Thus, when g next executes $Update_g$, it sends either an `UpdateCell` or a `DeleteCell` message to viewservers, depending on whether it is no longer the minimum id gateway in its cell[29].

The events of a viewserver x are specified in Figure 5. Note that when x adds an entry to $DView_x$ (upon receiving a `UpdateCell` message), it selectively chooses subset of neighbors from the cost set in the packet to include only the neighbor domains which are in $SView_x$. When a viewserver x recovers, $DView_x$ is set to $\{\}$. Its view becomes up-to-date as it receives new information from reporting gateways (and remove false information with the time-to-die period).

[29] Sending a `DeleteCell` message is essential. Because prior to the failure, g may have been the smallest id gateway in its cell. Hence, some viewserver's may still contain an entry for its old domain cell.

```
$Update_g$        {Executed periodically and also optionally upon a change in $IntraDomainRT_g$}
    {Determines the id of $g$'s cell and initiates UpdateCell and DeleteCell messages if needed.}
    $OldCellId = CellId_g$;
    $CellId_g$ := compute cell id using $LocalGateways_g$ and $IntraDomainRT_g$;
    if $nodeid(g) = CellId_g$ then
        $ncostset$ := compute costs for each neighbor domain cell using $IntraDomainRT_g$;
        $flood_g$((UpdateCell, $domainid(g)$, $CellId_g$, $Clock_g$, $FloodArea_g$, $ncostset$));
    endif
    if $nodeid(g) = OldCellId \neq CellId_g$ then
        $flood_g$((DeleteCell, $domainid(g)$, $nodeid(g)$, $Clock_g$, $FloodArea_g$));
    endif
$Receive_g(packet)$        {either an UpdateCell or a DeleteCell packet}
    $flood_g(packet)$

where procedure $flood_g(packet)$
    if $domainid(g) \in packet.floodarea$ then
        {remove domain of $g$ from the flood area to avoid infinite exchange of the same message.}
        $packet.floodarea := packet.floodarea - \{domainid(g)\}$;
        for all $h \in LocalGateways_g \cup LocalViewservers_g$ do
            Send($packet$) to $h$ using $\langle\rangle$;
        endif
        for all $h \in AdjForeignGateways_g \wedge domainid(h) \in packet.floodarea$ do
            Send($packet$) to $h$;
```

Gateway Failure Model: A gateway can undergo failures and recoveries at anytime. We assume failures are fail-stop (i.e. a failed gateway does not send erroneous messages). When a gateway g recovers, $CellId_g$ is set to $nodeid(g)$.

Figure 7: View-update protocol: Events of a gateway g.

B Results for Other Internetworks

Results for Internetwork 2

The parameters of the second internetwork topology, referred to as Internetwork 2, are the same as the parameters of Internetwork 1 (a different seed is used for the random number generation).

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum *spl* of these pairs was 2, the maximum *spl* was 13, and the average *spl* was 7.2.

Table 4 and Table 5 shows the results. Similar conclusions to Internetwork 1 hold for Internetwork 2. In Table 5, the reason that *local* and *QT* schemes have more pairs with longer paths than the *base* scheme is that these schemes found some paths (which are not shortest) for some pairs for which the *base* scheme did not find any path.

```
$Receive_x$(UpdateCell, $did$, $cid$, $ts$, $FloodArea$, $ncset$)
    if $did \in Precinct_x$ then
        if $\exists(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset) \in DView_x\ \wedge$
          $ts > timestamp$ then          {received is more recent; delete the old one}
            delete $(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset)$ from $DView_x$;
        endif
        if $\neg\exists(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset) \in DView_x$ then
            Choose $ncostset$ from $ncset$ using $SView_x$;
            insert $(did{:}cid,\ ts,\ Clock_x + TimeToDie_x,$ false, $ncostset)$ to $DView_x$;
        endif
    endif
$Receive_x$(DeleteCell, $did$, $cid$, $ts$, $floodarea$)
    if $did \in Precinct_x$ then
        if $\exists(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset) \in DView_x\ \wedge$
          $ts > timestamp$ then          {received is more recent; delete the old one}
            delete $(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset)$ from $DView_x$;
        endif
        if $\neg\exists(did{:}cid,\ timestamp,\ expirytime,\ deleted,\ ncostset) \in DView_x$ then
            insert $(did{:}cid,\ ts,\ Clock_x + TimeToDie_x,$ true, $\{\})$ to $DView_x$;
        endif
    endif
$Delete_x$          {Executed periodically to delete entries older than the time-to-die period}
    for all $(A{:}g,\ tstamp,\ expirytime,\ deleted,\ ncset) \in DView_x\ \wedge\ expirytime < Clock_x$ do
        delete $(A{:}g,\ tstamp,\ expirytime,\ deleted,\ ncset)$ from $DView_x$;
```

Viewserver Failure Model: A viewserver can undergo failures and recoveries at anytime. We assume failures are fail-stop. When a viewserver x recovers, $DView_x$ is set to $\{\}$.

Figure 8: View update events of a viewserver x.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 76	4 / 66.62 / 96	3 / 7.55 / 8
base-QT	2 / 3.2 / 76	29 / 72.76 / 96	8 / 8.00 / 8
locals	3 / 69.8 / 149	4 / 101.32 / 148	2 / 7.36 / 8
locals-QT	3 / 69.8 / 149	35 / 110.32 / 152	8 / 8.00 / 8
vertex-extension	3 / 19.47 / 817	15 / 339.60 / 469	3 / 7.55 / 8
full-QT	11 / 135.2 / 817	186 / 402.51 / 503	8 / 8.00 / 8

Table 4: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 2.

Results for Internetwork 3

The parameters of the third internetwork topology, referred to as Internetwork 3, are shown in Table 6. Internetwork 3 is more connected, more class 0, 1 and 2 domains are green, and more

Scheme	Number of paths found			No. of pairs	No. of pairs
	spl	$spl + 1$	$spl + 2$	with no path	with longer paths
simple	2.21	13.22	74.30	N/A	N/A
base	1.98	8.20	34.40	123	13 by 1.08 hops
base-QT	1.98	8.36	35.62	110	15 by 1.13 hops
locals	2.08	9.18	40.50	97	23 by 1.39 hops
locals-QT	2.08	9.38	42.08	67	23 by 1.30 hops
vertex-extension	2.18	12.57	64.98	19	6 by 1 hop
full-QT	2.19	12.85	67.37	4	4 by 1 hop

Table 5: Number of paths found for Internetwork 2.

class 3 domains are red. Hence, we expect more valid paths between source and destination pairs.

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum *spl* of these pairs was 2, the maximum *spl* was 10, and the average *spl* was 5.93.

Class i	No. of Domains	No. of Regions[30]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.85	0	8	7	0
1	100	16	0.80	0	190	20	0
				1	50	20	0
2	1000	64	0.75	0	500	50	0
				1	1200	100	0
				2	200	40	0
3	10000	256	0.10	0	300	50	0
				1	250	100	0
				2	10250	150	50
				3	200	150	100

Table 6: Parameters of Internetwork 3.

[30] Branching factor is 4 for all domain classes.

Table 7 and Table 8 shows the results. Similar conclusions to Internetwork 1 and 2 hold for Internetwork 3.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.5 / 171	5 / 134.41 / 206	3 / 7.26 / 8
base-QT	2 / 3.5 / 171	55 / 154.51 / 206	8 / 8.00 / 8
locals	3 / 70.17 / 171	4 / 164.16 / 257	2 / 7.09 / 8
locals-QT	3 / 70.17 / 171	57 / 191.06 / 258	8 / 8.00 / 8
vertex-extension	5 / 34.17 / 1986	18 / 601.56 / 695	3 / 7.26 / 8
full-QT	14 / 155.5 / 1986	503 / 655.79 / 743	8 / 8.00 / 8

Table 7: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 3.

	Number of paths found			No. of pairs	No. of pairs
Scheme	*spl*	*spl* + 1	*spl* + 2	with no path	with longer paths
simple	3.34	37.55	368.97	N/A	N/A
base	2.83	24.25	178.08	17	11 by 1.09 hops
base-QT	2.87	25.53	193.41	12	8 by 1.12 hops
locals	2.87	25.62	196.33	21	8 by 1 hop
locals-QT	2.97	27.59	219.63	2	6 by 1 hop
vertex-extension	3.32	35.73	332.54	5	1 by 1 hop
full-QT	3.33	36.47	346.44	0	0 by 0 hops

Table 8: Number of paths found for Internetwork 3.

Figure 9 through Figure 11 show the number of *spl*, *spl* + 1 and *spl* + 2 length paths found by the schemes as a function of *spl* (we only show results for *spl* values for which more than 10 pairs were found). We do not include *base-QT*, *locals* and *locals-QT* schemes since they are very close to *base* scheme. As expected, as *spl* increases, the number of paths for a source-destination pair increases, and the gap between the *simple* scheme and the viewserver schemes increases.

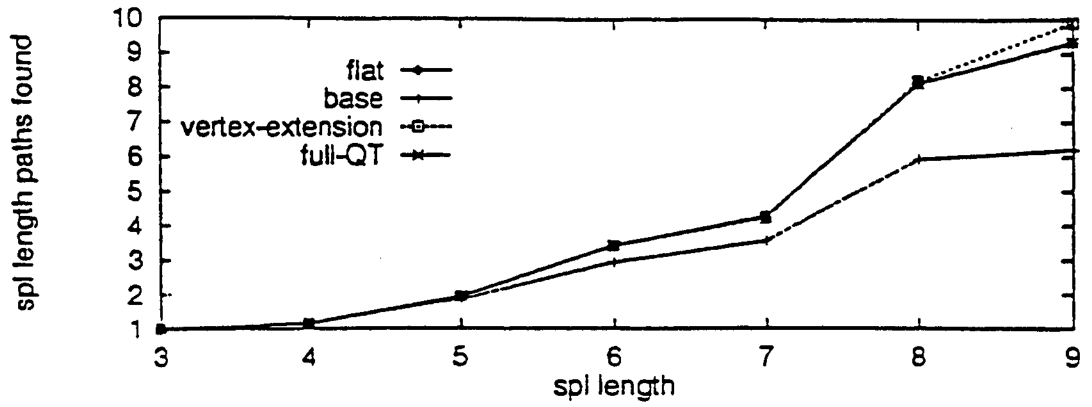


Figure 9: Number of *spl* length paths found for Internetwork 3.

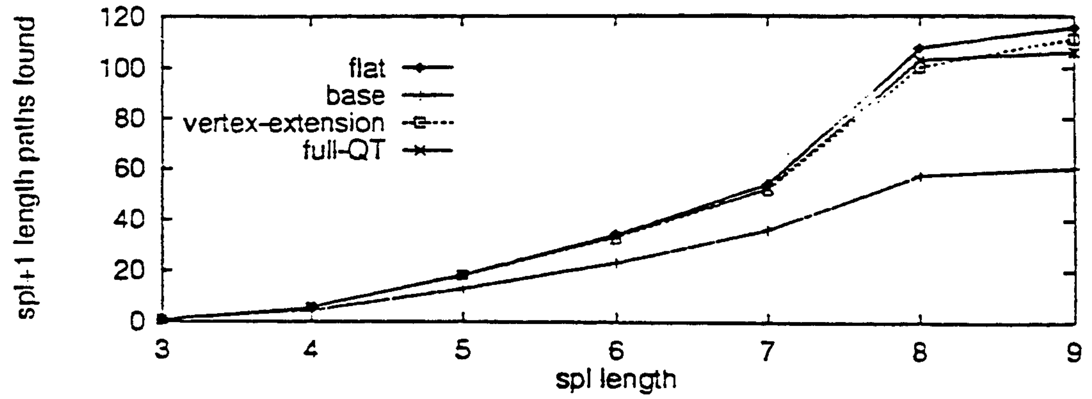


Figure 10: Number of *spl* + 1 length paths found for Internetwork 3.

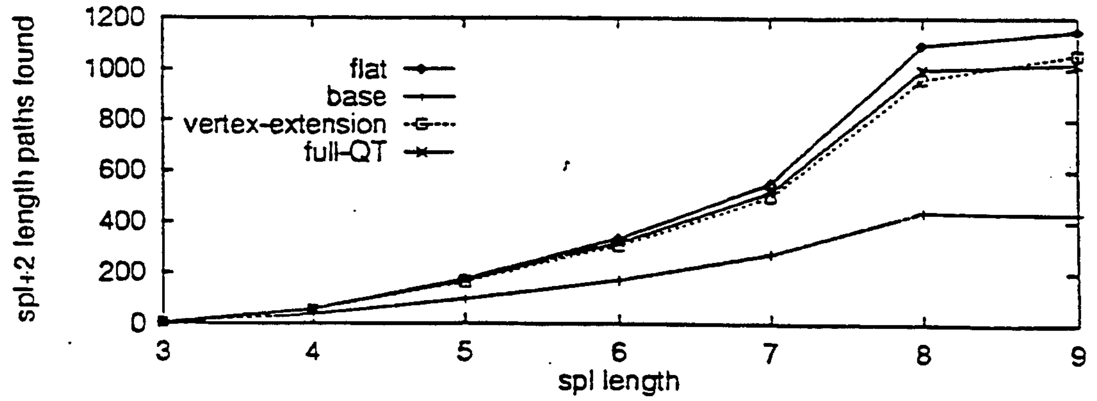


Figure 11: Number of *spl* + 2 length paths found for Internetwork 3.

REPORT DOCUMENTATION PAGE		Form approved OMB No 074-0188
1. AGENCY USE ONLY (leave blank)	2. REPORT DATE 10/15/93	3. REPORT TYPE END DATES COVERED Technical Report

4. TITLE AND SUBTITLE Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation	5. FUNDING NUMNBERS DASG-60-92-C-0055
6. AUTHOR(S) Cengiz Alaettinoglu, and A. Udaya Shankar	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science A.V. Williams Building College Park, MD 20742	8. PERFORMING ORGANIZATION REPORT NUMBER CS-TR-3151 UMIACS-TR-93-98
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) PhillipsLaboratory Director of Contracting 3651 Lowry Avenue SE Kirtland AFB, NM 87117-5777	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12.a. DISTRIBUTION/ AVAILABILITY STATEMENT	12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A simple approach to inter-domain routing is domain-level source routing with link-state approach where each node maintains a domain-level view of the internetwork. This does not scale up to large internetworks. The usual scaling tecnnique of aggregating domains into superdomains loses ToS and policy detail.

We present a new viewserver hierarchy and associated protocols that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and (3) scales well to large nuimber of dominas without losing detail. Domain-level views are maintained by special nodes called viewservers. Each viewserver maintains a domain-level view of surrounding precinct. Viewservers are organized hierarchically. To obtain domain-level source routes, the views of one or more viewservers are merged (upto a maximum of twice the levels in the hierarchy.

We also present a model for evaluating inter-domain routing protocols, and apply this model to compare our viewserver hierarchy against the simple approach . Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

14. SUBJECT TERMS Network Architecture and Design	15. NUMBER OF PAGES 33
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

TEMPORAL ANALYSIS FOR HARD REAL-TIME SCHEDULING*

Manas C. Saksena and Ashok K. Agrawala

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Static time driven scheduling has been advocated for use in Hard Real-Time systems and is particularly appropriate for many embedded systems. The approaches taken for static scheduling often use search techniques and may reduce the search by using heuristics. In this paper we present a technique for analyzing the temporal relations among the tasks, based on non-preemptive schedulability. The relationships can be used effectively to reduce the average complexity of scheduling these tasks. They also serve as a basis for selective preemption policies for scheduling by providing an early test for infeasibility. We present examples and simulation results to confirm the usefulness of temporal analysis as a phase prior to scheduling.

1 Introduction

Many safety critical real-time applications like process control, embedded tactical systems for military applications, air-traffic control, robotics etc. have stringent timing constraints imposed on their computations due to the characteristics of the physical system. A failure to observe the timing constraints can result in intolerable system degradation and in some cases it may have catastrophic consequences.

Scheduling is the primary means of ensuring the satisfaction of timing constraints for such systems[1]. As a result, significant effort has been invested in research on hard real time scheduling [2, 3, 4]. In this paper we discuss a scheduling technique for static scheduling to guarantee timely execution of time critical tasks.

The time driven scheduling model is being used by many experimental systems, including MARS[5], MARUTI[6] and Spring Kernel[7]. The static time driven scheduling technique involves constructing a schedule offline, which may be represented as a Gantt chart[8] or calendar[6] (Figure 1). Tasks are invoked at run-time whenever they are scheduled to execute. Such a scheduling model is particularly appropriate for many embedded systems. Recent effort in this direction has shown the viability of such an approach for practical real-time applications[9].

*This research was supported in part by ONR and DARPA under contract N00014-91-c-0195.

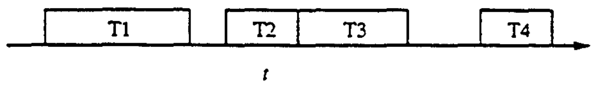


Figure 1: Gantt Chart or Calendar

The intractability of most scheduling problems has led to approaches based on search techniques for scheduling of real-time tasks. The feasibility of a task set is determined through construction of a schedule; failure to construct a schedule denotes infeasibility. Heuristics are often used as a means of controlling the complexity of scheduling. In many cases, heuristics perform well enough to result in an acceptable solution.

There has been little emphasis on the use of analytic techniques to assist in time driven scheduling. Decomposition scheduling[10] based on dominance properties of sequences[11] uses analytic techniques to decompose a set of tasks into a sequence of subsets. Significant reduction in average complexity can be achieved if the set of tasks can be decomposed into a large number of subsets, each having a small number of tasks.

In this paper, we present an analysis technique for time driven scheduling based on the timing requirements of tasks. The analysis results in the establishment of a set of temporal relations between pairs of tasks based on a non-preemptive scheduling model. These relations can be used by scheduling algorithms to reduce the complexity of scheduling in the average case, and as an early test for infeasibility. As a test for infeasibility, it provides a good basis for policies using selective preemption to enhance feasibility. When infeasibility is not detected, the temporal relations may be used by a search algorithm to effectively prune large portions of search space, thereby controlling the cost of scheduling.

2 Time Driven Scheduling

The time driven scheduling approach constructs a calendar for the set of tasks in the system. The tasks may be scheduled preemptively or non-preemptively. The non-preemptive scheduling problem for a uniprocessor is known to be NP-Complete[12]. When the tasks are mutually independent, and can be preempted at any time, it is known that the earliest deadline first policy is optimal[13] and obviates the need for non-preemptive scheduling. However, when tasks synchronize using critical sections, the preemptive scheduling problem is also known to be intractable(NP-Hard)[14].

In general, when the overhead of preemption is negligible, the non-preemptive solutions form a subset of preemptive solutions[8]. However, when tasks may interact with each other, the non-preemptive models are simpler, easier to implement and closer to reality[15]. They are also necessary for certain scheduling domains like I/O scheduling and provide a basis for selective preemption policies.

2.1 Task Model

We consider a set of n tasks $\Gamma = \{\tau_i : i = 1, 2, \ldots, n\}$ to be scheduled for execution on a single processor. Each task τ_i, abbreviated as i, is a 3-tuple $[r_i, c_i, d_i]$ denoting the ready time, computation time and deadline respectively. The time interval $[r_i, d_i]$ is called the timing window w_i of task τ_i, and indicates the time interval during which the task can execute. The computation time of each task is less than the window length $|w_i|$[1]. All tasks are assumed to be independent for simplicity of exposition even though such a requirement is not necessary for the analysis.

In a hard real-time system, processes may be *periodic* or *sporadic* [14]. Such a set of processes may be mapped to our scheduling model by techniques identified in [1, 14, 16] and constructing a schedule for the least common multiple of the periods of the tasks.

2.2 Non-Preemptive Scheduling Model

A non-preemptive schedule is the mapping of each task τ_i in Γ to a start time s_i. The task is then scheduled to run without preemption in the time interval $[s_i, f_i]$, with its finish time being $f_i = s_i + c_i$. A ***feasible*** schedule is a schedule in which the following conditions are satisfied for each task τ_i:

$$r_i \leq s_i \quad (1)$$
$$f_i \leq d_i \quad (2)$$

It is useful to consider a non-preemptive schedule as an ordered sequence of the set of tasks. To get a maximally packed schedule from a sequence $[\tau_1, \tau_2, \ldots, \tau_n]$, we can recursively derive the start time s_i and finish time f_i of the tasks as follows:

$$s_i = max(r_i, f_{i-1}) \quad (3)$$
$$f_i = s_i + c_i \quad (4)$$

with

$$s_1 = r_1$$

The scheduling problem can thus be considered as a search over the permutation space. A permutation (sequence) is feasible if the corresponding schedule is feasible. Notice that for any permutation schedule derived as above, equation 1 is implied by (3) and we only need to verify the deadline constraints for the tasks.

[1] $|w_i| = d_i - r_i$

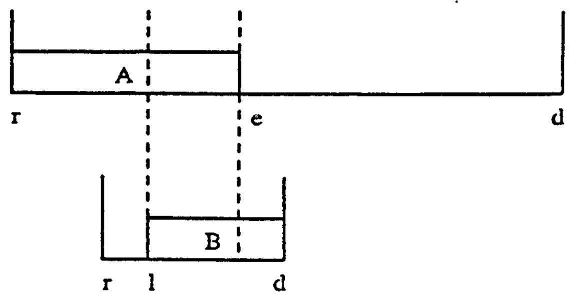


Figure 2: Infeasibility of task A executing before task B

3 Temporal Analysis

Temporal analysis uses pairwise schedulability analysis of tasks to generate a set of relations to eliminate sequences which cannot lead to feasible solutions. In this section we define the temporal relations and show how they may be derived from the timing constraints of tasks.

3.1 Definitions of Temporal Relations

Consider two tasks τ_i and τ_j. We wish to find out what we can say about the relative ordering of these tasks, given their timing constraints. A set of relations are identified below which identify the different possibilities.

Precedence Relation: A precedence relation denoted as $\tau_i \longrightarrow \tau_j$, implies that in any feasible schedule τ_i must execute before τ_j.

Infeasible Relation: An infeasible relation denoted by $\tau_i \oslash \tau_j$ implies that in any feasible schedule, τ_i and τ_j cannot run in a sequential order.

Concurrent Relation: $\tau_i \parallel \tau_j$ if there is no precedence or infeasible relation between them. A concurrent relation indicates that a feasible schedule may exist with any order of the tasks τ_i and τ_j. It does not, however, indicate the existence of a feasible schedule.

For each task τ_i let us define two terms e_i and l_i, denoting the ***earliest finish time*** and the ***latest start time*** as:

$$e_i = r_i + c_i \quad (5)$$
$$l_i = d_i - c_i \quad (6)$$

A preliminary set of relations can be established using the following rules, for every pair of tasks τ_i and τ_j.

$$(e_i \leq l_j) \wedge (l_i < e_j) \Rightarrow \tau_i \longrightarrow \tau_j \quad (7)$$
$$(e_i > l_j) \wedge (l_i \geq e_j) \Rightarrow \tau_j \longrightarrow \tau_i \quad (8)$$
$$(e_i \leq l_j) \wedge (l_i \geq e_j) \Rightarrow \tau_i \parallel \tau_j \quad (9)$$
$$(e_i > l_j) \wedge (l_i < e_j) \Rightarrow \tau_i \oslash \tau_j \quad (10)$$

The basic idea is that if the earliest finish time of a task A is greater than the latest start time of a task B, then a feasible schedule cannot be found in which A is scheduled before B

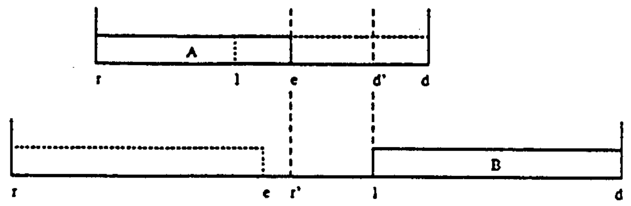


Figure 3: Window Modification (A $\longrightarrow$ B)

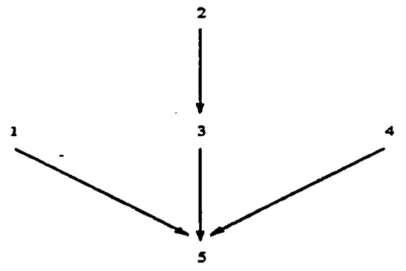


Figure 5: Precedence Graph for example of Figure 4

(Figure 2). Thus, for instance the first part of condition for rule 10 says that τ_i cannot precede τ_j, and the second part says that τ_j cannot precede τ_i, establishing the infeasible relation.

3.2 Window Modification

Consider two tasks τ_a and τ_b, and a precedence relation $\tau_a \longrightarrow \tau_b$ between them. As this indicates that in any feasible schedule τ_a must precede τ_b, we can update the timing windows, as follows (Figure 3):

$$d'_a = min(d_a, l_b) \quad (11)$$
$$r'_b = max(r_b, e_a) \quad (12)$$

The window modification does not alter the scheduling problem in the sense that every feasible sequence with the original timing constraints is a feasible sequence with the modified timing constraints and vice-versa. Further, the schedules for feasible sequences are identical in both cases. A task's window may shrink because of window modification. This may lead to a change in the relation of the modified task with other tasks. The procedure may be applied iteratively till no further changes can be made or an infeasible relation is detected.

3.3 Examples

(a) Consider a set of five tasks as shown in Figure 4. The temporal analysis leads us to the following set of precedence relations, sans the redundant ones:

$$\{\tau_2 \longrightarrow \tau_3, \tau_1 \longrightarrow \tau_5, \tau_3 \longrightarrow \tau_5, \tau_4 \longrightarrow \tau_5\}$$

The set of precedence relations may be represented as a precedence graph (Figure 5) and impose a partial order on the task set. Only sequences which are consistent with this partial order need to be considered for scheduling. For 5 tasks, the total number of permutations is 120(= 5!). The number of total orders consistent with the partial order of Figure 5 is 12, which is a drastic reduction in the number of sequences that need to be considered for scheduling. The modified task set is shown in Figure 4(b), with the modified values in bold.

(b) As another example, consider the set of 4 tasks as shown in Figure 6. The task set in different stages of temporal analysis is shown, with the new temporal relations[2] at each stage. This example shows how successive refinement of temporal relations can lead to detecting infeasibility.

3.4 Complexity of Temporal Analysis

It is easy to see that the initial set of relations can be established in $O(n^2)$ time. Further, each phase of refinement also takes no more than $O(n^2)$. An upper bound for the number of phases is n. Therefore, the worst case complexity of temporal analysis is $O(n^3)$. In practice, however, the cost of temporal analysis can be significantly less since concurrent relations and relations between non-overlapping tasks need not be generated explicitly. Furthermore, the number of phases required to stabilize window modification can be reduced if the release times are modified in the topological sort order of the precedence graph and deadlines are modified in the reverse topological sort order.

In any case, the cost of temporal analysis for static scheduling is not significant when used in conjunction with an exponential time scheduling algorithm. In section 5, we show empirically that the cost of temporal analysis is not a significant factor for static scheduling.

4 Non Preemptive Scheduling using Temporal Analysis

The relations established through temporal analysis serve as a basis for scheduling of the tasks. Temporal analysis may thus be perceived as a pre-processing stage for scheduling. The result of this pre-processing stage is one of the following:

1. The task set was detected to be infeasible, due to the existence of one or more infeasible relations.
2. A set of precedence relations were established generating a precedence graph. The precedence graph imposes a partial order on the set of tasks. It serves as an input to the scheduler which may exploit the partial order generated to prune the search space.

4.1 Detecting Infeasibility

Whenever, an infeasible relation exists between two tasks, it is known that no ordering of the two tasks is feasible. Thus,

[2]Concurrent Relations are not shown.

	τ_1	τ_2	τ_3	τ_4	τ_5
r	0	5	9	0	8
c	7	8	4	8	13
d	29	16	23	30	42

(a)

	τ_1	τ_2	τ_3	τ_4	τ_5
r	0	5	13	0	17
c	7	8	4	8	13
d	29	16	23	29	42

(b)

Figure 4: Window Modification: (a) Original Task Set (b) Task Set after Temporal Analysis

	τ_1	τ_2	τ_3	τ_4
r	40	30	0	0
e	55	55	25	25
c	15	25	25	25
l	45	65	75	75
d	60	90	100	100

$\tau_1 \longrightarrow \tau_2$

(a)

	τ_1	τ_2	τ_3	τ_4
r	40	55	0	0
e	55	80	25	25
c	15	25	25	25
l	45	65	75	75
d	60	90	100	100

$\tau_3 \longrightarrow \tau_2, \tau_4 \longrightarrow \tau_2$

(b)

	τ_1	τ_2	τ_3	τ_4
r	40	30	0	0
e	55	55	25	25
c	15	25	25	25
l	45	65	40	40
d	60	90	65	65

$\tau_3 \longrightarrow \tau_1, \tau_4 \longrightarrow \tau_1$

(c)

	τ_1	τ_2	τ_3	τ_4
r	40	55	0	0
e	55	80	25	25
c	15	25	25	25
l	45	65	20	20
d	60	90	45	45

$\tau_3 \oslash \tau_4$

(d)

Figure 6: Example for Determining Infeasibility with Temporal Analysis

the detection of an infeasible relation at any stage in temporal analysis indicates that the task set is infeasible. Even though only pairwise schedulability analysis is used for establishing relations, successive refinement of relations results in a possible percolation of this effect to other tasks too. This effect is exemplified in the example of Figure 6, where several iterations lead to a infeasible relation. It must be noted that whenever infeasibility is detected, the resulting task set and their relations also provide a good feedback as to what caused it. The feedback information may be used to allocate more resources, change resource allocation or allow for selective preemption as the case may be.

4.2 Search Technique for Scheduling

The intractability of non-preemptive scheduling has led to implicit enumeration techniques based on branch and bound search methods. The search space is the set of all possible permutation sequences. One way of enumerating schedules is to generate an initial schedule and then successively refine it using heuristics to generate "better" schedules, until a feasible schedule is obtained [3, 17, 16].

In this paper, we concentrate on another enumeration method which constructs a schedule in an incremental manner. Variants of this method have been used in [4, 18, 19, 20, 21] The search space is represented as a search tree. The root (level 0) of the tree is an empty schedule. The nodes of the tree represent partial schedules. A node at level k gives a partial schedule with k tasks. The leaves are complete schedules. The successors of an intermediate node are *immediate extensions* of the partial schedule corresponding to that node. From a node at level k, there are at most $n-k$ branches with each branch corresponding to an extension of the partial schedule by appending one more task to the schedule. Search is done in a *branch and bound* manner, wherein parts of the search tree are pruned when it is determined that no feasible schedule can arise from them. For each node being expanded, the following conditions must hold.

1. All immediate extensions of the node must be feasible [4, 18].
2. The remaining computational demand must not exceed the difference between the largest deadline of remaining tasks and current scheduling time [4].

If any condition is violated then no feasible schedule can be generated in the subtree originating from this node. No search is conducted on the subtree rooted at such a node.

4.2.1 Heuristically Guided Scheduling

Heuristics are commonly used to guide search in many combinatorial searching problems. For non-preemptive scheduling heuristics may be used to guide search along paths which are more likely to lead feasible schedules. Search is done in a depth first manner until either a complete feasible schedule is found, in which case the search terminates, or it is determined that no possible extensions of the current node can lead to a feasible schedule. Heuristics are used to determine which of the many children of a node should be searched next. Backtracking takes place when no further extensions of a node can be made. We evaluate temporal analysis using such a heuristic search for scheduling.

5 Empirical Evaluation of Temporal Analysis

In the previous sections, we have shown how temporal analysis may be used to restrict the search space for scheduling. Clearly, the existence of even a few precedence relations results in a drastic reduction of search space[3]. However, the usefulness of the scheme is not obvious since we are only interested in feasible schedules, hence a large part of the search space may never need to be examined. We have conducted various simulations to verify that indeed temporal analysis results in improved performance for scheduling. For reasons of space, we mention only a few significant results.

We used a heuristic search technique for scheduling as described in section 4.2. The heuristic used for our simulation study was a two level heuristic. The primary heuristic was *earliest start time*(EST).

$$EST_i = max(r_i, f_k)$$

where k is the last task in the partial schedule at that node.

In the case of a conflict, the secondary heuristic *earliest deadline* was used. Further conflicts were resolved arbitrarily. The heuristic has a natural intuitive appeal and is known to produce good results among linear heuristics[22].

For each set of parameters, we generated 200 "feasible" task sets with 100 tasks each. The task sets were generated with 100% utilization as this presents the most difficulty for scheduling. The computation times were generated using uniform distributions and laxities using normal distribution. We compared the *success percentage* (i.e. percentage of successfully scheduled task sets) of scheduling with and without temporal analysis as a pre-processing stage. The success percentage (SP) is plotted against "cut-off-time", indicating the maximum time allowed to the scheduling algorithm to successfully generate a schedule.

Our simulation results show that temporal analysis is not needed for scheduling when both the mean and the variation in laxities is low since the simple heuristics were able to schedule almost all task sets (success ratio ≈ 1.0). However, when the laxities are high (as compared to computation times) and the variation in laxities is also high[4], then the heuristics do

[3]Even one relation reduces the search space by half.

[4]Note that the task set utilization is 100%

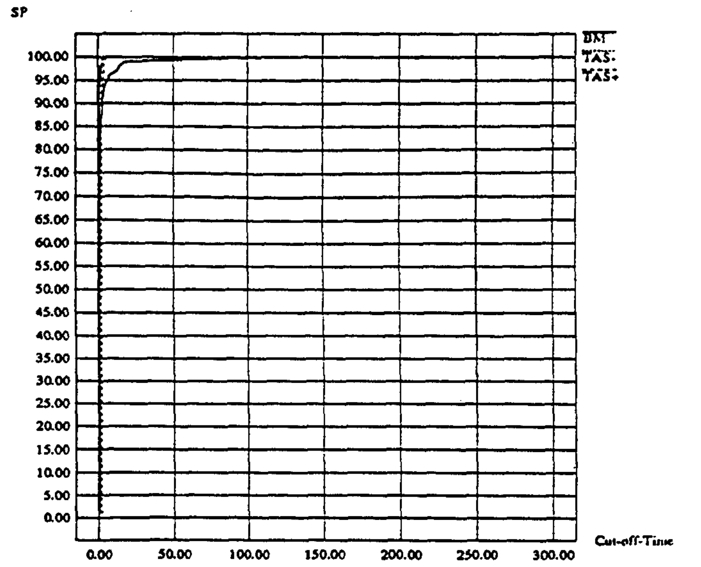


Figure 7: Success Ratio vs Cut-off-Time, $\mu_{\mathcal{L}} = 5.0\mu_{C}$, $COV_{\mathcal{L}} = 1.0$

not perform as well and the use of temporal analysis results in 10 – 20% improvement in success ratio.

As an illustration, we show a few plots which plot the success percentage (SP) of scheduling with temporal analysis (TAS) contrasted with success percentage of scheduling without temporal analysis, i.e. the baseline scheduling model (BM). For scheduling with temporal analysis, we consider two cases, one in which overhead of temporal analysis is added to scheduling time (TAS+) and the other in which it is not (TAS-). The parameters varied are the mean laxity $\mu_{\mathcal{L}}$ in terms of mean computation times μ_C, and the coefficient of variation for laxity $COV_{\mathcal{L}}$. Figures 7 and 8 show the plots for low laxity mean with low and high variation. For this case, there is no significant performance improvement due to temporal analysis and both schemes achieve almost 100% success percentage. On the other hand when the average laxity is high (Figures 9 and 10), coupled with high variation, we see that temporal analysis results in significant improvement in performance. The plots also show that the curves for (TAS+) and (TAS-) are almost identical showing that the overhead of temporal analysis is minimal when compared to the scheduling costs.

6 Concluding Remarks

In this paper we have presented *temporal analysis* as a technique for analyzing the timing relationships among a set of tasks to establish constraints on scheduling which are discernible from a pairwise analysis. The implications and the benefits of the approach as a pre-processing stage for scheduling has been shown through examples and simulation.

Time Driven Scheduling theory has relied heavily on search techniques for scheduling and little work has been done in

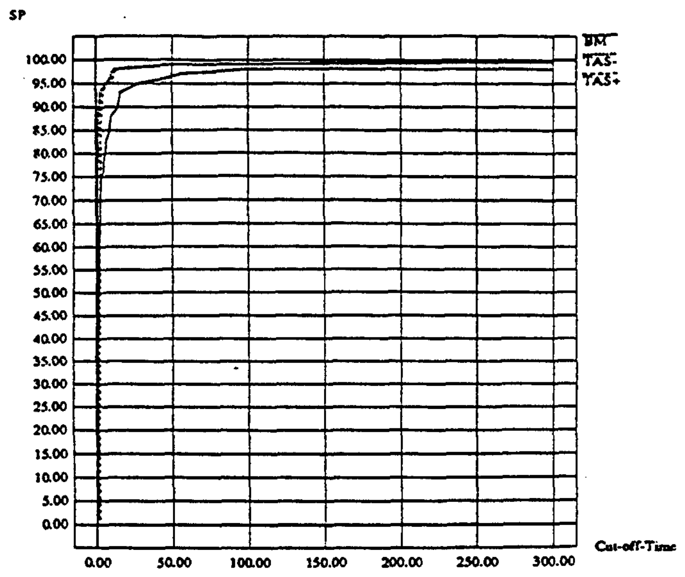


Figure 8: Success Ratio vs Cut-off-Time, $\mu_{\mathcal{L}} = 5.0\mu_{\mathcal{C}}$, $COV_{\mathcal{L}} = 2.0$

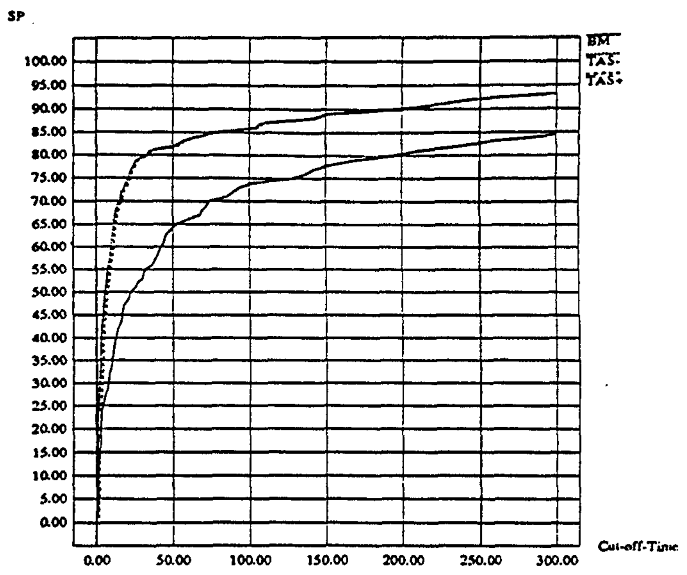


Figure 9: Success Percentage vs Cut-off-Time, $\mu_{\mathcal{L}} = 10.0\mu_{\mathcal{C}}$, $COV_{\mathcal{L}} = 1.0$

developing analytic techniques. Temporal analysis is a step in this direction and provides an efficient way of analyzing a task set and deducing valuable information for scheduling.

The existence of an infeasible relation in a task set gives a sufficient condition for infeasibility. This provides an early test for infeasibility, which can then be used as a basis for selective preemption to enhance feasibility. Alternatively, the detection of infeasibility may be used to allocate more resources or change resource allocation.

The precedence relations generated as a result of temporal analysis impose a partial order on the task set and may be effectively used to prune the search space for scheduling. Our simulations confirm that temporal analysis helps in improving the performance of a scheduling algorithm without incurring a significant overhead. In the simplest scheduling case, when heuristics perform very well, temporal analysis might be perceived as a way of formalizing the heuristics. For static time driven scheduling to be a feasible technique, it becomes imperative that the scheduling cost be controlled as the size of the problem increases. Temporal analysis provides a step in the right direction.

In this paper we have been concerned with single processor scheduling. An interesting extension of temporal analysis would be to use it for multi-processor scheduling. One way to extend the analysis to multi-processor scheduling is to perform it in two phases. In the first phase the infeasible and concurrent relations may be used to obtain an allocation of tasks to processors. Then in the second phase, the analysis shown in this paper can be used for each processor for scheduling.

Many real-time system specifications impose relative timing constraints on the tasks[23, 24]. In this paper, we have restricted ourselves to absolute constraints on the start and finish times of tasks. When more complex constraints are imposed on tasks, the role of temporal analysis in reducing the search space becomes even more important since simple heuristics are unlikely to perform well. It would be interesting to see how temporal analysis can be extended to use such constraints to further prune the search space.

We are currently implementing a scheduling tool based on the results shown in this paper. The tool is being developed for the MARUTI project, an experimental real-time system prototype being developed at the University of Maryland, based on the concept of pre-scheduling[6].

References

[1] J. Xu and D. L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems", *in Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pp. 132–146, December 1991.

[2] C. L. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment", *Journal of the ACM.*, vol. 20, pp. 46–61, Jan. 1973.

[3] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations", *IEEE Transactions on Software Engineering*, vol. SE-16, pp. 360–369, March 1990.

[4] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource requirements in a Hard Real-Time System", *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 564–577, May 1987.

[5] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger, "Distributed

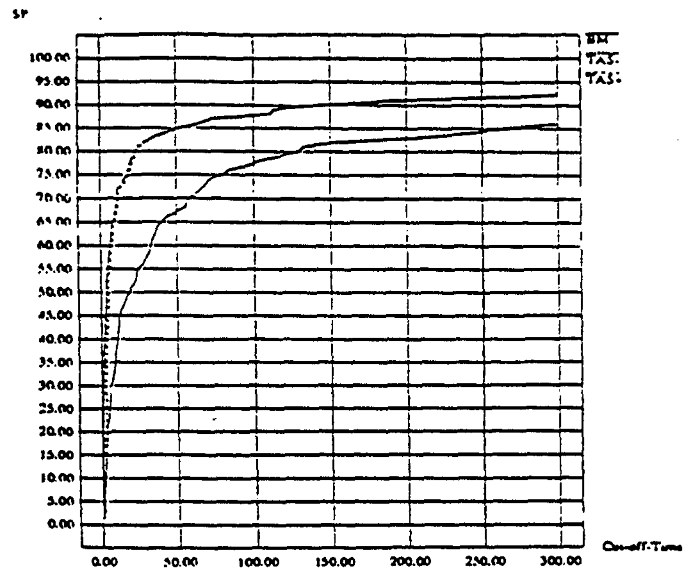


Figure 10: Success Percentage vs Cut-off-Time, $\mu_c = 10.0\mu_c$, $COV_c = 2.0$

Fault-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*, vol. 9, pp. 25–40, Feb. 1989.

[6] S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala, "The MARUTI Hard Real-Time Operating System", *ACM SIGOPS, Operating Systems Review*, vol. 23, pp. 90–106, July 1989.

[7] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *ACM SIGOPS, Operating Systems Review*, vol. 23, pp. 54–71, July 1989.

[8] E. G. Coffman, *Computer and Job-Shop Scheduling Theory, Ed.*, Wiley, New York, 1976.

[9] T. Shepard and J. A. M. Gagne, "A Model of The F-18 Mission Computer Software for Pre-Run Time Scheduling", *in Proceedings IEEE* 10^{th} *International Conference on Distributed Computer Systems*, pp. 62–69, May 1990.

[10] X. Yuan and A. K. Agrawala, "A Decomposition Approach to Nonpreemptive Scheduling in Hard Real-Time Systems", *in Proceedings IEEE Real-Time Systems Symposium*, Dec. 1989.

[11] J. Erschler, G. Fontan, C. Merce, and F. Roubellat, "A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates", *Operations Research*, vol. 31, pp. 114–127, Jan. 1983.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[13] M. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes", *Proceedings of the IFIP Congress*, pp. 807–813, 1974.

[14] Al Mok, *Fundamental Design Problems for the Hard Real-Time Environment*, PhD thesis, Massachussets Institute of technology, 1983.

[15] K. Jeffay, D. F. Stanat, and C. U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks", *in Proceedings IEEE Real-Time Systems Symposium*, pp. 129–139, December 1991.

[16] T. Shepard and J. A. M. Gagne, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, vol. 17, pp. 669–677, July 1991.

[17] G. McMahon and M. Florian, "On scheduling with ready times and due dates to minimize maximum lateness", *Operations Research*, vol. 23, pp. 475–482, May 1975.

[18] P. Bratley, M. Florian, and P. Robillard, "Scheduling with Earliest Start and Due Date Constraints", *Naval. Res. Log. Quart.*, vol. 18, pp. 511–519, Dec. 1971.

[19] K. R. Baker and Z. Su, "Sequencing with Due-Date and Early Start Times to Minimize Maximum Tardiness", *Naval Res. Log. Quart.*, vol. 21, pp. 171–176, 1974.

[20] J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, and E. Jansen, " A Static Scheduling Algorithm for Distributed Hard Real-Time Systems", *Journal of Real Time Systems*, pp. 227–246, 1991.

[21] G. Fohler and C. Koza, "Heuristic Scheduling for Distributed Real-Time Systems", MARS 6/89, Technische Universitat Wien, Vienna, Austria, April 1989.

[22] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints", *Journal of Systems and Software*, pp. 195–205, 1987.

[23] R. Gerber and W. Pugh and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks", Technical Report CS-TR-2985, UMIACS-TR-92-118, University of Maryland, Oct. 1992.

[24] C. C. Han and K. J. Lin, "Job scheduling with temporal distance constraints", Technical Report UIUCDCS-R-89-1560, University of Illinois at Urbana-Champaign, Department of Computer Science, 1989.

REPORT DOCUMENTATION PAGE		Form approved OMB No 074-0188
1. AGENCY USE ONLY (leave blank)	2. REPORT DATE 1/20/1993	3. REPORT TYPE END DATES COVERED Technical Reports

4. TITLE AND SUBTITLE Temporal Analysis and its Application in Non-Preemptive Scheduling	5. FUNDING NUMNBERS N00014-91-C-0195
6. AUTHOR(S) Manas C. Saksena and Ashok K. Agrawala	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science A.V. Williams Building College Park, MD 20742	8. PERFORMING ORGANIZATION REPORT NUMBER CS-TR-2698 UMIACS-TR-91-88
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) PhillipsLaboratory Director of Contracting 3651 Lowry Avenue SE Kirtland AFB, NM 87117-5777	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12.a. DISTRIBUTION/ AVAILABILITY STATEMENT	12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

In the problem of non-preemptive scheduling of a set of tasks on a single processor, each task has a ready time, a deadline and a computation time. A feasible schedule for this set requires that a start time be assigned to each task. The approaches taken for scheduling often use search techniques and may reduce the search by using heuristics. In this paper we present a technique for analyzing the temporal relations among the tasks to establish pairwise relationships among them. These relationships can then be used effectively to reduce the complexity of scheduling these tasks. We present simulation results to confirm the usefulness of temporal analysis as a phase prior to scheduling.

14. SUBJECT TERMS	15. NUMBER OF PAGES 7
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlisted

MSN 7540-01 280-5500 Standard Form 298 (Rev 2-89)

Implementation of the MPL Compiler[*][†]

Jan M. Rizzuto and James da Silva

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

February 14, 1995

Abstract

The Maruti Real-Time Operating System was developed for applications that must meet hard real-time constraints. In order to schedule real-time applications, the timing and resource requirements for the application must be determined. The development environment provided for Maruti applications consists of several stages that use various tools to assist the programmer in creating an application. By analyzing the source code provided by the programmer, these tools can extract and analyze the needed timing and resource requirements. The initial stage in development is the compilation of the source code for an application written in the Maruti Programming Language (MPL). MPL is based on the C programming language. The MPL Compiler was developed to provide support for requirement specification. This report introduces MPL and describes the implementation of the MPL Compiler.

[*]This work is supported in part by ONR and DARPA under contract N00014-91-C-0195 to Honeywell and Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, ONR, the U.S. Government or Honeywell.
Computer facilities were provided in part by NSF grant CCR-8811954.

[†]This work is supported in part by ARPA and Philips Labs under contract DASG60-92-0055 to Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, or the U.S. Government.

1 Introduction

A *real-time* system requires that an application meet the timing constraints specified for it. For *hard real-time*, a failure to meet the specified timing constraints may result in a fatal error [2]. Timing constraints are not as critical for *soft real-time*. The Maruti Operating System was developed to meet the real-time constraints required by many applications. In order to schedule and run an application under Maruti, the timing and resource requirements for that application must be determined. The development environment for Maruti consists of several tools that can be used to extract and analyze these requirements [2].

The *Maruti Programming Language* (MPL) is a language developed to assist users in creating applications that can be run under Maruti. MPL is based on the C programming language, and assumes the programmer is familiar with C. MPL provides some additional constructs that are not part of standard C to allow for resource and timing specification [1]. In addition, when an MPL file is compiled, some of the resource requirements can be recognized and recorded to an output file. This output file is used as input to the integration stage, which is the next stage in the development cycle. During integration, additional timing requirements may be specified.

Previously, an MPL file was compiled by first running the source code through the Maruti pre-compiler, which created a C file that was then compiled using a C compiler [1]. The pre-compiler extracted the necessary information, and converted the MPL constructs that were not valid C statements into C code. This required the additional pass of the pre-compiler over the source code. We have created a compiler for MPL that integrates both the actions of the pre-compiler and the compiler into one stage. In this report, we present MPL, and a description of the compiler we implemented. Section 2 defines the abstractions used in Maruti. In Section 3, the syntax of the constructs unique to MPL is defined. The details of the implementation of the compiler are given in Section 4. Section 5 describes the resource information that is recorded during compilation. Conclusions appear in Section 6, followed by an Appendix containing a sample MPL file, and the resource information recorded for that file.

2 Maruti Abstractions

An MPL application is broken up into units of computation called *elemental units* (EUs). Execution within an EU is sequential, and resource and timing requirements are specified for each EU. A *thread* is a sequential unit of execution that may consist of multiple EUs. MPL allows threads of execution to be specified by the programmer through several of the constructs provided. A *task* consists of a single address space, and threads that execute in that address space. *Modules* contain the source code of the application as defined by the programmer. An application may consist of several modules. During execution, modules are mapped to one or more tasks.

3 MPL Constructs

There are several constructs defined in MPL that are not a part of standard C. These constructs have been implemented in the MPL compiler.

3.1 Module Name Specification

A *module* may consist of one or more source files written in MPL. At the start of each MPL file, the name of the module that the source file corresponds to must be indicated. This is given by the following syntax: .

```
module-name-spec ::= 'module' <module-name> ';'.
```

The module-name may be any valid identifier that is accepted by standard C. The module name specification must appear at the beginning of the source file, before any other MPL code. The specification is not compiled into any executable code. It is simply used to indicate the module that the functions within the file belong to.

3.2 Shared Buffers

A *shared buffer* can be used to declare memory that may be shared by several tasks, to permit communication between the tasks. A declaration of a shared buffer requires the type be defined as with a variable declaration. The syntax of a shared declaration is:

```
shared-buffer-decl ::= 'shared' <type-specifier> <shared-buffer-name>.
```

The shared-buffer-name can be any valid identifier, and the type-specifier can be any valid type for a variable. A shared declaration is compiled as a pointer to the type given in the declaration of the shared buffer, rather than the type given.

3.3 Region Constructs

The are two constructs used to allow for mutual exclusion within an application.

3.3.1 Region Statement

The *region* statement is used to enforce mutual exclusion globally throughout an entire application, and is given by the syntax:

```
region-statement ::= 'region' <region-name>
                     { mpl-statements }.
```

The mpl-statements may be any number of valid MPL statements. These statements make up a critical section.

3.3.2 Local Region Statement

The *local_region* statement is used to enforce mutual exclusion within a task, and follows the same syntax of the region statement:

```
local-region-statement ::= 'local_region' <local-region-name>
                           { mpl-statements }.
```

3.4 Channel Declarations

Channels are used to allow for message passing within a Maruti application. Each channel declared has a type associated with it given by a valid C type-specifier. This type indicates the type of data that the channel will carry.

Channels may be declared in both *entry* and *service functions*, which will be defined below. The syntax for channel declarations is:

```
channel-declaration-list-opt ::= { channel-declaration-list }.
channel-declaration-list ::= channel-declaration { channel-declaration }.
channel-declaration ::= channel-type channels ';'.
channel-type ::= 'out' | 'in' | 'in-first' | 'in-last'.
channels ::= channel { ',' channel }.
channel ::= <channel-name> ':' type-specifier.
```

3.5 Entry Functions

An *entry function* is a special type of function that may be defined in an MPL source file. Each entry function corresponds to a thread within the application. The syntax for an entry function definition is:

```
entry-function ::= 'entry' <entry-name> '(' ')' entry-function-body.
entry-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.6 Service Functions

Service functions are another type of special function supported by MPL. A service function is invoked when a message is received from a client. Each service function definition requires an *in* channel and message buffer be included in the definition. The service function will be executed when there is a message on the channel given in the definition. The definition of a service function is similar to that of an entry function:

```
service-function ::= 'service' <service-name>
                '(' <in-channel-name> ':' type_specifier ',' <msg-ptr-name> ')'
                service-function-body.
service-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.7 Communication Function Calls

There are several library functions used to allow for message passing within a Maruti application.

3.7.1 Send Calls

Each call to the *send* function must specify an outgoing channel for the message:

```
void send ( channel channel_name, void *message_ptr );
```

3.7.2 Receive and Optreceive Calls

Both *receive* calls, and *optreceive* calls must be associated with an incoming channel (*in*, *in_first*, or *in_last*):

```
void receive ( channel channel_name, void *message_ptr );
int optreceive ( channel channel_name, void *message_ptr );
```

A call to receive requires that there be a message on the incoming channel. Optreceive should be used when a message may or may not be on the channel. Optreceive checks for the message, and returns a value indicating if a message was found.

3.8 Initialization Function

Each task has an *initialization routine* that is executed when the application is loaded. This function is specified by the user with the following name and arguments:

```
int maruti_main (int argc, char **argv)
```

4 Implementation

We started with version 2.5.8 of the Gnu C compiler. By modifying the source code for the C compiler, we have created a compiler for applications written in MPL. In addition to what the standard Gnu C compiler does, this modified compiler handles the additional constructs defined in MPL, and records information about the source code that is needed by Maruti. A source code file written in MPL is specified with an *mpl* extension.

4.1 Modifications to GCC File Structure

In the process of modifying the compiler, some existing files were modified. In addition, some new files were also created. The source code for version 2.5.8 of GCC allows compilers to be created for several different languages: C, C++, and Objective C. The GCC compiler uses different executable files for the different languages that it compiles. There are separate files for C, C++, and Objective C *(cc1, cc1plus, cc1obj)*. The GCC driver, *gcc.c*, uses the extension of the source file specified to determine the appropriate executable (and therefore language) to compile the source file. The driver then executes the compiler, passing on the appropriate switches. The driver was modified to accept input files with an *mpl* extension. *Cc1mpl* is the new executable that was created to compile MPL source files. When a file with an *mpl* extension is specified as a source file to be compiled, this new executable file is used. When an MPL file is compiled, it automatically passes on the switch *-Maruti_output*, which indicates that the needed output should be recorded to a file with an *eu* extension.

The executable files for each language are composed of many object files. Some of these files are common to all the languages, and some of the files are language-specific. The language-specific files added for compiling MPL files are those files with an *mpl-* prefix.

Gperf is a tool used to generate a perfect hash function for a set of words. Gperf is used to create a hash function for the reserved words for each language. The files containing the input to gperf are indicated by a file name with a *gperf* extension. There are several different **.gperf* files containing the reserved words for the different languages recognized by the compiler. The *mpl-parse.gperf* file contains all the reserved words for C, in addition to those added for MPL. For each language, the output from running gperf is then incorporated into the **-lex.c* file. This output includes a function *is_reserved_word()* that is used to check if a token is a reserved word. The file *mpl-lex.c* is basically the *c-lex.c* file, with the output of running gperf on *mpl-parse.gperf* instead of *c-parse.gperf*.

The file *maruti.c* contains the routines that have been written to implement MPL. This file is linked in with the executable for all of the languages, to prevent undefined symbol errors from occuring. Calls to the routines contained in this file occur in both the language-specific, and the common files. The flag *maruti_dump* is set in *main()* to indicate whether information about the source code should be recorded to the appropriate output file. This flag prevents calls to the routines in *maruti.c* which are made in the common files from occuring for the languages other than MPL. The files containing these calls are:

- *calls.c*
- *explow.c*
- *expr.c*
- *function.c*
- *toplev.c*

There are several reasons why the new language-specific files have to be created for MPL. The files *mpl-lex.h* and *mpl-lex.c* needed to be created for MPL because MPL contains several additional reserved words not present in C, as mentioned earlier. The file *c-common.c* relies on information in the header file *c-lex.h*. Since MPL uses *mpl-lex.h*, *mpl-common.c* includes *mpl-lex.h*, instead of *c-lex.h*. *Bison* is a tool that allows a programmer to define a grammar through rules, and converts them into a C program that will parse an input file. The **-parse.y* files are the bison files used to create the grammar to parse a source file. Since the grammar for MPL needed to be modified to accept the additional constructs, the *mpl-parse.y* file was created. There is one function used in compiling MPL source files that is defined in *mpl-parse.y*, instead of *maruti.c*. This function needed to access the static variables declared in *mpl-parse.y*, and in order to do so, the function definition was placed in that file. Finally, the file *mpl-decl.c* was created, because of its dependence on *mpl-lex.h*, and also to allow for an additional type specification used in MPL.

4.2 Compiling MPL Constructs

MPL extends the C language to allow for various constructs. In order to implement these extensions, the grammar used to recognize C in GCC had to be extended. The following are recognized as reserved words for MPL, in addition to the standard reserved words for C: *shared*, *region*, *local_region*, *module*, *in*, *out*, *in_first*, *in_last*, *entry*, *service*, *send*, *receive*, and *optreceive*. The keywords *in* and *out* were reserved words in the *c-** files, because they are used by Objective C, but in MPL they are used as channel types. In addition to the new reserved words, rules were added and modified resulting in the rules in *mpl-parse.y*.

4.2.1 Module Name Specification

A rule was added to the grammar to parse the module name specification in an MPL file. The rule for a whole program was also modified to include this module statement. This rule expects the module statement to appear before any other definitions. Since the module name specification does not result in any executable code, the only action taken is to record the module name given by the programmer.

4.2.2 Shared Buffers

There are no rules added to the grammar for a shared buffer declaration. When a variable declaration is parsed, a tree is created that keeps track of all the specification information given for that declaration. For example, *typedef* and *extern* are two of the possible type specifications. The token *shared* is recognized as a type specification, just as *typedef* and *extern* are recognized. When a declaration is made, these specifications are processed in the function *grokdeclarator()* in *mpl-decl.c*. When a shared specification is encountered, the declaration is converted to a pointer to the type specified, instead of just the type specified. Other than this conversion to a pointer, the declaration is compiled just as any other declaration would be compiled in C.

4.2.3 Region Constructs

The region constructs are considered statements in MPL. Several rules were added to parse these constructs, and the *region* and *local_region* statements were added as options for a valid statement in the grammar for MPL.

Both region and local_region statements are compiled in the same manner. Each region has a name, and a body which is the code within the critical section. In order to protect these critical sections, calls are made to the Maruti library function *maruti_eu()*. When a region is parsed, the compiler generates two calls to *maruti_eu()*, in addition to the code in the body of the region. The first call is generated just before the body, and the second call just after. These calls are generated through functions in *maruti.c*. The functions are based on the actions that would have been taken, had the parser actually parsed the calls to *maruti_eu()* in the source file.

4.2.4 Channel Declarations

The rules added for a channel declaration allow any number of channels to be declared in either an entry or a service function. Each channel declaration requires several pieces of information:

- *Channel-type*
- *Channel-name*
- *Type specifier indicating the type of data that channel carries*

A linked list of declared channels is maintained. For each declared channel the following information is saved:

- *Channel-name*
- *Type information*
  1. *Size in bytes*
  2. *String encoding the type of the data*
- *Channel-id*

The *channel-id* is a unique identification number assigned to each declared channel. Channel declarations do not add to the compiled code. The channels are not allocated memory. The information describing each channel is simple stored in the linked list. During compilation, whenever a channel is referenced, the appropriate information is obtained from this list.

4.2.5 Entry Functions

Entry function definitions are compiled differently than other function definitions. An entry function would appear in an MPL file in the following form:

```
entry <entry_name> ()
<channel_declaration_list_opt>
{
  <mpl_function_body>
}
```

Where `entry_name` is an identifier that is the name of the entry function, the `channel_declaration_list_opt` contains any channels the user wants to define for that function, and `mpl_function_body` is any function body that would be accepted as a definition in a standard MPL function. Semantically the entry function is equivalent to the following MPL code:

```
_maruti_entry_name ()
{
  while(1)
  {
    maruti_eu();
    entry_name () ;
  }
}

entry_name ()
{
  mpl_function_body
}
```

An entry function is compiled into two functions, as if the two functions given above had been part of the source file. Essentially, the first function is just a stub function that calls *maruti_eu()*, then calls the second function compiled. As with generating function calls, the routines to generate the code for entry function definitions are based on the actions that would have been taken had the parser actually parsed the code for the two separate functions.

4.2.6 Service Functions

Service functions definitions are handled very much like entry function definitions. The syntax of a service function differs slightly from that of an entry function, since it requires that an incoming channel and a message buffer be defined:

```
service <service_name> (<in_channel_name> : <type_specifier>, <msg_ptr_name>)
<channel_declaration_list_opt>
{
   <mpl_function_body>
}
```

Like the entry functions, service functions are semantically equivalent to two functions, where one is simply a stub function calling the second function that is generated:

```
_maruti_service_name ()
{
type_specifier _maruti_msg_ptr_name ;

  while(1)
  {
    if ( optreceive ( _maruti_in , id  , & _maruti_msg_ptr_name, size ) )
    {
       service_name (& _maruti_msg_ptr_name );
    }
  }
}

service_name (msg_ptr_name)
type_specifier  *msg_ptr_name;
{
  mpl_function_body
}
```

The `service_name`, `channel_declaration_list`, and `mpl_function_body` are all the same as described previously for entry functions. In addition, service functions have two other items specified in their definitions. The first is a channel. Every service function requires a channel be specified. This channel is always declared as an *in* channel with the name `in_channel_name`. The type is given by `type_specifier` as if it had been declared in the `channel_declaration_list`. The channel is used to invoke the service function. This in channel is used by the optreceive in the stub function that calls the function containing the service function body. When a message is received on this channel, the service function is executed. The second additional item is a message buffer used by the service function. The name of this message buffer is given by `msg_ptr_name`, the type is given by `type_specifier`. This buffer is used to hold the message received from the client that invoked the service function, and is passed to the second function containing the body of the service function.

4.2.7 Communication Function Calls

There were three library functions provided for message passing mentioned previously: send, receive, and optreceive. Function calls to any of these three library functions are handled differently than other function calls. In the MPL grammar, *send, receive*, and *optreceive* are all reserved words. The MPL syntax for all of these calls is the following:

```
<function-name> (<channel-name>, <parameter-2>);
```

`Channel-name` should be a previously declared channel, and `parameter-2` should be a pointer. These function calls must be compiled differently, since these are not the actual parameters used when the call is generated. In the case of a call to send, the actual parameters must be as follows:

```
send (<channel-id>, <parameter-2>, <channel-size>);
```

In the case of a call to either receive, or optreceive, the parameters required are:

```
receive | optreceive (<channel-type>, <channel-id>, <parameter-2>, <channel-size>);
```

The `channel-type` for a receive or optreceive call is an integer generated by the compiler that will indicate an *in*, *in_first*, or *in_last* channel.

When one of these three function calls are encountered, there are special rules in the grammar to handle it. A function in *maruti.c* is called which generates the appropriate parameters, and then the function call itself. These function calls are generated as mentioned above for the calls to *maruti_eu()*. The `channel-name` specified by the user is used to obtain the necessary parameters. Given the channel name, the linked list of channels is searched to find the corresponding channel, then the `channel-id` and the `channel-size` are obtained from that node in the linked list. There is also some type checking done at this stage. The compiler verifies that only an outgoing channel is specified for a send call, or an incoming channel for the receive and optreceive calls. The compiler also checks that any channel referenced has been previously defined.

The grammar for MPL was modified so that a call to any of the communication functions may occur anywhere that a primary expression occurs, since that is where other function calls are permitted to occur.

4.2.8 Initialization Function

The user-defined function *maruti_main()* is compiled as an ordinary C function.

5 PEUG File

The source code of an MPL file is broken up into *elemental units*. Each elemental unit identifies the resources that it requires. These elemental units are used later in the development process for scheduling the application. The output file created by the MPL compiler creates a *Partial Elemental Unit Graph* (PEUG) for the given source file. The name of this file is the name of the source file, with the *mpl* extension replaced by an *eu* extension.

There are several different types of information recorded in this PEUG file.

5.1 Module Name

The first line in the output file indicates the name of the module, and will appear as:

```
peug <module-name>
```

The `module-name` is taken directly from the module name specification given in the MPL source file.

5.2 File Name

The second line in the source file indicates the name of the target file that is created by the compiler, where `file-name` is the target:

```
file <file-name>
```

5.3 Shared Buffers

Each time a shared buffer is declared its name and type information is recorded to the output file:

```
shared <shared-buffer-name> : (type-description-string>, <type-size>)
```

The `type-description-string` and `type-size` of a shared buffer is obtained from the type specification, and is represented in the same manner as the type and size for a channel. Although the shared buffer is actually a pointer to the type it is declared as, the `type-description-string` represents the object being pointed to, and not the pointer itself.

5.4 Entry, Service, and User Function Definitions

In MPL, a user may define ordinary functions in addition to the entry and service functions that are permitted in MPL. For each entry, service or ordinary user-defined function, there is an entry in the output file. This entry has the following format:

```
<function-type> <function-name>
            .
            .
            .
            size <stack-size>
```

`Function-type` can be either function, entry, or service, indicating which type of function is being defined. `Function-name` is the declared name of the function in the source file. `Stack-size` is the maximum stack size needed by this function. This `stack-size` includes the arguments pushed onto the stack preceding any function calls occuring within the function body. There will also be other information concerning the body of the function that will appear between the `function-name`, and the `stack-size`. The entry for the *maruti_main()* function will be the same as those for other user defined functions. Entry and service functions will contain some additional information not applicable to ordinary functions that will be described below.

5.4.1 Channels

For each channel that is declared, a description of the channel is written to the output file. These descriptions will occur right after the statement indicating the name of the current function:

```
            <channel-type> <name> : (<description-string>, <size>)
```

The `channel-type` and `channel-name` will be the type and name specified in the source file. The `description-string` and `size` are based on the type specification in the channel declaration. Channel descriptions will occur only in entry and service functions. A service function will always contain at least one channel description, since the syntax of a service function requires a channel be named in the definition. A channel description will also be output for every send, receive, and optreceive call, since these calls require a channel as one of their parameters.

5.4.2 Function Calls

Each time a function call is parsed, there will be a line in the output file:

```
calls <function-name> {in_cond} {in_loop}
```

This line indicates where a function call occurs, and which function is being called. The `in_cond` and `in_loop` indicate if this function call appears within a conditional statement or within a loop. These labels will be seen only if their respective conditions are true.

5.4.3 Communication Function Calls

Any call to a communication function is recorded similarly to other function calls. There is a line indicating the name of the function, as shown above for a function call. In addition, there will be a line describing the channel associated with that communication function call. This line will appear just as the line for the channel definition described above appears.

5.4.4 EU Boundaries

The output file for an MPL source file indicates where each elemental unit (EU) begins by the following:

```
eu <N> {region_list}
```

The N indicates an EU number. Each EU within a source file has a unique number. There are several places where EU boundaries are created:

- *Start of a function*
- *Start of a region*
- *End of a region*
- *Explicit calls to maruti_eu()*

The initial EU occuring at the beginning of a function that is not a service or entry function is a special case. This is always labeled as "eu 0" in the output file, and does not represent an actual EU.

Each EU may also be followed by a list describing one or more regions. This list represents the regions that this EU occurs within. The description of a region appears as:

```
(region-name instance access type)
```

The `region-name` is just that given by the user, and the `type` indicates if a region is *local* (local_region construct) or *global* (region construct). The `access` indicates if the access is read or write. The `instance` indicates the instance of this region within the source file. Each instance for a region within a source file is unique.

6 Conclusions

Basing MPL on C has simplified the development of both the language and its compiler. The language is easy to learn for any programmer that has used C before, since there are a limited number of additional constructs unique to MPL. Using the GCC C source code provided an existing compiler, rather than implementing a new one. The source code for GCC only needed to be modified to handle some additional constructs, and produce some additional output. This made the implementation fairly simple. However, the GCC C compiler also provides some functionality that is not needed by MPL. Much of this functionality provided is not even permitted. These restrictions are not enforced by the compiler, but should be detected within the development cycle.

Prior to the development of the MPL compiler using GCC, compiling an MPL source file required two steps. The source files were initially passed through a pre-compiler to extract the available resource information and parse the MPL constructs. The pre-compiler was responsible for converting the MPL code into valid C code, which was then compiled using a standard C compiler. The new implementation of the compiler eliminates some of the redundant processing that is done when the pre-compiler is used. The information obtained through the pre-compiler already existed in the internal structure used by the GCC compiler. This information just needed to be recorded. Instead of parsing source code files in the two steps independently, the functionality of the pre-compiler has been incorporated into the compiler itself. The MPL compiler provides a single tool that extracts all the available information at the initial stage of develpment.

In the future, a version of MPL may be implemented that is based on the Ada programming language. GNAT is a compiler for Ada 9X that is being developed at NYU. GNAT depends on the backend of the GCC compiler. Using the source code for GNAT, an implementation of MPL based on Ada would be similar to the current implementation based on C.

Appendix

A MPL File

The following is a sample of MPL source code:

```
module timer;

typedef struct {
   int seconds;
   int minutes;
   int hours;
   } time_type;

shared time_type global_time;

maruti_main(argc, argv)
int argc;
char **argv;
{
global_time->seconds = 0;
global_time->minutes = 0;
global_time->hours = 0;

return 0;
}

entry update_second()
out disp : time_type;
{
time_type msg;

region time_region {
   global_time->seconds++;
   if (global_time->seconds == 60)
      global_time->seconds = 0;
   msg = *global_time;
   }

send (disp, &msg);
}

entry update_minute()
out display : time_type;
{
time_type msg;

region time_region {
   global_time->minutes++;
   if (global_time->minutes == 60)
      global_time->minutes = 0;
   msg = *global_time;
   }

send (display, &msg);
}

entry update_hour()
out display : time_type;
{
time_type msg;

region time_region {
   global_time->hours++;
   if (global_time->hours == 24)
      global_time->hours = 0;
   msg = *global_time;
   }

send (display, &msg);
}

service display_time(inchan : time_type, time)
{
printf("Current Time: %d : %d : %d", time->hours, time->minutes, time->seconds);
}
```

B PEUG File

The corresponding PEUG file for the source code above is:

```
peug timer
file timer.o
shared global_time : ($(iii), 12)
function maruti_main
        eu 0
        size 4
entry update_second
        out disp : ($(iii), 12)
        eu 2
        eu 3 (time_region 1 W global)
        calls maruti_eu
        eu 4
                calls maruti_eu
                calls send
                out disp : ($(iii), 12)
        size 32
entry update_minute
        out display : ($(iii), 12)
        eu 5
        eu 6 (time_region 2 W global)
                calls maruti_eu
        eu 7
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
entry update_hour
        out display : ($(iii), 12)
        eu 8
        eu 9 (time_region 3 W global)
                calls maruti_eu
        eu 10
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
service display_time
        in inchan : ($(iii), 12)
        eu 11
                calls optreceive
                in inchan : ($(iii), 12)
                calls printf
        size 52
```

References

[1] James da Silva, Eric Nassor, Seongsoo Hong, Bao Trinh, and Olafur Gudmundsson. Maruti 2.0 Programmer's Manual. Unpublished.

[2] Manas Saksena, James da Silva, and Ashok Agrawala. Design and Implementation of Maruti-II. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 4. Prentice Hall, 1995.

[3] Richard Stallman. The GNU C compiler, version 2.5.8., Manual. Info file obtained from gcc.texi in source code distribution.

OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE February 14, 1995	3. REPORT TYPE AND DATES COVERED Technical Report

4. TITLE AND SUBTITLE

Implementation of the MPL Compiler

5. FUNDING NUMBERS

N00014-91-C-0195 and
DSAG-60-92-C-0055

6. AUTHOR(S)

Jan M. Rizzuto and James da Silva

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Maryland
A.V. Williams Building
College Park, Maryland 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

CS-TR-3413
UMIACS-TR- 95-17

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Honeywell
3660 Technology Drive
Minneapolis, MN 55418

Phillips Labs
3550 Aberdeen Ave. SE
Kirtland AFB, NM
87117-5776

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Maruti Real-Time Operating System was developed for applications that must meet hard real-time constraints. In order to schedule real-time applications, the timing and resource requirements for the application must be determined. The development environment provided for Maruti applications consists of several stages that use various tools to assist the programmer in creating an application. By analyzing the source code provided by the programmer, these tools can extract and analyze the needed timing and resource requirements. The initial stage in development is the compilation of the source code for an application written in the Maruti Programming Language (MPL). MPL is based on the C programming language. The MPL compiler was developed to provide support for requirement specification. This report introduces MPL and describes the implementation of the MPL Compiler.

14. SUBJECT TERMS

D.3.2, Language Classifications
D.4.7, Organization and Design

15. NUMBER OF PAGES

17 pages

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

NSN 7540-01-280-5500 Standard Form 298 (Rev 2-89)

Maruti 3.1 Programmer's Manual First Edition

Systems Design and Analysis Group
Department of Compter Science
University of Maryland at College Park

December 1996

Contents

Chapter 1

Introduction

The *Maruti Programming Language* (MPL) is used to write Maruti application code. Currently, MPL is based on the ANSI C programming language, with extensions to support modules, real-time constructs, communications primitives, and shared memory.

The *Maruti Configuration Language* (MCL) is used to specify how individual program modules are to be connected together to form an application and the details of the hardware platform on which the application is to be executed.

1.1. General Program Organization

A complete Maruti system is called an *application*. Applications can be large, distributed systems made up of many subsystems. Each application is defined by a *configuration file*, which defines all the subsystems and their interactions. The following entities make up an application:

Jobs Jobs are the active entities in a Maruti application. Jobs are specified in the configuration file with timing constraints, including the job period. A job is made up of multiple *entry points*, which are the threads of execution that will be run for the job.

Modules The code of an application is divided into modules. Each module consists of *entry points*, which define the code which will be executed as part of a job, *services*, which define code to be invoked on behalf of a client module, and *functions*, which are called from entries and services.

Tasks At run-time, modules map to tasks (a module may be mapped to more than one task). Each task consists of an *address spac* and *threads* of execution for the entry points and services of the module.

Channels Channels are the communication paths for Maruti applications. Each channel is a one-way connection through which typed messages are passed. The end points are defined by *out* and *in* channel specifiers, and are connected as specified in the application configuration file. Each end point is associated with one entry or service, and its message type and channel type are declared within the entry or service header. The types of the in and out channel specifiers must match.

Regions Regions are the mechanism for mutual exclusion between Maruti threads: only one thread can enter a particular region at a time. Two types of regions may be specified: *global* regions enforce exclusion for the entire Maruti application, while *local* regions enforce exclusion only within a single task.

Shared buffers Named memory buffers can be shared between tasks. The buffer is mapped into the address space of each task that uses that buffer.

1.1.1. Maruti Programming Language

Rather than develop completely new programming languages, we have taken the approach of using existing languages as base programming languages and augmenting them with Maruti primitives needed to provide real-time support.

In the current version, the base programming language used is ANSI C. MPL adds *modules*, *shared memory blocks*, *critical regions*, *typed message passing*, *periodic functions*, and *message-invoked functions* to the C language. To make analyzing the resource usage of programs feasible, certain C idioms are not allowed in MPL; in particular, recursive function calls are not allowed nor are unbounded loops containing externally visible events, such as message passing and critical region transitions.

- The code of an application is divided into *modules*. A module is a collection of procedures, functions, and local data structures. A module forms an independently compiled unit and may be connected with other modules to form a complete application. Each module may have an *initialization function* which is invoked to initialize the module when it is loaded into memory. The initialization function may be called with arguments.

- Communication primitives send and receive messages on one-way, typed *channels*. There are several options for defining channel endpoints that specify what to do on buffer overflow or when no message is in the channel. The connection of two end-points is done in the MCL specification for the application - Maruti insures that end-points are of the same type and are connected properly at runtime.

- Periodic functions define *entry points* for execution in the application. The MCL specification for the application will determine when these functions execute.

- Message-invoked functions, called *services*, are executed whenever messages are received on a channel.

- *Shared memory blocks* can be declared inside modules and are connected together as specified in the MCL specifications for the application.

- *Critical Regions* are used to safely maintain data consistency between executing entities. Maruti ensures that no two entities are scheduled to execute inside their critical regions at the same time.

1.1.2. Maruti Configuration Language

MPL Modules are brought together into as an executable application by a specification file written in the Maruti Configuration Language (MCL). The MCL specification determines the application's hard real-time constraints, the allocation of tasks, threads, and shared memory blocks, and all message-passing connections. MCL is an interpreted C-like language rather than a declarative language, allowing the instantiation of complicated subsystems using loops and subroutines in the specification. The key features of MCL include:

- **Tasks, Threads, and Channel Binding.** Each module may be instantiated any number of times to generate tasks. The threads of a task are created by instantiating the entries and services of the corresponding module. An entry instantiation also indicates the job to which the entry belongs. A service instantiation belongs to the job of its client. The instantiation of a service or entry requires binding the input and output ports to a channel. A channel has a single input port indicating the sender and one or more output ports indicating the receivers. The configuration language uses channel variables for defining the channels. The definition of a channel also includes the type of communication it supports, i.e., synchronous or asynchronous.

- **Resources.** All global resources (i.e., resources which are visible outside a module) are specified in the configuration file, along with the access restrictions on the resource. The configuration language allows for binding of resources in a module to the global resources. Any resources used by a module which are not mapped to a global resource are considered local to the module.

- **Timing Requirements and Constraints.** These are used to specify the temporal requirements and constraints of the program. An application consists of a set of cooperating jobs. A job is a set of entries (and the services called by the entries) which closely cooperate. Associated with each job are its invocation characteristics, i.e., whether it is periodic or aperiodic. For a periodic job, its period and, optionally, the ready time and deadline within the period are specified. The constraints of a job apply to all component threads. In addition to constraints on jobs and threads, finer level timing constraints may be specified on the observable actions. An observable action may be specified in the code of the program. For any observable action, a ready time and a deadline may be specified. These are relative to the job arrival. An action may not start executing before the ready time and must finish before the deadline. Each thread is an implicitly observable action, and hence may have a ready time and a deadline.

Apart from the ready time and deadline constraints, programs in Maruti can also specify *relative* timing constraints, those which constrain the interval between two events. For each action, the start and end of the action mark the observable events. A relative constraint is used to constrain the temporal separation between two such events. It may be a relative deadline constraint which specifies the upper bound on time between two events, or a delay constraint which specifies the lower bound on time between the occurrence of the two events. The interval constraints are closer to the event-based real-time specifications, which constrain the minimum and/or maximum distance between two events and allow for a rich expression of timing constraints for real-time programs.

- **Replication and Fault Tolerance.** At the application level, fault tolerance is achieved by creating resilient applications by replicating part, or all, of the application. The configuration language eases the task of achieving fault tolerance, by allowing mechanisms to replicate the modules, and services, thus achieving the desired amount of resiliency. By specifying allocation constraints, a programmer can ensure that the replicated modules are executed on different partitions.

Chapter 2

Tutorial

2.1. Basic Maruti Program Structure

Maruti applications are built up out of one or more MPL modules, and tied together with a *configuration file* written in MCL. We'll start our tutorial with an explanation of a very simple application consisting of one module, called simple.mp. Our simple application will contain a producer thread that sends out integer data, and a consumer thread, which receives integer values and prints them out.

The Module

```
module simple;

int data;

maruti_main(int argc, char **argv)
{
     if(argc < 1) {
          printf("simple: requires an integer argument\n");
          return 1;
     }

     data = atoi(argv[0]);
     return 0;
}
```

This first part of the module will be similar in all Maruti modules. The module always starts with the module name declaration. After the module declaration, the MPL module is much like any ANSI C program, but with some special Maruti definitions.

Every module must contain a function named maruti_main, which initializes the module at load time. This initialization would normally include things like device probing or painting the screen. The maruti_main function, exactly like the main function of a C program, takes an argument count and list as its parameters, and returns an error code to its environment. In Maruti, the environment is the system loader, and any non-zero return results in a load failure, in which case the application will not run. In our example, maruti_main is responsible for setting the initial value of our datum from the environment, and returning a failure code if there is no argument.

Periodic Functions

```
entry producer()
    out och: int;
    {
        data++;                    /* produce data */
        send(och, &data);
    }
```

The producer is a periodic function, or Maruti *entry point.* It serves as the top-level function for a Maruti thread that will be invoked repeatedly, with a period specified in the MCL config file (which we will see below).

The producer outputs its data on a Maruti *channel,* using the built in MPL send function. The channel och is declared as part of the function header of producer. Maruti channels are declared to have a type, usually a structure but in this case a simple integer. All messages sent on the channel will be of the same type.

Note that there is no open, bind, or connect statement needed to initiate communication on the channel. The connection of the channel will be specified in the config file, and initiated automatically by the runtime system.

Message-Invoked Functions

```
    service consumer(ich: int, msg)
    {
        printf("consumer got %d\n", *msg);     /* consume data */
    }
```

The consumer is a message-invoked function, or Maruti *service.* It serves as the top-level function for a Maruti thread that is invoked whenever there is a message delivered on the channel declared in the function header. The msg parameter is the name of the pointer to the message buffer that will contain the delivered message.

Since the receipt of the invoking message is automatic for a Maruti service, the only thing our consumer has to do is print out the data value contained in the message.

This completes our simple module, but in order to have a Maruti application, we must have a config file that tells the system how to run our program.

The Config File

The config file is written in the Maruti Configuration Language (MCL), an interpreted C-like language with constructs that allow an application to be built up from pieces and interconnected. The MCL processor, called the *integrator,* builds a program graph from the specifications, analyses it for type correctness and completeness, and checks for dependency cycles. Here is the config file, *simple.cfg,* that goes with our application :

```
application simple {

    job j;                  /* declare variables */
    task si;
    channel c;

    init j: period 1 s;     /* specify job parameters */
    start si: simple(27);   /* specify task parameters */

    <> si.producer <c> in j;   /* producer thread */
    <c> si.consumer <>;        /* consumer thread */
}
```

The variables in MCL correspond to the objects that make up an application, such as channels, tasks, and jobs. As in C, these variables must be declared before they are used.

In Maruti, a *job* is a logical collection of threads that run with the same period. All *entry* functions in the application must be put *in* some job. The *init* statement sets the period for a particular job. In our case, the job *j* will run once every second.

A *task* is the runtime instantiation of an MPL module, just as in Unix a process is the runtime image of a program. Many tasks may be executed from the same module, each will run independently in the Maruti application. The MCL start command instantiates a task from a module. In our example, we instantiate one task from the module *simple* and pass it the initial data value of *27*.

We instantiate the threads for the *entry* and *service* functions inside a particular task, with particular input and output channels. In our example, the statement

```
<> si.producer <c> in j;   /* producer thread */
```

instantiates the *si.producer* thread in job j with no input channels and one output channel, *c*. Likewise, the statement

```
<c> si.consumer <>;        /* consumer thread */
```

instantiates the *si.consumer* thread with one input channel *c*, and no output channels. Service functions are not put in a job, but rather inherit the scheduling characteristics of the thread that is sending to their invoking channel.

The *integrator* checks to insure that the use of *producer* and *consumer* in the config file match the declarations in the program module.

Building and Running the Application

We can build the *simple* application by putting *simple.mpl* and *simple.cfg* in a directory, and running the *mbuild* command there:

```
% ls
simple.cfg     simple.mpl
% mbuild
mbuild: extracting module info from MCL file 'simple.cfg'
mbuild: creating obj subdirectory for output files.
mbuild: generating obj/simple-build.mk
mbuild: running make -f obj/simple-build.mk
 ...
```

Mbuild takes care of running the MPL compiler, the MCL integrator, as well as the analysis and binding programs needed to build the runnable Maruti application. By default, mbuild creates both a stand-alone binary that can be booted on the bare machine, and a Unix binary that runs in virtual real time from within the Unix development environment. These different versions of the runtime system are called *flavors.*

We can try out the simple application by running the *ux+x11* flavor from the command line:

```
% obj/simple.ux+x11
 <... startup messages ...>
consumer got 28
consumer got 29
consumer got 30
consumer got 31
consumer got 32
consumer got 33
consumer got 34
consumer got 35
consumer got 36
consumer got 37
 ...
application quit
```

The application boots up and outputs the consumer message once every second. We can exit the application by typing '*q*'.

2.2. Using the Graphics Library

Many Maruti programs will want to use the graphical screen as a monitor for an embedded system, producing oscilloscope or bar-graph style displays, or for animating a simulation or demonstration. Maruti provides a console graphics library as an integral part of the system to make the development of visually oriented applications simpler. Our next example application, *clock*, demonstrates the use of the graphics library as well as the use of multiple jobs to take advantage of Maruti's scheduling abilities.

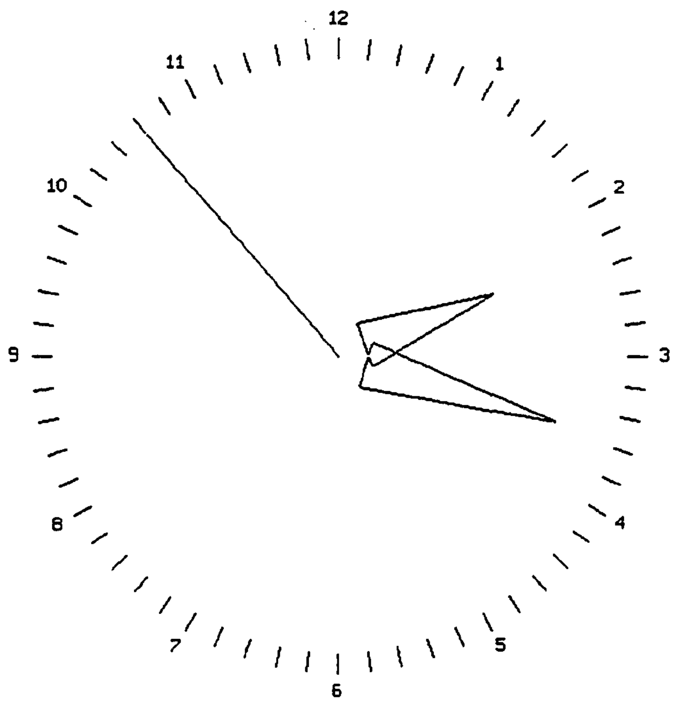


Figure 2.1: Dsiplay of clock example application

The *clock* application will display a circular clock face on the screen, with the hour, minute, and second hands moving as independent Maruti threads in different jobs. The clock screen is shown in Figure 2.1.

We will now go through the clock.mpl module and see how it works.

```
module clock;

   #include <maruti/mtime.h>
   #include <maruti/console.h>
   #include <math.h>

   #include "clock.h"

   #define CENTER_X      (CONSOLE_WIDTH/2)     /* useful constants */
   #define CENTER_Y      (CONSOLE_HEIGHT/2)

   void check_for_quitkey(void);             /* subroutines */
   void polar_point(int pos, int radius, int *x, int *y);
   void xor_triangle(int pos, int apex_radius,int color);
   void xor_ray(int pos, int color);

   int sec_pos, min_pos, hour_pos;           /* system state */
```

The first part of the module is much like any other ANSI-C program, with *# includes, # defines,* and function prototype declarations for subroutines to be used later in the program. Notice the two Maruti header files included: <maruti/mtime.h> contains declarations related to Maruti time management, and <maruti/console.h> contains declarations that define the graphics library interface. The "*clock.h*" header, which we'll see below, will contain definitions that customize the look of the clock face.

```
maruti_main()
  {
    int i, x1, y1, x2, y2, color;
    char num_str[4];
    mtime curtime;
    mdate curdate;

    /* initialize screen library, paint screen black */

    cons_graphics_init();
    cons_fill_area(0, 0, CONSOLE_WIDTH, CONSOLE_HEIGHT, BLACK);
```

The maruti_main function in the clock application draws the clock face display and initializes the system state - in our case, the positions of the three clock hands. Before drawing on the screen, the application must call cons_graphics_init, and initialize the contents of the screen. The call to cons_fill_area does this by filling the entire screen with the color BLACK.

```
    /* draw tick marks for clock face */

    for(i = 0; i < 60; i++) {
        polar_point(i*POS_PER_TICMARK, OUTER_RADIUS, &x1, &y1);
        polar_point(i*POS_PER_TICMARK, INNER_RADIUS, &x2, &y2);

        if(i % 5) color = GRAY;
        else color = WHITE;

        cons_draw_line(x1, y1, x2, y2, color);
    }
```

The step in initialization is to draw the tick marks for the clock face. There will be sixty tick lines drawn around the circle, one for each second. Every fifth tick mark will be WHITE to mark the hour positions, and the rest will be GRAY. The lines are drawn using the cons_draw_line library routine, which draws a one-pixel-wide line between two points in the desired color.

The location of the endpoints of our tick marks are calculated using a helper routine, polar_point (shown below), which calculates the cartesian coordinates for a given angle and radius. We conveniently adopt integer angle positions starting from 0 at the top, clockwise around up to 60*POS_PER_TICMARK back at the top again.

```
    /* draw numerals for clock face */

    for(i = 1; i <= 12; i++) {
      sprintf(num_str, "%d", i);
      polar_point(i*5*POS_PER_TICMARK, NUMBER_RADIUS, &x1, &y1);
      y1 -= 8; x1 -= strlen(num_str)*8 / 2;  /* center the string */
      cons_print(x1, y1, num_str, strlen(num_str), YELLOW);
    }
```

The numerals are placed on the clock face similarly to the tick marks. The cons_print graphics library function places text on the screen at a given position and color.

```
    /* initialize the hand positions to current time */

    maruti_get_current_time(&curtime);
    curdate = maruti_time_to_date(curtime);

    sec_pos = curdate.second * POS_PER_TICMARK;
    min_pos = curdate.minute * POS_PER_TICMARK +
           curdate.second * POS_PER_TICMARK / 60 ;
    hour_pos = (curdate.hour % 12) * 5 * POS_PER_TICMARK +
               curdate.minute * 5 * POS_PER_TICMARK / 60;

    return 0;
  }
```

The final part of the initialization is the calculation of the initial placement of the clock hands. The maruti_get_current_time|system call returns the current system time, given as a mtime structure. The system time is kept just as in Unix---as the number of seconds and microseconds since the *Epoch* time, defined as 00:00 GMT on January 1, 1970. The maruti_time_to_date library routine does the job of calculating the date and time-of-day from an mtime value.

```
  entry sec_hand()
  {
    static int erase = 0;

    if(erase) xor_ray(sec_pos, WHITE);
    else erase = 1;
```

```
      sec_pos = (sec_pos + POS_PER_TICMARK) % NUM_POSITIONS;
      xor_ray(sec_pos, WHITE);

      check_for_quitkey();
   }
```

The periodic function sec_hand will be run once per second. It erases the previously placed second-hand ray, calculates the new position and draws again there. The check_for_quitkey subroutine (shown below) will poll the keyboard and exit the application if a key is pressed.

```
 entry min_hand()
   {
      static int erase = 0;

      if(erase) xor_triangle(min_pos, MIN_RADIUS, MIN_COLOR);
      else erase = 1;

      min_pos = (min_pos + 1) % NUM_POSITIONS;
      xor_triangle(min_pos, MIN_RADIUS, MIN_COLOR);
   }

   entry hour_hand()
   {
      static int erase = 0;

      if(erase) xor_triangle(hour_pos, HOUR_RADIUS,HOUR_COLOR);
      else erase = 1;

      hour_pos = (hour_pos + 1) % NUM_POSITIONS;
      xor_triangle(hour_pos, HOUR_RADIUS, HOUR_COLOR);
   }
```

The min_hand and hour_hand periodic functions update their respective hand positions by one each time they are called. The second hand jumps forward one second each time it is called, but the minute and hour hands creep forward in smaller relative increments (rather than jumping forward once per minute or hour, which would not look right).

```
   void polar_point(int pos, int radius, int *x, int *y)
   {
        double angle = (2.0*M_PI/NUM_POSITIONS) * (NUM_POSITIONS-pos) +
M_PI/2;
```

```
        *x = CENTER_X + cos(angle) * radius;
        *y = CENTER_Y - sin(angle) * radius;
    }
```

Finally we come to the helper functions. The polar_point function converts from our convenient "positions" to real angles in radians, taking into account that radians start at the right and run counter-clockwise, whereas our positions start at the top and run clockwise. Given an angle in radians and a radius from the center, the x and y coordinates of the point are found by taking the cosine and sine of the angle. The final twist is that in cartesian coordinates, the y axis points up, whereas in screen coordinates it traditionally points down, so the y coordinate must be flipped around.

```
void xor_ray(int pos, int color)
    {
      int x, y;
      polar_point(pos, SEC_RADIUS, &x, &y);
      cons_xor_line(CENTER_X, CENTER_Y, x, y, color);
    }
    void xor_triangle(int pos, int apex_radius, int color)
    {
      int xb1, yb1, xb2, yb2, xp, yp;
      int bp1, bp2;
      bp1 = (pos + TRIANGLE_BASEL/2) % NUM_POSITIONS;
      bp2 = (pos - TRIANGLE_BASEL/2) % NUM_POSITIONS;

      polar_point(bp1, TRIANGLE_BASER, &xb1, &yb1);
      polar_point(bp2, TRIANGLE_BASER, &xb2, &yb2);
      polar_point(pos, apex_radius, &xp, &yp);

      cons_xor_line(xb1, yb1, xb2, yb2, color);  /* base of triangle */
      cons_xor_line(xb1, yb1, xp,  yp,  color);  /* first arm */
      cons_xor_line(xb2, yb2, xp,  yp,  color);  /* second arm */
    }
```

These graphic helper routines draw the line for the second hand and the triangle for the minute and hour hands. The **cons_xor_line** routine is similar to **cons_draw_line**, but exclusive-or's its pixels with the screen rather than just painting them. The xor technique is often used in graphics programming because it allows the drawing and erasing of objects without disturbing the background. When multiple objects overlap, the overlapping portions may become a strange color due to xor'ing, but you are guaranteed that when the objects are erased by xor'ing them a second time in the same location, whatever color was there before will be restored.

```
void check_for_quitkey(void)
  {
    console_event_t ev;

    if(cons_poll_event(&ev) != 0 && ev.device == EVENT_KEYBOARD
      && ev.keycode == KEY_SPACE)
       quit();
  }
```

The final helper routine polls the console keyboard for events, and quits the application if the space bar is pressed. The cons_poll_eventlsystem call reports both key press and key release events, and reports a scan-code rather than an ASCII value. This interface is rather low level, but allows the application complete access to the up/down state of every key on the keyboard.

This completes the clock.mpl module. The clock.cfg config file follows:

```
#include "clock.h"

application clock {

  job sec_job;     init sec_job:  period SEC_PERIOD s;    /* jobs */
  job min_job;     init min_job:  period MIN_PERIOD s;
  job hour_job;    init hour_job: period HOUR_PERIOD s;

  task ct;         start ct: clock;              /* task */

  <> ct.sec_hand  <> in sec_job;                /* threads */
  <> ct.min_hand  <> in min_job;
  <> ct.hour_hand <> in hour_job;
}
```

Notice that the config file can include header files just like the MPL module can. This allows the programmer to put configuration-related constants in one header and use them in both the config file and the application modules.

The clock config simply creates one task, plus a job for each hand of the clock. The periods are defined in "clock.h":

```
#define INNER_RADIUS    235
#define OUTER_RADIUS    (INNER_RADIUS+15)
#define NUMBER_RADIUS   (OUTER_RADIUS+15)

#define TRIANGLE_BASER  30
```

```
#define TRIANGLE_BASEL  50

#define SEC_RADIUS     INNER_RADIUS

#define MIN_COLOR      YELLOW
#define MIN_RADIUS     (SEC_RADIUS-50)

#define HOUR_COLOR     GREEN
#define HOUR_RADIUS    (MIN_RADIUS-50)

#define NUM_POSITIONS  240
#define POS_PER_TICMARK (NUM_POSITIONS/60)

#define SEC_PERIOD   1                              /* jumps 1 tickmark/sec */
#define MIN_PERIOD   (60/POS_PER_TICMARK) /* creeps 1 tickmark/min */
#define HOUR_PERIOD (3600/5/POS_PER_TICMARK)  /* creeps 5 tickmarks/hr */
```

First, a number of constants describing the visual appearance of the clock face are defined. These can be modified to taste.

Second, the timing characteristics of the program are given. The key parameter is NUM_POSITIONS, which gives the number of positions which the minute and hour hands take around the clock face. The larger this number, the smaller the distance the hands move each time, and the more frequently their jobs are executed. The minute hand must move through all 60 tick marks once every hour, and the hour hand 5 tick marks each hour. With NUM_POSITIONS set to 240, each hand moves four times for each tick mark on the face of the clock, which works out to one move every 15 seconds for the minute hand, and one move every 180 seconds for the hour hand.

Chapter 3

MPL / C References

Maruti Programming Language (MPL) is a simple extension to ANSI C to support modules, synchronization and communications primitives, and shared memory variables. MPL adds some restrictions that enable analysis of the CPU and memory requirements of the program. This chapter will define the MPL-specific features that differ from ANSI C.

3.1. EBNF Syntax Notation

In this manual, syntax is given in Extended Backus-Naur Formalism (EBNF). In this notation:

- literal strings are quoted, e.g. 'module'.
- other terminal symbols are bracketed, e.g. <module-name>.
- X|Y denotes alternatives.
- {Z} denotes zero-or-more.
- {X} denotes zero-or-one.

3.2. MPL Modules

The *module* is the compilation unit in MPL. It is presented to the MPL compiler as one file, but may contain normal C #include directives so that the parts of the module can be kept as distinct files. The MPL compiler generates a binary object file for the module, as well as a *partial EU graph file* for the module, which contains information about the module needed by the Maruti analysis tools.

At runtime, each MPL module is mapped to a Maruti *task*, which logically runs in its own address space. Communication between tasks is through *channels* or *shared blocks*. Each task can contain multiple *threads* of execution, each thread corresponding to an *entry* or *service* function of MPL.

Each module starts with the module name declaration:

```
module_name_spec ::= 'module' <module-name>.
```

3.3. Module Initialization

When the task corresponding to a module is loaded, the Maruti runtime system executes a non-real-time initializer function provided by the programmer. The initializer is a normal C function, but it must be present in every module. It is declared as:

```
int maruti_main(int argc, char **argv);
```

The job of this function is to initialize the state of the task, taking any parameter values into account. If the initializer returns 0, then the task is considered successfully loaded, otherwise the load fails. The initializer thread can not send or receive messages on Maruti channels.

3.4. Entry Functions

Maruti *entry* functions occur as top-level definitions in the MPL source file, similar in syntax to normal C function definitions.

```
entry-function ::= 'entry' <entry-name> (' ')' entry-function-body.
entry-function-body :: = channel-declaration-list c-function-body.
```

Entry functions serve as the top-level function of a Maruti thread which is invoked repeatedly with a period as specified externally, in the MCL configuration. Multiple instances of the entry thread can be active in a single task at runtime, so care must be taken to protect accesses to shared data with a region or local_region construct.

3.5. Service Functions

Maruti *service* functions also occur as top-level definitions in the MPL source file.

```
service-function ::= 'service' <service-name>
                         '('<in-channel-name>':'<type_specifier>','<msg-ptr-name>')'
                      service-function-body.
service-function-body :: = channel-declaration-list c-function-body.
```

Services are declared with the initiating channel and pointer to a message buffer. A service thread is invoked whenever a message on the channel has been received, thus it inherits the scheduling characteristics of the sender to the channel. Multiple instances of the service may be active in a single task at the same time, servicing messages from different senders, so care must be taken to protect accesses to shared data with a region or local_region construct.

The receipt of the invoking message into private storage is automatic, and the service function is called with a pointer to the message buffer. Fore example, given the service declaration:

```
service consumer(inch: ch_type, msg) { ... }
```

The service is actually invoked as if it were a C function declared:

```
void consumer(ch_type *msg) { ... }
```

3.6. MPL Channels

In Maruti, *channels* are one-way, typed, communications paths whose traffic patterns are analyzed and scheduled by the system. The channel end-points are declared as part of the *entry* or *service* functions which take part in the communication. The endpoints are connected in the MCL configuration for the application.

The syntax of MPL channels is similar to a C variable declaration:

```
channel-declaration-list ::= [ channel-decl { channel-decl }].
    channel-declaration ::= channel-type channel { ',' channel } ';'.

     channel-type ::= 'out' | 'in' | 'in_first' | 'in_last'.
    channel ::= <channel-name> ':' type-specifier.
```

A channel endpoint declaration will normally be either an out endpoint or an in endpoint, used in the sending thread and receiving thread, respectively. There are two special variants of in endpoint, in_first and in_last, which denote asynchronous channels in which the communications will not be scheduled, and the input buffers are allowed to overflow. For in_first channels, the first messages received will be retained and the rest dropped, for in_last the most recent messages will be retained and older messages overwritten.

3.7. Communication Primitives

The message passing primitives appear as normal C function calls, but they are built in primitives of the MPL compiler, and their use is recorded so that the communications on the channel can be analyzed.

The three primitives each take a channel name and a pointer to a message buffer. Their declarations would look something like this:

```
void send     (out ch_name, ch_type* message_ptr);
void receive  (in  ch_name, ch_type* message_ptr);
int  optreceive(in  ch_name, ch_type* message_ptr);
```

There are two variants of the receive primitive. A normal receive is used in most cases, and it raises an exception if there is no message delivered at the time it is executed. Normally the Maruti scheduler will arrange things so that this is never happens. When messages might not be present when the receiver is run, as when threads are communicating asynchronously with in_first and in_last channels, or when the sender sometimes will not send the message due to run time conditions, an optreceive must be used. The optreceive variant checks if a message is present, and receives it if so. It returns 1 if a message was delivered, or 0 if no message was delivered.

3.8. Critical Regions

Mutual exclusion is often necessary to prevent the corruption of data structures modified and accessed by concurrent threads. In Maruti, the region statement delineates a critical region.

region-statement ::= ('region'|'local_region') <region-name> c-statement.

The local region variant is used within a task, usually to serialize multiple thread access to data structures. The region variant is global to the application, and is used to serialize access to shared buffers and other application-defined resources, as specified in the MCL configuration for the application.

3.9. Shared Buffers

Finally, MPL adds shared buffers to the C language. Shared buffers declarations are similar in syntax to typedef declarations:

shared-buffer-decl ::= 'shared' <type-specifier> <shared-buffer-name>.

The shared buffer declaration is effectively a pointer declaration. For example:

```
shared some_type shared_buffer;
```

is treated as if it were a declaration of the form:

```
some_type *shared_buffer = &some_buffer;
```

The MPL specification for the application determines which tasks share each shared memory area. The runtime system takes care of allocating memory for the shared buffers, and initializing the buffer pointers. The MPL program can at all times dereference the pointer.

3.10. Restrictions to ANSI C in MPL

The Maruti real-time scheduling methodology requires that the tools be able to analyze the control flow and stack usage of the MPL programs, and that synchronization points be well known. Thus the following restrictions to ANSI C must be followed by the MPL programmer:

- No receive primitives are allowed within either loops or conditionals.
- No region construct are allowed within either loops or conditionals.
- No send primitive within a loop.
- Direct or indirect recursion is not allowed.
- Function calls via function pointers should not be used.

Chapter 4

MCL Reference

Maruti Configuration Language (MCL) is used to specify how individual program modules are to be connected together to form an application and to specify the details of the hardware platform on which the application is to be executed.

MCL is an interpreted C-like language. The MCL processor is called the *integrator*. The integrator interprets the instructions of the MCL program, instantiating and connecting the components of the application, checking for type correctness as it goes, and outputs the application graph and all allocation and scheduling constraints for further processing by other Maruti tools.

4.1. Top-level Declarations

Like a C program, an MCL configuration file is composed of a number of top-level declarations. The C preprocessor is invoked first, so the configuration file may contain #include and #define directives to make the configuration very customizable.

```
configuration ::= {toplevel-declaration}.
  toplevel-declaration ::= variable-declaration | system-declaration
                 block-declaration | application-declaration.
```

The declarations may occur in any order---they do not have to be defined before used. The four types of top level declaration are described in more detail below.

4.1.1. The Application Declaration

```
application-declaration ::= 'application' <application-name>
                 '{' {instruction} '}'.
```

Like the main function of C, the application declaration is where the integrator will begin execution of the configuration directives. Only one application may be declared in the configuration.

4.1.2. The System Declaration

```
  system-declaration ::= 'system' <system-name>
                 '{' {node-declaration} '}'.

  node-declaration ::= 'node' <variable-name> ['with' attributes].
  attributes ::= attribute {',' attribute}.

  attribute ::= <symbol> ['=' <integer> | '=' <symbol> | '=' <string>].
```

Like the application declaration, the system declaration can occur at most once in a configuration. It is not needed for single-node operation. The system declaration names the nodes that an application will run on, and specifies attributes for them. For example:

```
system hdw {
   node northstar with address = "{0x00,0x60,0x8c,0xb1,0xfb,0xc6}", master;
   node raduga    with address = "{0x00,0x60,0x8c,0xb1,0xf6,0x67}";
}
```

The integrator does not assign any meaning to the attributes declared for the nodes, it just passes the information along. However, the Maruti binder does require the addresslattribute for each node, which specifies the node's ethernet address, and the master attribute on only one node, to specify which node will be the boot and time master. The Maruti/Virtual environment further requires that the node <variable-name> correspond to the hostname of the node in the testbed environment.

4.1.3. Block Declarations

```
block-declaration ::= 'block' <block-name> '(' [block-parameters] ')'
                      block-parameter-channels
                        '{' {instruction} '}'.
```

A block is something like a function in C. When a block is declared, it may be called by any other block, except that no self-recursion is allowed. A block can not be declared inside another block. A block is called by giving its name and parameters. There are 2 kinds of parameters: classical parameters and channel parameters.

```
block-parameters ::= parameter { ',' parameter }.
parameter ::= ['var'] <parameter-name> ['[]'] [':' type].
```

Classical parameters are like function parameters in C or Pascal. They can be passed by value or by variable (var for variable passing). Arrays may also be be given as var parameters. The type of the parameter must be given for the first parameter. It may be omitted for following parameters: the integrator will assume that the parameter with no given type has the same type as the previous parameter.

```
block-parameter-channels ::= { ('in'|'out') channel-names ';' }.
channel-names ::= channel { ',' channel }.
channel ::= <channel-name> [ '[' <integer> ']' ].
```

The channel parameters decribe the inputs and outputs of the block. The in and out keywords do not have exactly the same meaning has in MPL: they only show which channels are connected at the left and which are connected at the right of the block call (see connection below). The communication type of the channel in_first, in_last, or synchronous) and the type of the messages on the channel are determined by the connections of the channels to the tasks.

When there is an array of channel parameters, the connections will occur in ascending order. For example:

```
block foo()
  in ch[3];
  { ... }
```

```
application bar {
  channel a[3];
  <a[0..2]> foo() <>; /* a[0]->ch[0], a[1]->ch[2], a[2]->ch[2] */
   ...
}
```

4.1.4. Variable Declarations

variable-declaration ::= type variable-names ';'.

type ::= 'float' | 'int' | 'string' | 'time' | 'channel'
| 'task' | 'job' | 'node' | 'shared' | 'region'.

variable-names ::= variable { ',' variable }.
variable ::= <variable-name> ['[' <integer> ']'] ['[' <integer> ']'].

Variables may be declared globally at the top-level, or locally in a block. Global variables can be accessed in all blocks, while local variables can only be accessed in the block where they are declared. A local variable (or a parameter) may be declared with the same name as a global variable. In this case only the local variable (or the parameter) can be accessed in the block.

The order of the variable declarations does not matter. For example:

```
block foo()
{
  i = 4s + 5mn;        /* correct */
  time i;
}
```

Arrays may be declared. As in C, the array indicies are numbered from 0 to size-of-array less 1. Arrays of 1 or 2 dimensions are accepted. For example:

```
block foo()
  {
    string s[10];
    s[5] = "a string";        /* correct */
    s[0] = s[5] + " foo";       /* correct */
    s[10] = "";             /* incorrect: out of array limits */
  }
```

4.2. Instructions

The MCL integrator interprets a number of instructions that express the way an application is to be built up from components. The different instructions are explained below.

```
instruction ::= variable-declaration
             | task-initialization
             | job-initialization
             | connect-declaration
             | link-intruction
| allocation-instruction
 | expression ';'
| print-instruction
| compound-instruction
  | '{' {instruction} '}'.
```

4.2.1. Compound Instructions

```
compound-instruction ::=
              'if' '(' test-expression ')' intruction
             | 'if' '(' test-expression ')' intruction 'else' instruction
              | 'do' instruction 'while' '(' test-expression ')' ';'
             | 'while' '(' test-expression ')' instruction
              | 'for' '(' expression ';' test-expression ';' expression ')'
                 instruction.
   test-expression ::= expression.
```

The meaning of these constructs is the same as in the C language. The test-expression should evaluate to an integer, where 0 means false, and all other values mean true.

4.2.2. Tasks

```
task-initialization ::= 'start' names ':' <module-name> [module-parms]
                  [instantiation] [task-allocation] ';'

module-parms ::= '(' [module-parameter-list] ')'.
module-parameter-list ::= expression {',' expression}.

instantiation ::= 'with' <symbol> '=' constant
                 {',' <symbol> '=' constant }.

task-allocation ::= 'on' expression.
```

A variable of type task must be initialized before it can be used. This initialization consists of giving the name of a module: the task will be an instantiation of this module. Module parameters may be given: after evaluation they will be given to the initializer thread of the module. The initializations during the loading of an application will take place in exactly the same order as thay are found by the intergrator during the execution of the configuration.

All the shared buffers and the global regions of the module must be instantiated using the with clause: the corresponding shared or region variables must be given.

The on clause may be used to force allocation of the task on a particular node.

4.2.3. Job Initialization

```
job-initialization ::= 'init' names ':' timing-job ';'.
timing-job ::= { 'period' expression }.
```

A variable of type job must be initialized before it can be used. The job will refer to a collection of threads with the same period.

4.2.4. Connections

```
connect-declaration ::= chan-list connect-name chan-list
                        [in-job] {timing-service} [task-alloc]';'.
   chan-list ::= '<' [names] '>'.
   connect-name ::= <task-name> ['['expression']'] '.' <routine-name>
              | <block-name> '(' [expression {',' expression}] ')'.
   in-job ::= 'in' constant.
   timing-service ::= ( 'ready' expression | 'deadline' expression ).
```

There are two types of connections: routine connections and block connections. In both cases the inputs are connected (or mapped) to the channels declared at the left of the connection and the outputs at the right. The number of input (or ouput) channels must be the same as in the definition of the routine (or the block). The mapping is done following the order of this definition.

In a routine connection the inputs and ouputs of an entry or a service of a task are connected to channels. This connection creates a new instance of a service if the routine was a service, otherwise it creates the only instance of an entry. An entry can not be connected many times.

For an entry connection a job name must be given, the entry will be a part of this job. For a service, a job can not be declared: the job of the service is implicitly given by the connection: the first input channel of a service is the triggering channel of the service. The job of the service is the same as the job of the origine of the triggering channel.

A timing characterization may only be given to a routine connection.

In a block connection the input and output channels of the block are mapped to the given channels. A mapping is also done for all the block parameters, following the order in the block definition. The number of parameters must be the same as in this definition, and all the types must be coherent.

4.2.5. Allocation Instructions

```
allocation-instruction ::= 'separate' '(' names ')' ';'
                         | 'together' '(' names ')' ';'.
```

A separate instruction is a command to the allocator to keep the tasks on different nodes in the final system. A together instruction specifies that all tasks must be allocated to the samenode.

4.2.6. Link Instruction

```
link-intruction ::= 'link' expression 'to' expression ';'.
```

In a few cases the connections are not sufficient to describe a communication graph with the structure of the blocks. In these cases a link instruction may be used.

A link between two channels means that the two channels are the same.

Example: if we want to connect directly an input and an output channel of a block a link must be used.

```
block foo()
  in in_channel;
  out out_channel;

  {
    link in_channel to out_channel;
  }
```

4.2.7. Print Instruction

print-instruction ::= 'print (' expression {',' expression} ');'.

The print instruction outputs messages to the standard output during integration. This instruction can be used for the debugging of a configuration file. Any string, number, or time may be printed. A newline is added at the end.

4.3. Expressions

Expressions in MCL are very similar to C expressions:

```
      expression ::= expression '=' expression
      | expression '||' expression
      | expression '&&' expression
      | expression '==' expression
      | expression '!=' expression
      | expression '<' expression
      | expression '>' expression
      | expression '<=' expression
      | expression '>=' expression
      |  expression  ('d'|'h'|'mn'|'s'|'ms'|'us')
      | expression '+' expression
      | expression '-' expression
      | expression '*' expression
      | expression '/' expression
      | expression '%' expression
      | '!' expression
      | '(' expression ')'
      | expression '++'
      | expression '--'
      | constant

  constant ::= <symbol> [ '[' expression ']' ] [ '[' expression ']' ]
      | <symbol> '[' expression '..' expression ']' [ '[' expression ']' ]
      | <symbol> '[' expression ']' '[' expression '..' expression ']'
      | <integer>
      | <double>
      | <string>.
```

In addition to the usual C expressions, MCL supports time unit expressions, for example, '3 s + 500 ms' is a time expression that evaluates to 3.5 seconds.

Also, MCL supports array range notation as a shorthand for lists. For example, the expression 'c[2..4]' is shorthand for c[2], c[3], c[4]'. This notation is most often used for passing arrays of channel values to blocks or in connection instructions.

CHAPTER 5

Maruti Runtime System Reference

The Maruti runtime system is bound together with the application binary files by the *mbind* utility. Only those parts of the runtime needed by the application are linked in. There are several versions of the runtime system available depending on the environment in which the application will be run. For example, there are two different versions of the core library: a stand-alone version that can boot directly on bare hardware, and a Unix version that runs as a user-level process under Unix, providing virtual-time execution and access to debugging tools.

The set of library versions that an application links with are called *flavors*. Flavors are specified by the programmer as strings of library names separated by a '+', for example, 'ux+x11'.

5.1. Core Library Reference

#include <maruti/maruti-core.h>

The Maruti core library implements the scheduling, thread and memory management, and network communication subsystem. It provides primitives for applications to send and receive messages, insert preemption points, manipulate the schedule (via calendars), and do time and date calculations. There are currently two flavors of the core library:

- **sa** -- The Maruti/Standalone core library. Applications linked with this flavor can be booted directly (by the NetBSD boot blocks). It includes the distributed operation support, based on the 3Com 3c507 Etherlink/16 adapter.
- **ux** -- The Maruti/Virtual debugging core library. Applications linked with this flavor are run as normal Unix processes from the NetBSD command line. It includes a virtual-time scheduler and debugging monitor (described below) and implements distributed operation using normal Unix TCP/IP networking facilities.

5.1.1. MPL Built-in Primitives

void maruti_eu(void)

The maruti_eu primitive inserts a Maruti EU break into the program at the location of the call. It is not normally used explicitly in an application, as the system tools put EU breaks where necessary for synchronization. It is useful, however, for breaking up long-running EUs -- the maruti_eu then serves as a possible preemption point.

void send(out ch_name, ch_type* message_ptr)
void receive(in ch_name, ch_type* message_ptr)
int optreceive(in ch_name, ch_type* message_ptr)

The communications primitives are documented in section 3.7. in the MCL Reference Chapter.

5.1.2. Calendar Switching

int maruti_calendar_activate(int calendar_num, mtime switch_time, mtime offset_time)
void maruti_calendar_deactivate(int calendar_num, mtime switch_time)

Maruti calendars may be activated and deactivated (*switched* on or off) at any time. The switch_time is the time at which the de/activation should take place. The switch can occur at any point in the future, and the switch requests can come out of order with respect to the switch time. Requests with the same switch time are executed in the order of the requests.

Calendars can be activated with a particular offset_time, which is the relative position within the calendar to start executing at the switch time. The offset time will normally be zero, but can be any relative time up to the lcm time of the calendar.

The runtime system does not check the feasibility of the combined schedules represented by the calendars - that should be done offline.

5.1.3. Calendar Modification

void maruti_calendar_create(int calendar_num, int num_entries, mtime lcm_time) void maruti_calendar_delete(int calendar_num)

Calendars are normally created offline and compiled into the Maruti application, but it is possible to create new calendars at runtime. The application is responsible for insuring that the generated schedules are feasible.

When a calendar is created, the maximum number of entries it will contain must be specified, as well as the lcm_time, which is the period of the calendar as a whole. At the end of its period, the calendar will wrap around and begin executing from the beginning again.

```
 typedef struct calendar_s {
   entry_t *entries;
   int num_entries;
   mtime lcm_time;
   mtime base_time;
   entry_t *cur;        /* cur-entries is the current offset */
    <...>
} calendar_t;
typedef struct {
   int eu_thread, eu_id;
   mtime eu_start, eu_deadline;
   int eu_type;
#     define EU_EMPTY      0     /* empty EU slot */
#     define EU_PERIODIC   1     /* periodic EU */
    <...>
} entry_t;
```

```
void maruti_calendar_get_header(int calendar_num, calendar_t *calendarp)
void maruti_calendar_get_entry(int calendar_num, int entry_num, entry_t *entryp)
void maruti_calendar_set_entry(int calendar_num, int entry_num, entry_t entry)
```

The maruti_calendar_set_entry call is used to populate new calendars. It can overwrite any entry in any inactive calendar. The entry eu_start and eu_deadline times are the earliest start time and latest end time, respectively. The eu_id serves to identify the eu when tracing or reporting timing results.

The maruti_calendar_get_header and maruti_calendar_get_entry calls can be used to query the contents of a calendar. These are useful when 'cloning' an existing calendar into a new calendar, perhaps with modifications.

5.1.4. Date and Time Manipulation

```
#include <maruti/mtime.h>
```

The Maruti core library provides routines and macros for simple time and date calculations.

```
typedef struct {
  long seconds;
  long microseconds;
} mtime;

#define time_cmp(a,b)            /* like strcmp, 0 if eq, lt 0 if a < b, etc */
#define time_add(a,b)            /* a += b */
#define time_sub(a,b)            /* a -= b */
#define time_add_scalar(t, s)    /* t += s (s is an int, in microseconds) */
#define time_sub_scalar(t, s)    /* t -= s (s is an int, in microseconds) */
#define time_mul_scalar(t, s)    /* t *= s (s is an int) */
#define time_div_scalar(t, s)    /* t /= s (s is an int) */
```

The mtime type is the basic Maruti time structure. A number of convenience macros for arithmetic on mtime values are provided. Two mtime values may be compared, added, or subtracted. In addition, an integer time in microseconds may be added to and subtracted from an mtime value, and mtime values may be multiplied or divided by integer scaling factors.

Note: The microseconds field is always in the range 0 to 999999, and the time represented by an mtime value is always the number of seconds plus the number of microseconds. These rules hold even for negative mtime values, which can arise when subtracting mtimes. Thus the mtime representation for the time -1.3 seconds is { -2, 700000 }.

```
void maruti_get_current_time(mtime *curtime)
```

The current system time is returned by maruti_get_current_time. Maruti, like Unix, represents absolute time as the number of seconds and microseconds since the *Epoch* time, defined as 00:00 GMT on January 1, 1970.

```
typedef struct {
    short year;                 /* year - 1900 */
    short month;                /* month (0..11) */
    short wday;                 /* day of week  (0..6) */
    short mday;                 /* day of month (1..31) */
    short yday;                 /* day of year (0..365) */
    short second, minute;       /* 0..59 */
    short hour;                 /* 0..23 */
    int  microsecond;           /* 0..999999 */
} mdate;
```

```
mtime maruti_date_to_time(mdate d)
mdate maruti_time_to_date(mtime t)

mtime maruti_gmtdate_to_time(mdate d)
mdate maruti_time_to_gmtdate(mtime t)

int maruti_set_gmtoff(int gmtoff)
int maruti_get_gmtoff(int *gmtoffp)
```

Applications will often want to view the time as something more convenient than the number of seconds since the Epoch. The Maruti mdate type denotes a time expressed as a date plus a time of day. The functions maruti_time_to_gmtdate and maruti_gmtdate_to_time convert between mtime and mdate values using the GMT timezone. The functions maruti_time_to_date and maruti_date_to_time convert using the local offset from GMT.

The local timezone used in these conversions is initially set by the runtime system, but may be changed by the application. The timezone is expressed as an offset from GMT in seconds. For example the U.S. timezone EST is 5 hours behind GMT, or -18000 seconds offset.

Note: Maruti does not at this time attempt to handle leap seconds or automatically switching the local timezone to account for daylight savings times. The cost of providing these features in code and table space was deemed prohibitive.

5.1.5. Miscellaneous Functions

```
void quit(int exitcode)
```

The quit call terminates the application. The exit code is not usually relevant in an embedded system, but will be returned to the environment where that makes sense (such as in the Unix debugging environment).

5.2. Console Library Reference

#include <maruti/console.h>

The Maruti console graphics library provides access to the console device, including the keyboard and speaker, but most importantly the graphical display. The graphics library includes support for placing text anywhere on the screen, simple 2d geometry primitives suitable for generating line and bar graphs, and includes optimized routines for moving bitmaps without flicker, for animated simulations. There are currently three flavors of the graphics library implemented:

- **et4k** -- This flavor supports Super VGA graphics cards based on the Tseng Labs ET4000 chip and its accelerated descendents, like the ET4000/W32. The et4k flavor runs the screen at a resolution of 1024x768 in 256 color mode.

- **vga16** -- This flavor supports all standard VGA graphics cards, running the screen at a resolution of 640x480, in 16 color banked mode.

- **x11** -- This flavor works with the {\bf ux} core flavor, displaying the Maruti screen in an X11 window under Unix.

5.2.1. Screen Colors

The Maruti console graphics library supports the following colors, defined in <maruti/console.h>:

```
#define BLACK           0
#define DARK_BLUE       1
#define DARK_GREEN      2
#define DARK_CYAN       3
#define DARK_RED        4
#define DARK_VIOLET     5
#define DARK_YELLOW     6
#define DARK_WHITE      7
#define BROWN           8
#define BLUE            9
#define GREEN           10
#define CYAN            11
#define RED             12
#define VIOLET          13
#define YELLOW          14
#define WHITE           15
/* aliases */
#define GREY            DARK_WHITE
#define GRAY            DARK_WHITE
```

The maximum screen size supported is also defined:

```
#define CONSOLE_WIDTH       1024
#define CONSOLE_HEIGHT      768
```

5.2.2. Graphics Functions

```
void cons_graphics_init(void)
```

The cons_graphics_init function must be called before any other graphics functions, usually from the maruti_main function of the application's screen driver task.

```
void cons_fill_area(int x, int y, int width, int height, int color)
void cons_xor_area(int x, int y, int width, int height, int color)
```

These functions paint an area of the screen, specified by its upper-left coordinates (x, y), and its width and height, in the given color. The cons_fill_area variant overwrites the previous contents of that area of the screen,while cons_xor_area exclusive-or's the screen contents with the specified color.

```
function cons_draw_pixel(int x, int y, int color)
function cons_xor_pixel(int x, int y, int color)
```

These functions draw and xor, respectively, a single pixel at (x, y) in the specified color.

```
void cons_draw_line(int x1, int y1, int x2, int y2, int color)
void cons_xor_line(int x1, int y1, int x2, int y2, int color)
```

These functions draw and xor, respectively, a single-pixel width line from coordinates (x1, y1) to (x2, y2) in the specified color.

```
void cons_draw_bitmap(int x, int y, int width, int height,
                void *bitmap, int color)
void cons_xor_bitmap(int x, int y, int width, int height,
                void *bitmap, int color)
```

These functions draw and xor, respectively, a width-by-height sized bitmap onto the screen in the specified color, with its upper-left corner at (x, y). The bitmap is in standard X bitmap format, with eight pixels per byte, and an even multiple of eight pixels per scan line.

```
void cons_move_bitmap(int x1, int y1, int x2, int y2, int width, int height,
                  void *bitmap, int color)
  void cons_xor_move_bitmap(int x1, int y1, int x2, int y2, int width, int height,
                  void *bitmap, int color)
```

These functions optimize the erasing and redrawing of a bitmap by combining the operations into one loop, modifying one scan-line at a time. This optimization eliminates the flicker that can occur when erasing the entire bitmap then redrawing it, making animations more effective.

The call cons_move_bitmap(x1,y1,x2,y2,w,h,b,c) is equivalent to the sequence:

```
cons_draw_bitmap(x1,x2,w,h,b,BLACK);
cons_draw_bitmap(x2,y2,w,h,b,c);
```

The call cons_xor_move_bitmap(x1,y1,x2,y2,w,h,b,c) is equivalent to the sequence:

```
cons_xor_bitmap(x1,y1,w,h,b,c);
cons_xor_bitmap(x2,y2,w,h,b,c);
```

```
void cons_puts(int x, int y, int color, char *string)
void cons_xor_puts(int x, int y, int color, char *string)
```

These functions draw and xor, respectively, a text string at (x, y) in the specified color.

5.2.3. Keyboard and Speaker functions

```
typedef struct
{
  unsigned char device;     /* just keyboard works for now */
#    define EVENT_OTHER        0
#    define EVENT_KEYBOARD     2
  unsigned char keycode;
} console_event_t;
```

```
int cons_poll_event(console_event_t *event)
```

The cons_poll_event call returns 1 if a console event has occurred, 0 otherwise. There there is a pending console event, the event structure is filled in. The device field is set to EVENT_KEYBOARD and the keycode field is set to the scan code of the key that was pressed. The list of scan codes is in <maruti/keycodes.h>.

```
void cons_start_beep(int pitch)
void cons_stop_beep(void)
```

The console speaker can be turned on and off with these functions. The cons_start_beep call programs the speaker to sound at a particular frequency, in hertz, and cons_stop_beep turns it
off.

5.3. Maruti/Virtual Monitor

The *ux* flavor of the Maruti core library includes some basic debugging facilities called the Maruti *monitor*. While an application compiled with *ux* is running, aspects of its execution can be controlled from the Unix tty (which will be distinct from the console keyboard device). The monitor provides the following facilities:

Tracing scheduler actions. The user can independently toggle the tracing of elemental unit executions, calendar wrap-around events, and calendar-switch events.

Single-stepping calendars or elemental units. The user can toggle single stepping through each elemental unit execution, or a whole calendar's execution.

Controlling virtual-time execution speed. The user can control the speed of the application in two ways. First, the user can toggle as-soon-as-possible execution of elemental units, called *asap mode*. Second, the user can set the speed at which virtual time advances relative to real clock time.

Both single-keystroke and command-line operation. All monitor switches may be toggled with a single keystroke while the application continues running. Also, the user can enter a command-line mode in which various parts of the system state may be queried and modified.

5.3.1. Controlling Virtual Time

The Maruti monitor contains a user-settable *speed* variable which determines the rate at which virtual time advances relative to the actual clock time.

The speed may be set to any floating point value greater than zero. Thus virtual speed may be set to run, for example, five times faster than clock time (speed = 5) or at four times slower (speed = 0.25). The speed is logically limited on the side by the utilization of the CPU. The execution of application code can not be sped up, only the idle time between executions.

Idle time can be eliminated completely by turning on as-soon-as-possible scheduling of elemental unit (*asap-mode*). In asap-mode the virtual time is advanced to the start time of the next elemental unit as soon as the previous one completes, resulting in the execution of all EUs in immediate succession. Asap-mode is separate from the *speed* variable -- it can be toggled independently, and when turned off, scheduling continues at the previously set speed.

5.3.2. Single-Keystroke Operation

The following keys are active from the Unix tty session (**not** the console keyboard) while the application is running:

?	shows the list of keystrokes and current values for the toggle switches.
a	toggle as-soon-as-possible mode.
e	toggle elemental unit tracing.
c	toggle calendar tracing.
x	toggle calendar-switch tracing.
s	toggle elemental unit single-stepping.
S	toggle calendar single-stepping.
q	quit application completely.
<ESC>	stop application and enter command-line mode.

5.3.3. Command-line Operation

The following commands are available from command line mode. At this time, command-line mode is a just a framework with just a few commands. More commands to query and set the system state are envisioned for future releases.

help
Get a list of command-line mode commands.

quit
Quit the application completely.

vars
Show all user-settable monitor variables and their values.

speed <value>
Set the virtual-time speed to **value.** The value can be any floating point value greater than zero.

cstep [on|off]
Set or toggle calendar single-stepping.

estep [on|off]
Set or toggle eu single-stepping.

ctrace [on|off]
Set or toggle calendar tracing.

etrace [on|off]
Set or toggle eu tracing.

strace [on|off]
Set or toggle calendar switch tracing.

Chapter 6

Maruti Tool Reference

6.1. Maruti Builder

The mbuild program automates the process of building a runnable Maruti application. This involves building the constituent application binaries, integrating and scheduling the application, and binding the application with the desired Maruti runtime flavor.

Mbuild is normally run in the directory in which the application config file and constituent module source files are located. It will automatically find the config file by its .cfg extension, read it, and generate a makefile that builds what modules it finds used there, then calls the other Maruti tools. Mbuild works by creating an obj subdirectory, and putting all output files there.

If there is more than one config file in the current directory, the desired file must be specified with the -f <config file> option.

The user may optionally customize the mbuild actions by providing an Mbuild.inc file in the current directory. This file will be included into the makefile generated by mbuild. In addition to providing additional build targets and dependency lines, the user may set some variables to modify the mbuild actions themselves:

FLAVORS Default: **ux+x11 ux+et4k sa+et4k** The list of runtime flavors with which to link the application.

MPC Default: mpc. The program executed to compile MPL programs. Not normally modified by users.

MPC_FLAGS Default: **<empty>**. Supplemental flags for the MPL compiler. Most GCC flags will work here. Most often the user will want to customize the include directories with -I <dir>.

CFG Default: cfg. The program executed to interpret the MCL config file and integrate the application. Not normally modified by users.

CFG_FLAGS Default: **<empty>**. Supplemental flags for the MPC integrator. Not normally modified by users.

ALLOCATOR Default: allocator. The program executed to allocate and schedule the application. Not normally modified by users.

ALLOCATOR_FLAGS Default: -p 1. Flags for the Allocator. See section 6.4. on the Allocator below for more details.

MBIND Default: mbind. The program executed for binding the application and runtime system. Not normally modified by the users.

MBIND_FLAGS Default:**<empty>**. Flags for the Mbind program. Not normally modified by users.

6.2. MPL/C Compiler

The MPL/C compiler (mpc) consists of a modified gcc plus some attendant scripts to post-process the compiler output. It generates a .o file for a module, plus a .eul file containing a partial elemental-unit graph to be read by the integrator.

The mpc program will accept GCC command-line options. See the gcc(1) manual page for details on the available options. The most commonly used option will be -I dir to customize the include directories.

6.3. MPL/C Integrator

The MCL Integrator (cfg) reads the application config file (*appname.cfg*) and all the module elemental-unit graph files (*modulename.eul*), then generates and checks all the jobs, tasks, threads, and connections for the application. It outputs a loader map file (*appname.ldf*}), and a complete application elemntal-unit graph annotated with allocation and scheduling constraints and communication parameters (*appname.sch*). There are no cfg options normally used.

6.4. Allocator/Scheduler

The Allocator/Scheduler (**allocator**) attempts to find a valid allocation for the application tasks across the nodes of the network, and a valid schedule for each node and for the network bus. The allocation and schedules are considered valid if all allocation, communication, and scheduling constraints are met.

The allocator/scheduler stops when a valid allocation and schedule is found, or when it is determined that one cannot be found. There is no attempt to load-balance the nodes or minimize network communications beyond what is needed for a minimally valid schedule. The allocator outputs an allocation information file (*appname.alloc*) and calendar schedules file (*appname.cal*).
The allocator takes two flags:

-p <number of processors> Default: 1. The number of processors in the target system. It should match the number of nodes defined in the config file.

-t <tdma slot size> Default: 1000. The Time Division Multiplexed Access (TDMA) slot size for the network bus. This is the time, in microseconds, that each node will be alloted to transmit on the network. All the nodes get a TDMA slot in turn. The tdma slot size should stay between 1000 and 16000 microseconds, depending on the application's latency requirements and the network hardware's buffering capacity.

6.5. Maruti Binder

The Maruti Binder (mbind) reads in the loader map .ldf file, the allocation .alloc file, and the calendars .cal file, and generates the static data structures needed by the runtime system (*appname-globals.c*). It also generates a makefile (*appname-bind.mk*) that manages the linking of each task of the application within its own logical address space, then linking all tasks together with the various flavors of the runtime library.

6.6. Timing Trace Analyzer

The Timing Trace Analyser (timestat) takes a list of timing output files as generated by the runtime system and generates a .wcet file that contains the worst case execution times for the elemental units, as needed by the allocator. Timestat also prints other statistics generated by the runtime system.

6.7. Timing Stats Monitor

Timing information is output from a stand-alone Maruti system through a serial port when the application terminates. The mgettimes program, running on another computer connected to the other end of that serial line, will receive the timing data and store it in a file suitable for processing by timestat. Mgettimes can process the output of multiple runs on the test setup, even from different applications. Simply leave the program running and any data that is received will be saved.

Mgettimes is called as follows:

```
mgettimes <speed> <serial-port>
```

where <speed> is the communications rate at which the times will be output (19200 bps in the default core), and <serial-port> is the device file for the communications port (for example, /dev/tty00 for the PC's COM1 port).

Maruti 3.1
Design Overview
First Edition

Systems Design and Analysis Group
Department of Computer Science
University of Maryland at College Park

1. Introduction

Many complex, mission critical systems depend not only on correct functional behavior, but also on correct temporal behavior. These systems are called. The most critical systems in this domain are those which must support applications with hard real-time constraints, in which missing a deadline may cause a fatal error. Due to their criticality, jobs with hard real-time constraints must always execute satisfying the user specified timing constraints, despite the presence of faults such as site crashes or link failures.

A real-time operating system, besides having to support most functions of a conventional operating system, carries the extra burden of guaranteeing that the execution of its requested jobs will satisfy their timing constraints. In order to carry out real-time processing, the requirements of the jobs have to be specified to the system, so that a suitable schedule can be made for the job execution. Thus, conventional application development techniques must be enhanced to incorporate support for specification of timing and resource requirements. Further, tools must be made available to extract these requirements from the application programs, and analyze them for schedulability.

Based on the characteristics of its jobs, a real-time system can be classified as static, dynamic or. In a static system, all (hard real-time) jobs and their execution characteristics are known ahead of time, and thus can be statically analyzed prior to system operation. Many such systems are built using the cyclic executive or static priority architecture. In contrast, there are many systems in which new processing requests may be made while the system is in operation. In a dynamic system, new requests arrive asynchronously and must be processed immediately. However, since new requests demand immediate attention, such systems must either have "soft" constraints, or be lightly loaded and rely on exception mechanisms for violation of timing constraints. In contrast, reactive systems have certain lead time to decide whether or not to accept a newly arriving processing request. Due the presence of the lead time, a reactive system can carry out analysis without adversely affecting the schedulability of currently accepted requests. If adequate resources are available then the job is accepted for execution. On the other hand, if adequate resources are not available then the job is rejected and does not execute. The ability to reject new jobs distinguishes a reactive system from a completely dynamic system.

The purpose of the Maruti project is to create an environment for the development and deployment of critical applications with hard real-time constraints in a reactive environment. Such applications must be able to execute on a platform consisting of distributed and heterogeneous resources, and operate continuously in the presence of faults.

The Maruti project started in 1988. The first version of the system was designed as an object-oriented system with suitable extensions for objects to support real-time operation. The proof-of-concept version of this design was implemented to run on top of the Unix operating system and supported hard and non-real-time applications running in a distributed, heterogeneous environment. The feasibility of the fault tolerance concepts incorporated in the design of Maruti system were also demonstrated. No changes to the Unix kernel were made in that implementation, which was operational in 1990. We realized that Unix is not a very hospitable host for real-time applications, as very little control over the use of resources can be exercised in that system without extensive modifications to the kernel. Therefore, based on the lessons learned from the first design, we proceeded with the design of the current version of Maruti and changed the implementation base to CMU Mach which permitted a more direct control of resources.

Most recently, we have implemented Maruti directly on 486 PC hardware, providing Maruti applications total control over resources. The initial version of the distributed Maruti has also been implemented, allowing Maruti applications to run across a network in a synchronized, hard real-time manner.

In this paper, we summarize the design philosophy of the Maruti system and discuss the design and implementation of Maruti. We also present the development tools and operating system support for mission critical applications. While the system is being designed to provide integrated support for multiple requirements of mission critical applications, we focus our attention on real-time requirements on a single processor system.

2. Maruti Design Goals

The design of a real-time system must take into consideration the primary characteristics of the applications which are to be supported. The design of Maruti has been guided by the following application characteristics and requirements.

- **Real-Time Requirements.** The most important requirement for real-time systems is the capability to support the timely execution of applications. In contrast with many existing systems, the next-generation systems will require support for hard, soft,and non-real-time applications on the same platform.

- **Fault Tolerance**. Many mission-critical systems are safety-critical, and therefore have fault tolerance requirements. In this context, fault tolerance is the ability of a system to support continuous operation in the presence of faults.

Although a number of techniques for supporting fault-tolerant systems have been suggested in the literature, they rarely consider the real-time requirements of the system. A real-time operating system must provide support for fault tolerance and exception handling capabilities for increased reliability while continuing to satisfy the timing requirements.

- **Distributivity.** The inherent characteristics of many systems require that multiple autonomous computers, connected through a local area network, cooperate in a distributed manner. The computers and other resources in the system may be homogeneous or heterogeneous. Due to the autonomous operation of the components which cooperate, system control and coordination becomes a much more difficult task than if the system were implemented in a centralized manner. The techniques learned in the design and implementation of centralized systems do not always extend to distributed systems in a straightforward manner.

- **Scenarios.** Many real-time applications undergo different modes of operation during their life cycle. A scenario defines the set of jobs executing in the system at any given time. A hard real-time system must be capable of switching from one scenario to another, maintaining the system in a safe and stable state at all times, without violating the timing constraints.

- **Integration of Multiple Requirements.** The major challenge in building operating systems for mission critical computing is the integration of multiple requirements. Because of the conflicting nature of some of the requirements and the solutions developed to date, integration of all the requirements in a single system is a formidable task. For example, the real-time requirements preclude the use of many of the fault-handling techniques used in other fault-tolerant systems.

3. Design Approach and Principles

Maruti is a time-based system in which the resources are reserved prior to execution. Resource reservation is done on the time-line, thus allowing for reasoning about real-time properties in a natural way. The time-driven architecture provides predictable execution for real-time systems, a necessary requirement for critical applications requiring hard real-time performance. The basic design approach is outlined below:

- **Resource Reservation for Hard Real-Time Jobs.** Hard real-time applications in Maruti have advance resource reservation resulting in a priori guarantees about the timely execution of hard real-time jobs. This is achieved through a calendar data structure which keeps track of all resource reservations and the assigned time intervals. The resource requirements are specified as early as possible in the development stage of an application and are manipulated, analyzed, and refined through all phases of application development.

- **Predictability through Reduction of Resource Contention.** Hard real-time jobs are scheduled using a time-driven scheduling paradigm in which the resource contention between jobs is eliminated through scheduling. This results in reduced runtime overheads and leads to a high degree of predictability. However, not all jobs can be pre-scheduled. Since resources may be shared between jobs in the calendar and other jobs in the system, such as non-real-time activities, there may be resource contention leading to lack of predictability. This is countered by eliminating as much resource contention as possible and reducing it whenever it is not possible to eliminate it entirely. The lack of predictability is compensated for by allowing enough slack in the schedule.

- **Integrated Support for Fault Tolerance.** Fault tolerance objectives are achieved by integrating the support for fault tolerance at all levels in the system design. Fault tolerance is supported by early fault detection and handling, resilient application structures through redundancy, and the capability to switch modes of operation. Fault detection capabilities are integrated into the application during its development, permitting the use of application specific fault detection and fault handling. As fault handling may result in violation of temporal constraints, replication is used to make the application resilient. Failure of a replica may not affect the timely execution of other replicas and, thereby, the operation of the system it may be controlling. Under anticipated load and failure conditions, it may become necessary for the system to revoke the guarantees given to the hard real-time applications and change its mode of operation dynamically so that an acceptable degraded mode of operation may continue.

- **Separation of Mechanism and Policy**. In the design of Maruti, an emphasis has been placed on separating mechanism from policy. Thus, for instance, the system provides basic dispatching mechanisms for a time-driven system, keeping the design of specific scheduling policies separate. The same approach is followed in other aspects of the system. By separating the mechanism from the policy, the system can be tailored and optimized to different environments.

- **Portability and Extensibility.** Unlike many other real-time systems, the aim of the Maruti project has been to develop a system which can be tailored to use in a wide variety of situations---from small embedded systems to complex mission-critical systems. With the rapid change in hardware technology, it is imperative that the design be such that it is portable to different platforms and makes minimal assumptions about the underlying hardware platform. Portability and extensibility is also enhanced by using modular design with well defined interfaces. This allows for integration of new techniques into the design with relative ease.

- **Support of Hard, Soft, and Non-Real-Time in the Same Environment**. Many critical systems consist of applications with a mix of hard, soft, and non-real-time requirements. Since they may be sharing data and resources, they must execute within the same environment. The approach taken in Maruti is to support the integrated execution of applications with multiple requirements by reducing and bounding the unpredictable interaction between them.

- **Support for Distributed Operation**. Many embedded systems require several processors. When multiple processors function autonomously, their use in hard real-time applications requires operating system support for coordinated resource management. Maruti provides coordinated, time-based resource management of all resources in a distributed environment including the processors and the communication channels.

- **Support for Multiple Execution Environments**. Maruti provides support for multiple execution environments to facilitate program development as well as execution. Real-time applications may execute in the Maruti/Mach or Maruti/Standalone environments and maintain a high degree of temporal determinacy. The Maruti/Standalone environment is best suited for the embedded applications while Maruti/Mach permits the concurrent execution of hard real-time and non-real-time Unix applications. In addition, the Maruti/Virtual environment has been designed to aid the development of real-time applications. In this environment the same code which runs in the other two environments can execute while access to all Unix debugging tools is available. In this environment, temporal accuracy is maintained with respect to a virtual real-time.

- **Support for Temporal Debugging**. When an application executes in the Maruti/Virtual environment its interactions are carried out with respect to virtual real-time which is under the control of the user. The user may speed it up with respect to actual time or slow it down. The virtual time may be paused at any instant and the debugging tools used to examine the state of the execution. In this way we may debug an application while maintaining all temporal relationships, a process we call temporal debugging.

4. Application Development Environment

To support applications in a real-time system, conventional application development techniques and tools must be augmented with support for specification and extraction of resource requirements and timing constraints. The application development system provides a set of programming tools to support and facilitate the development of real-time applications with diverse requirements. The Maruti Programming Language (MPL) is used to develop individual program modules. The Maruti Configuration Language (MCL) is used to specify how individual program modules are to be connected together to form an application and the details of the hardware platform on which the application is to be executed.

4.1. Maruti Programming Language

Rather than develop completely new programming languages, we have taken the approach of using existing languages as base programming languages and augmenting them with Maruti primitives needed to provide real-time support.

In the current version, the base programming language used is ANSI C. MPL adds *modules*, *shared memory blocks*, *critical regions*, *typed message passing*, *periodic functions*, and *message-invoked functions* to the C language. To make analyzing the resource usage of programs feasible, certain C idioms are not allowed in MPL; in particular, recursive function calls are not allowed nor are unbounded loops containing externally visible events, such as message passing and critical region transitions.

- The code of an application is divided into *modules*. A module is a collection of procedures, functions, and local data structures. A module forms an independently compiled unit and may be connected with other modules to form a complete application. Each module may have an *initialization function* which is invoked to initialize the module when it is loaded into memory. The initialization function may be called with arguments.

- Communication primitives send and receive messages on one-way typed *channels*. There are several options for defining channel endpoints that specify what to do on buffer overflow or when no message is in the channel. The connection of two end-points is done in the MCL specification for the application-Maruti insures that end-points are of the same type and are connected properly at runtime.

- Periodic functions define *entry points* for execution in the application. The MCL specification for the application will determine when these functions execute.

- Message-invoked functions, called *services*, are executed whenever messages are received on a channel.

- *Shared memory blocks* can be declared inside modules and are connected together as specified in the MCL specifications for the application.

- An *action* defines a sequence of code that denotes an externally observable action of the module. Actions are used to specify timing constraints in the MCL specification.

- *Critical Regions* are used to safely access and maintain data consistency between executing entities. Maruti ensures that no two entities are scheduled to execute inside their critical regions at the same time.

4.2. Maruti Configuration Language

MPL Modules are brought together into as an executable application by a specification file written in the Maruti Configuration Language (MCL). The MCL specification determines the application's hard real-time constraints, the allocation of tasks, threads, and shared memory blocks, and all message-passing connections. MCL is an interpreted C-like language rather than a declarative language, allowing the instantiation of complicated subsystems using loops and subroutines in the specification. The key features of MCL include:

- **Tasks, Threads, and Channel Binding**. Each module may be instantiated any number of times to generate tasks. The threads of a task are created by instantiating the entries and services of the corresponding module. An entry instantiation also indicates the job to which the entry belongs. A service instantiation belongs to the job of its client. The instantiation of a service or entry requires binding the input and output ports to a channel. A channel has a single input port indicating the sender and one or more output ports indicating the receivers. The configuration language uses channel variables for defining the channels. The definition of a channel also includes the type of communication it supports, i.e., synchronous or asynchronous.

- **Resources.** All global resources (i.e., resources which are visible outside a module) are specified in the configuration file, along with the access restrictions on the resource. The configuration language allows for binding of resources in a module to the global resources. Any resources use by a module which are not mapped to a global resource are considered local to the module.

- **Timing Requirements and Constraints**. These are used to specify the temporal requirements and constraints of the program. An application consists of a set of cooperating jobs. A job is a set of entries (and the services called by the entries) which closely cooperate. Associated with each job are its invocation characteristics, i.e., whether it is periodic or aperiodic. For a periodic job, its period and, optionally, the ready time and deadline within the period are specified. The constraints of a job apply to all component threads. In addition to constraints on jobs and threads, finer level timing constraints may be specified on the observable actions. An observable action may be specified in the code of the program. For any observable action, a ready time and a deadline may be specified. These are relative to the job arrival. An action may not start executing before the ready time and must finish before the deadline. Each thread is an implicitly observable action, and hence may have a ready time and a deadline.

Apart from the ready time and deadline constraints, programs in Maruti can also specify *relative* timing constraints, those which constrain the interval between two events. For each action, the start and end of the action mark the observable events. A relative constraint is used to constrain the temporal separation between two such events. It may be a relative deadline constraint which specifies the upper bound on time between two events, or a delay constraint which specifies the lower bound on time between the occurrence of the two events. The interval constraints are closer to the event-based real-time specifications, which constrain the minimum and/or maximum distance between two events and allow for a rich expression of timing constraints for real-time programs.

- Replication and Fault Tolerance. At the application level fault tolerance is achieved by creating resilient applications by replicating parts, or all, of the application. The configuration language eases the task of achieving fault tolerance by allowing mechanisms to replicate the modules, and services, thus achieving the desired amount of resiliency. By specifying allocation constraints, a programmer can ensure that the replicated modules are executed on different partitions.

5. Analysis and Resource Allocation

This phase involves analyzing the resource allocation and scheduling of a collection of applications in terms of their real-time and fault-tolerance properties. The properties of the system are analyzed with respect to the system configuration and the characteristics of the runtime system, and resource calendars are generated.

The analysis phase converts the application program into fine-grained segments called *elemental units*. All subsequent analysis and resource allocation are based on EUs.

5.1. Elemental Unit Model

The basic building block of the Maruti computation model is the elemental unit (EU). In general, an elemental unit is an executable entity which is triggered by incoming data and signals, operates on the input data, and produces some output data and signals. The behavior of an EU is atomic with respect to its environment. Specifically:

- All resources needed by an elemental unit are assumed to be required for the entire length of its execution.

- The interaction of an EU with other entities of the systems occurs either before it starts executing or after it finishes execution.

The components of an EU are illustrated in Figure 1 and are described below:

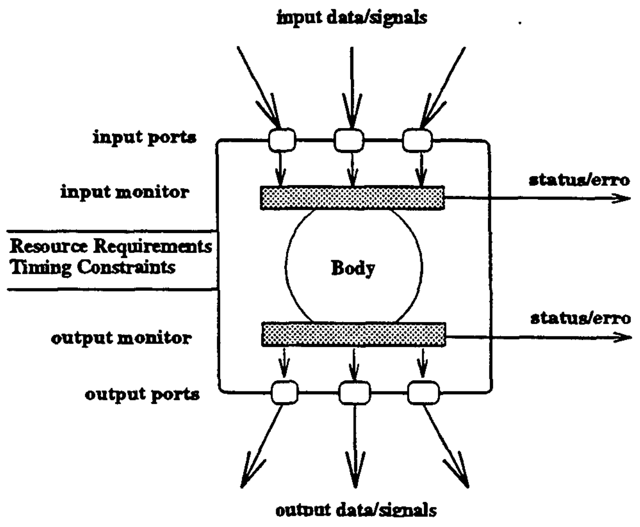


Figure 1: Structure of an Elemental Unit

- **Input and Output Ports**. Each EU mȧy have several input and/or output ports. Each port specifies a part of the interface of the EU. The input ports are used to accept incoming input data to the EU, while the output ports are used for feeding the output of the EU to other entities in the system.

- **Input and Output Monitors**. An input monitor collects the data from the input ports, and provides it to the main body. In doing so, it acts as a filter, and may also be used for error detection and debugging. The input monitors are also used for supporting different triggering conditions for the EU. Similar to input monitors, the output monitors act as filters to the outgoing data. The output monitor may be used for error detection and timingconstraint enforcement. The monitors may be connected to other EUs in the system and may send (asynchronous) messages to them reporting errors or status messages. The receiving EU may perform some error-handling functions.

- **Main Body**. The main body accepts the input data from the input monitor, acts on it, and supplies the output to the output monitor. It defines the functionality provided by the EU.

Annotated with an elemental unit are its resource requirements and timingconstraints, which are supplied to the resource schedulers. The resource schedulers must ensure that the resources are made available to the EU at the time of execution and that its timing constraints are satisfied.

5.2. Composition of EUs

In order to define complex executions, the EUs may be composed together and properties specified on the composition. Elemental units are composed by connecting an output port of an EU with an input port of another EU. A valid connection requires that the input and output port types are compatible, i.e., they carry the same message type. Such a connection marks a one-way flow of data or control, depending on the nature of the ports. A composition of Eus can be viewed as a directed acyclic graph, called an *elemental unit graph* (EUG), in which the nodes are the EUs, and the edges are the connections between EUs. An incompletely specified EUG in which all input and output ports are not connected is termed as a *partial EUG* (PEUG). A partial EUG may be viewed as a higher level EU. In a complete EUG, all input and output ports are connected and there are no cycles in the graph. The acyclic requirement comes from the required time determinacy of execution. A program with unbounded cycles or recursions may not have a temporally determinate execution time. Bounded cycles in an EUG are converted into an acyclic graph by loop unrolling.

The composition of EUs supports higher level abstractions and the properties associated with them. By carefully choosing the abstractions, the task of developing applications and ensuring that the timing and other operational constraints are satisfied can be greatly simplified. In Maruti, we have chosen the following abstractions:

- *A thread* is a sequential composition of elemental units. It has a sequential flow of control which is triggered by a message to the first EU in the thread. The flow of control is terminated with the last EU in the thread. Two adjacent EUs of a thread are connected by a single link carrying the flow of control. The component elemental units may receive messages or send messages to elemental units outside the thread. All EUs of a thread share the execution stack and processor state.

- *A job* is a collection of threads which cooperate with each other to provide some functionality. The partial EUGs of the component threads are connected together in a well defined manner to form a complete EUG. All threads within a job operate under a global timing constraint specified for the job.

5.3. Program Analysis

Program modules are independently compiled. In addition to the generation of the object code, compilation also results in the creation of partial EUGs for the modules, i.e., for the services and entries in the module, as well as the extraction of resource requirements such as stack sizes for threads, memory requirements, and logical resource requirements.

Invocation of an entry point and service call starts a new thread of execution. A control flow graph is generated for each service and entry. The control flow graph and the MPL primitives are used to delineate EU boundaries. Note that an EU execution is atomic, i.e., all resources required by the EU are assumed to be used for the entire duration of its execution. Further, all input messages are assumed to be logically received at the start of an EU and all output messages are assumed to be logically sent at the end of an EU. At compilation time, the code for each entry and service is broken up into one or more elemental units. The delineation of EU boundaries is done in a manner that ensures that no cycles are formed in the resultant EUG. Thus, for instance, a send followed by a receive within the same EU may result in a cyclic precedence and must be prevented. We follow certain rules of thumb to delineate EU boundaries, which may be overridden and explicitly changed by the user. The EU boundaries are created at a receive statement, the beginning and end of a resource block, and the beginning and end of an observable action. For each elemental unit a symbolic name is generated and is used to identify it. The predecessors and successors of the EU as well as the source code line numbers associated with the EU are identified and stored. The resource and timing requirements that can be identified during compilation are also stored, and place holders are created for the remaining information.

Given an application specification in the Maruti Configuration Language and the component application modules, the integration tools are responsible for creating a complete application program and extracting out the resource and timing information for scheduling and resource allocation. The input to the integration process are the program modules, the partial EUGs corresponding to the modules, the application configuration specification, and the hardware specifications. The outputs of the integration process are: a specification for the loader for creating tasks, populating their address spaces, creating the threads ad channels, and initializing the task; loadable executables of the program; and the complete application EUG along with the resource descriptions for the resource allocation and scheduling subsystem.

5.4. Communication Model

Maruti supports message passing and shared memory models for communication.

- **Message Passing.** Maruti supports the notion of one-way message passing between elemental units. Message passing provides a location-independent and architecture-transparent communication paradigm. A channel abstraction is used to specify a one way message communication path between a sender and a receiver. A one-way message-passing channel is set up by declaring the output port on the sender EU, the input port on the receiver EU, and the type of message. The communication is asynchronous with respect to the sender, i.e., the sender does not block.

- **Synchronous Communication**. Synchronous communication is used for tightly coupled message passing between elemental units of the same job. For every invocation of the sender there is an invocation of the receiver which accepts the message sent by the sender. The receiver is blocked (de-scheduled) until message arrival under normal circumstances. The messages in a synchronous communication channel are delivered in FIFO order.

- **Asynchronous Communication**. Asynchronous communication may be used for message passing between elemental units not belonging to the same job. It may also be used between real-time and non-real-time jobs. In such communication, neither the sender nor the receiver is blocked (i.e., there is no synchronization). Since the sender and receiver may execute at different rates, it is possible that no finite amount of buffers suffice. Hence, an asynchronous communication channel is inherently lossy. The receiver may specify its input port to be inFirst or inLast to indicate which messages to drop when the buffers are full. The first message is dropped in an inLast channel, while the last message is dropped in an inFirst channel.

There may be multiple receivers of a message, thus allowing for multi-cast messages. Similar to a one-to-one channel, a multicast channel may also be synchronous or asynchronous. All receivers of a multi-cast message must be of the same type.

- **Shared Memory.** Shared memory is also supported in Maruti. The simplest way to share memory between EUs is to allow them to exist within the same address space. We use task abstraction for this purpose. A task consists of multiple threads operating within it, sharing the address space. The task serves as an execution environment for the component threads. A thread may belong to only one task. In addition to the shared memory within a task, inter-task sharing is also supported through the creation of shared memory partitions. A shared memory partition is a shared buffer which can be accessed by any EU permitted to do so. The shared memory partitions provide an efficient way to access data shared between multiple EUs. The shared memory communication paradigm provides just the shared memory - it is the user's responsibility to ensure safe access to the shared data. This can be done by defining a logical resource and ensuring that the resource is acquired every time the shared data is accessed. By providing appropriate restrictions on the logical resource, safe access to data can be ensured.

5.5. Resource Model

A distributed system consists of a collection of autonomous processing nodes connected via a local area network. Each processing node has resources classified as processors, logical resources, and peripheral devices. Logical resources are used to provide safe access to shared data structures and are passive in nature. The peripheral devices include sensors and actuators. Restrictions may be placed on the preemptability of resources to maintain resource consistency. The type of the resource determines the restrictions that are placed on the preemptability of the resource and serves to identify operational constraints for the purpose of resource allocation and scheduling. We classify the resources into the following types based on the restrictions that are imposed on their usage.

- **Non-preemptable**. The inherent characteristics of a resource may be such that it prevents preemptability, i.e., any usage of the resource must not be preempted. Many devices require non-preemptive scheduling. For resources which require the use of CPU, this implies non-preemptive execution from the time the resource is acquired until the time the resource is released.

- **Exclusive.** Unlike a non-preemptive resource, an exclusive resource can be preempted. However, the resource may not be granted to anyone else in the meantime. A critical section is an example of a resource which must be used in exclusive mode.

- **Serially Reusable**. A serially reusable resource can not only be preempted but may also be granted to another EU. The state of such resources can be preserved and restored when the resource is granted back.

- **Shared**. A shared resource may be used by multiple entities simultaneously. In a single processor system, since only one entity is executing at a given time, there is no distinction between a shared resource and a serially reusable resource.

A non-preemptable resource is the most restrictive and a shared resource is the least restrictive in terms of the type of usage allowed. An application requesting the use of a resource must specify when the resource is to be acquired, when it is to be released, and the restrictions on the preemptability of the resource. The resource requirements for applications may be specified at different levels of computational abstractions as identified below.

- **EU level**. The lowest level a resource requirement can be specified at is the EU level. A resource requirement specified at the EU level implies that the resource is acquired and released within the EU. For scheduling purposes, it is assumed that the resource is required for the entire duration of the execution of the EU.

- **Thread Level.** Resource specification at the thread level is used for resources which are acquired and released by different EUs belonging to the same thread. For instance, a critical section may be acquired in one EU and released in another one.

- **Job Level.** Job-level resource specifications are used to specify resources which are not acquired and released for each invocation of a periodic or sporadic job. Instead, these resources are acquired at the job initialization and released at job termination. For a periodic job, an implicit resource associated with each thread are the thread data structures (including procesor stack and registers).

5.6. Operational Constraints

The execution of EUs is constrained through various kinds of operational constraints. Such constraints may arise out of restricted resource usage or through the operational requirements of the application. Examples of such constraints are: precedence, mutual exclusion, ready time, and deadline. They may be classified into the following categories:

- **Synchronization Constraints**. Synchronization constraints arise out of data and control dependencies or through resource preemption restrictions. Typical examples of such constraints are precedence and mutual exclusion.

- **Timing Constraints.** Many types of timing constraints may be specified at different levels, i.e., at job level, thread level, or EU level. At the job level, one may specify the ready time, deadline, and whether the job is periodic, sporadic, or aperiodic. For threads, a ready time and deadline may be specified relative to the job arrival. Likewise, a ready time and deadline may be specified for an individual EU. We also support the notion of relative timing constraints, i.e., constraints on the temporal distance between the execution of two EUs.

- **Allocation Constraints.** In our model, tasks are allocated to processing nodes. Allocation constraints are used to restrict the task allocation decisions. Allocation constraints often arise due to fault-tolerance requirements, where the replicas of EUs must be allocated on different processing nodes. Similarly, when two tasks share memory, they must be allocated on the same processing node. Sometimes a task must be bound to a processing node since it uses a particular resource bound to the node (e.g., a sensor).

The operational constraints are made available to the resource allocation and scheduling tools which must ensure that the allocation and scheduling maintains the restrictions imposed by the constraints. The model does not place any a priori restrictions on the nature of the constraints that may be specified. However, the techniques used by the resource allocator and scheduler will depend on the type of constraints that can be specified.

5.7. Allocation and Scheduling

After the application program has been analyzed and its resource requirements and execution constraints identified, it can be allocated and scheduled for a runtime system.

This final phase of program development depends upon the physical characteristics of the hardware on which the application will be run, for example, the location of devices and the number of nodes and type of processors on each node in the distributed system.

Maruti uses time-based scheduling and the scheduler creates a data structure called a *calendar* which defines the execution instances in time for all executable components of the applications to be run concurrently.

We consider the static allocation and scheduling in which a task is the finest granularity object of allocation and an EU instance is the unit of scheduling. In order to make the execution of instances satisfy the specifications and meet the timing constraints, we consider a scheduling frame whose length is the least common multiple of all tasks' periods. As long as one instance of each EU is scheduled in each period within the scheduling frame and these executions meet the timing constraints a feasible schedule is obtained.

As a part of the Maruti development effort, a number of scheduling techniques have been developed and are used for generating schedules and calendars for task sets. These techniques include the use of temporal analysis and simulated annealing. Schedules for single-processor systems as well as multiple-processor networks are developed using these techniques.

6. Maruti Runtime System

The runtime system provides the conventional functionality of an operating system in a manner that supports the timely dispatching of jobs. There are two major components of the runtime system - the *Maruti core*, which is the operating system code that implements scheduling, message passing, process control, thread control, and low level hardware control, and the *runtime dispatcher*, which performs resource allocation and scheduling for dynamic arrivals.

6.1. The Dispatcher

The dispatcher carries out the following tasks:

- **Resource Management.** The dispatcher handles requests to load applications. This involves creating all the tasks and threads of the application, reserving memory, and loading the code and data into memory. All the resources are reserved before an application is considered successfully loaded and ready to run.

- **Calendar Management**. The dispatcher creates and loads the calendars used by applications and activates them when the application run time arrives. The application itself can activate and deactivate calendars for scenario changes.

- **Connection Management**. A Maruti application may consist of many different tasks using channels for communication. The dispatcher sets up the connections between the application tasks using direct shared buffers for local connections or a shared buffer with a communications agent for remote connections.

- **Exception Handling**. Rogue application threads may generate exceptions such as missed deadlines, arithmetic exceptions, stack overflows, and stray accesses to unreserved memory. These exceptions are normally handled by the dispatcher for all the Maruti application threads. Various exception handling behaviors can be configured, from terminating the entire application or just the errant thread, to simply invoking a task-specific handler.

6.2. Core Organization

The core of the Maruti hard real-time runtime system consists of three data structures:

- *The calendars* are created and loaded by the dispatcher. Kernel memory is reserved for each calendar at the time it is created. Several system calls serve to create, delete, modify, activate, and deactivate calendars.

- *The results table* holds timing and status results for the execution of each elemental unit. The maruti_calendar_results system call reports these results back up to the user level, usually to the dispatcher. The dispatcher can then keep statistics or write a trace file.

- *The pending activation table* holds all outstanding calendar activation and deactivation requests. Since the requests can come before the switch time, the kernel must track the requests and execute them at the correct time in the correct order.

The scheduler gains control of the CPU at every clock tick interrupt. At that time, if a Maruti thread is currently running and its deadline has passed its execution is stopped and an exception raised.

If any pending activations are due to be executed those requests are handled, thereby changing the set of active calendars. Then the next calendar entry is hecked to see if it is scheduled to execute at this time. If so, the scheduler switches immediately to the specified thread. If no hard real-time threads are scheduled to execute, the calendar scheduler falls through to the soft and non-real-time, priority-based schedulers.

Maruti threads indicate to the scheduler that they have successfully reached the end of their elemental unit with the maruti_unit_done system call. This call marks the current calendar entry as done and fills in the time actually used by the thread. The Maruti thread is then suspended until it next appears in the calendars. Soft and non-real-time threads can be run until the next calendar entry is scheduled and are executed using a priority based scheduling for the available time slots.

At all times the Maruti scheduler knows which calendar entry will be the next one to run so that the calendars are not continually searched for work. This is recalculated when maruti_unit_done is called or whenever the set of active calendars changes.

6.3. Multiple Scenarios

The Maruti design includes the concept of scenarios, implemented at runtime as sets of alternative calendars that can be switched quickly to handle an emergency or a change in operating mode. These calendars are pre-scheduled and able to begin execution without having to invoke any user-level machinery. The dispatcher loads the initial scenarios specified by the application and activates one of them to begin normal execution. However, the application itself can activate and deactivate scenarios. For example, an application might need to respond instantaneously to the pressing of an emergency shutdown button. A single system call then causes the immediate suspension of normal activity and the running of the shutdown code sequence. Calendar activation and deactivation commands can be issued before the desired switch time. The requests are recorded and the switches occur at the precise moment specified. This allows the application to insure smooth transitions at safe points in the execution.

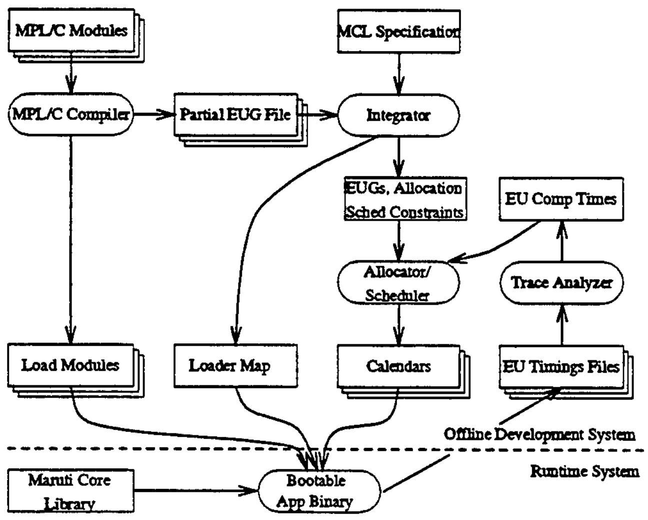


Figure 2: Maruti System Architecture

7. Maruti 3.1 System Architecture

Maruti 3.1, the current version of the operating system, implements most of the above design with a series of development tools that operate in a Berkeley Unix development environment (NetBSD 1.0) on IBM-compatible 486 or Pentium PCs. Maruti applications can be run stand-alone on the bare hardware or under a Unix-based debugging environment.

- MPL code is processed by the MPL compiler, a modified version of the GNU C compiler. The MPL compiler generates both the compiled object code and partial EUG file that contains all information extracted from the module for further analysis, including the boundaries of the elemental units of the program.

- The application's MCL specification is read and interpreted by the *integrator*. The PEUG file describing each module used in the application is processed and intermodule type checking performed. The integrator generates a file specifying the full application EUG, allocation, and scheduling constraints.

- The allocator/scheduler reads in the data supplied by the integrator and a description of the physical system on which to allocate the application. The allocator searches for an arrangement of elemental units on the nodes of the network that satisfies all the timing and allocation constraints, considering the computation times for each elemental unit. If a feasible schedule can be found, a *calendar file* for each resource is generated. A loader map is also generated which describes, for the runtime system, each task, thread, shared memory area, and communications link so that all the resources can be reserved when the application is loaded.

- The computation time analyzer takes timing trace information generated by the runtime system and generates worst-case execution times for all the Eus of the application. This timing information can be used in subsequent runs of the scheduler to refine the schedule and verify its feasibility given changes in computation times. Use of the timing tool during testing leads to very high confidence in the schedule.

7.1. Runtime Environments

Compiled and analyzed Maruti applications can be executed in multiple runtime environments.

- The *Maruti/Virtual* runtime environment allows the debugging of Maruti applications within the development environment. Applications run in virtual real-time under Unix, allowing temporal debugging, including single stepping the real-time calendars.

- The *Maruti/Mach* runtime environment is a modified version of Mach which allows the running of real-time Maruti programs within the Mach environment, where the real-time and non-real-time task can co-exist and interact in the same host.

- The *Maruti/Standalone* runtime environment runs the application on the bare hardware, suitable for embedded systems. The application is linked with a minimal Maruti core library and can be booted directly.

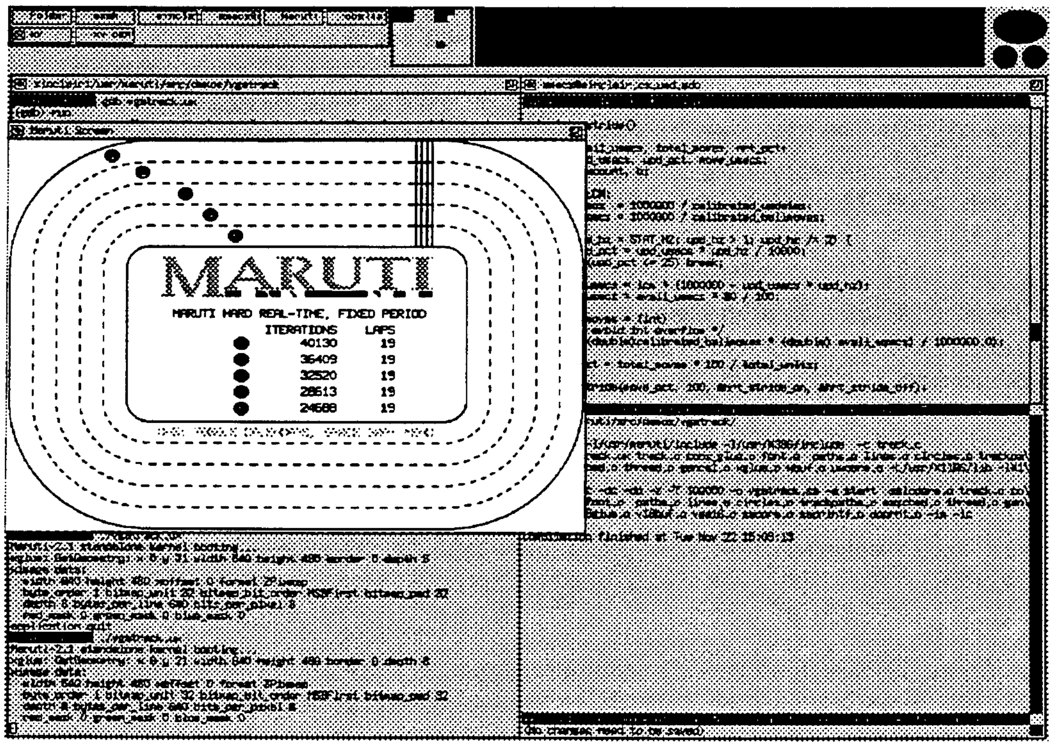
7.2. Maruti/Virtual Runtime Environment

Testing real-time programs in their native embedded environments can be tedious and very time-consuming because of the lack of debugging facilities and the requirement to reload and reboot the target computer every time a change is made. Maruti provides a Unix-based runtime system that allows the execution of Maruti hard-real-time applications from within the Unix development environment. This Unix execution environment supports the following features:

- The Maruti application has direct control of its I/O device hardware.

- Graphical output and keyboard input can go either to the PC console, as in the Maruti/Standalone and Maruti/Mach environments, or appear in an X window on any Unix workstation, possibly across the network.

- The application can be run under the Unix GNU Debugger, allowing the examination of program variables and stack traces, setting of breakpoints, and post-mortem analysis.



```
(gdb) run
```

```
MARUTI
MARUTI HARD REAL-TIME, FIXED PERIOD
     ITERATIONS   LAPS
●       40130      19
●       36409      19
●       32520      19
●       28613      19
●       24688      19
```

```
Maruti-2.1 standalone kernel booting...
vgalinux: GetGeometry: x 0 y 21 width 640 height 480 border 0 depth 8
ximage data:
  width 640 height 480 xoffset 0 format ZPixmap
  byte_order 1 bitmap_unit 32 bitmap_bit_order MSBFirst bitmap_pad 32
  depth 8 bytes_per_line 640 bits_per_pixel 8
  red_mask 0 green_mask 0 blue_mask 0
application quit
./vgatrack.ux
Maruti-2.1 standalone kernel booting...
vgalinux: GetGeometry: x 0 y 21 width 640 height 480 border 0 depth 8
ximage data:
  width 640 height 480 xoffset 0 format ZPixmap
  byte_order 1 bitmap_unit 32 bitmap_bit_order MSBFirst bitmap_pad 32
  depth 8 bytes_per_line 640 bits_per_pixel 8
  red_mask 0 green_mask 0 blue_mask 0
```

Figure 3: Maruti/Virtual screen running in the development environment

- The application has access to Unix standard output so it can print debug and status messages to the interactive session while running.

- The Maruti application runs in *virtual real-time*; that is, it sees itself running in hard-real-time against a virtual time base.

- The virtual time can be manipulated through the runtime system for temporal debugging. Virtual time can be slowed down or sped up, and individual elemental units (EU) or whole calendars can be single-stepped or traced.

7.3. Maruti/Standalone Real-Time Environment

- Maruti/Standalone provides a minimal runtime system for the execution of a Maruti application on the bare hardware. The stand-alone environment has the following attributes:

- The stand-alone version of an application is built from the same object modules as are used in the Unix and Maruti/Mach execution environments.

- All the modules of the application are bound with only those routines of the Maruti core that are needed into one executable, suitable for booting directly or converting into ROM.

- The application has complete control of the computer hardware.

- The application runs in hard real-time with very low overhead and variability.

- The minimal Maruti/Standalone core library currently consists of about 16 KB of code and 16 KB of data.

- The optional Maruti Distributed Operation support (including network driver) is about 14 KB of code and 9 KB of data.

- The optional Maruti graphics package currently consists of, for the standard VGA version, 10 KB of code and 20 KB of data (plus 150K for a secondary frame buffer for best performance).

7.4. Maruti/Mach Real-Time Environment

The original execution environment for Maruti-2 was a modified version of the CMU Mach 3.0 kernel. Maruti/Mach is potentially useful in hybrid environments in which the real-time components co-exist with Mach and Unix processes on the same CPU. Because of preemptability problems in CMU Mach we will not be distributing Maruti/Mach until it can be rehosted onto OSF1/MK real-time kernels.

The Maruti/Mach features include the following:

- A calendar-based real-time scheduler has been added to the CMU Mach 3.0 kernel. This scheduler takes precedence over the existing Mach scheduler, running Maruti elemental units from the calendar at the proper release time.

- The Maruti application and most of the runtime system run as normal Mach user-level tasks and threads, which are wired down in memory.

- The Maruti application may communicate with non-Maruti Unix and Mach processes through shared memory.

- The Maruti/Mach kernel maintains runtime information for each elemental unit executed, and makes that information available to the user-level code for worst-case computation time analysis.

- Parts of the CMU Mach kernel remain unpreemptable. Nevertheless, on a dedicated system we can achieve release time variability of about 100 microseconds. The context switch time is about 200 microseconds.

- The new release of OSF Research Institute Mach MK6.0 addresses most of the Mach kernel preemptability concerns. We will be porting Maruti/Mach to this base in the near future.

8. Future Directions

The Maruti Project is an ongoing research effort. We hope to extend the current system in a number of possible directions. Of course, since this is a research project, we expect our ideas to evolve over time as we gain experience and get feedback from users.

8.1. Scheduling and Analysis Extensions

Preemptable Scheduling of Hard-Real-Time Tasks

We are planning to extend our scheduling approach to incorporate controlled preemptions of tasks. To date we have concentrated on using non-preemptable executions of tasks, which simplifies scheduling and eases exclusion problems in application development. However, the non-preemptability assumption to exclusion is not scalable to a multiprocessor, as threads running on different processors can interfere with each other. Controlled preemption is more powerful, as it allows scheduling of long-running tasks concurrently with high frequency tasks. Preemption will remain under the control of the application.

Language support for atomic actions will be developed to replace the assumption of non-preemptable EU's. Action statements will serve to delineate sections of code on which precise timing requirements can be imposed by the application designer. Combined with critical region statements (already implemented), actions will allow the programmer to specify precisely the desired timing and resource interrelationships in a manner that is scalable to a multiprocessor or network cluster, unlike the non-preemptability assumption.

We will extend the Maruti run-time system to handle preemptable hard real-time tasks. This will be done in coordination with the analysis tools which will generate multiple calendar entries for the preempted EUs. All but the last entry for the EU will be marked as preemptable, and all but the first will be marked as continuation entries. This is enough information for the run-time scheduler to correctly handle the preemption in a controlled manner, even when the EU completes early.

Integration of Time-based and Priority-Based Scheduling

We plan to integrate the time-based and priority-based scheduling in a single framework. To date we have concentrated on time-based scheduling only. To support other scheduling paradigms within the time-based framework, we may reserve time slots in the schedule and associate a queue of waiting tasks which are executed on the basis of their priorities. In this way we can implement rate-monotonic style static priority schemes as well as Earliest-Deadline-First style dynamic priority schemes within the Maruti framework. However, in order to assure that the tasks executed under priority-based scheduling will continue to meet their temporal requirements, extensions to the analysis techniques are required. We will develop analysis techniques suitable for this purpose.

We will extend the Maruti implementation to support non-calendar schedulers, such as priority based or earliest-deadline-first based schedulers. These schedulers will run in particular slots specified in the Maruti calendar, or when the calendar is idle.

POSIX-RT Subset API

In a related area, we plan to study the use of a subset of the POSIX API as the Maruti API for soft and non-real-time tasks. We will implement as much of the POSIX-RT API as is appropriate and practicable.

Asynchronous Events

Generally, in a time-based system, events are polled for at the maximum frequency at which they are expected. This type of event handling is easy to analyze within the time-based framework, and makes explicit the need to reserve enough time to handle the event stream at its worst-case arrival rate. At this worst-case rate, polling is more efficient than interrupt-driven event handling because the interrupt overhead is avoided. However, at low event rates, polling is less efficient and fragments the cpu idle time (where we define idle time from the point of view of hard real-time tasks). While conservation of idle time is not an issue for small controllers, it becomes very important when there are soft- and non-real-time tasks running in the system.

Currently, Maruti takes the polling approach to ease analysis and to better handle the worst case rate. We plan to study the analysis required to accommodate asynchronous events within a calendar schedule. Our intended approach is to work with a specified maximum frequency, relative deadline, and computation time of the asynchronous event, and to reserve enough time in the calendar for the event to occur at its maximum frequency.

We will extend the Maruti run-time system to register and dispatch event handlers in response to external events. Included in this extension will be the ability to detect and appropriately handle overload conditions (i.e. when the events occur more quickly that expected).

Multi-Dimensional Resource Scheduling Research

A typical real-time application requires several resources for it to execute. While CPU is the most critical resource, others have to be made available in a timely manner. Generation of schedules for multiple resources is known to be a difficult problem. Our approach to date has been to develop efficient search techniques, such as one based on simulated annealing.

Realistic problems contain a variety of interdependencies among tasks which must be reflected as constraints in scheduling. We plan to develop efficient techniques for scheduling the allocation and deallocation of portions of multidimensional resources. In particular, we will address the problems of allocation and management of resources such as memory and disk space, that can accommodate many entities simultaneously.

Scheduling System-Specific Topologies

In a related area, many communications networks have more complex structures than a simple bus and cannot be treated as a single dedicated resource. We will study the extension of our scheduling algorithms to support point-to-point meshes of nodes (with store-and-forward of messages), switched networks (such as MyriNet), and sophisticated backplanes such as that used in the Intel Paragon.

We will investigate the use of a general framework for specifying the properties of connection topologies to the Maruti scheduler. In the worst cases, the scheduler for a complex interconnection technology may have to be programmed explicitly. To handle such cases, we will develop a modular interface into our allocator/scheduler into which such backplane-specific schedulers can be plugged.

Static Estimation of Execution Times

Currently, execution times are derived through extensive testing of the program on the target hardware environment. Deriving the execution time through static analysis is hampered by the data dependencies present in large number in most programs.

We will investigate the use of static analysis to help prove the execution time limits of programs. While generating a reasonable computation time estimate through static analysis is not feasible in general, it is possible to get accurate results for large segments of a program, and to clearly identify the existing data dependencies so that the programmer can-through program modifications or directives to the analysis tool-eliminate, curtail, or characterize the data dependencies well enough to get very useful verification of the time properties of the program.

Temporal Debugging

When we develop real-time applications we need techniques for observing the temporal behavior of programs. For their functional characteristics we can use standard debuggers which permit the observation of the state of execution at any stage. This, however, destroys the temporal relationships completely. In Maruti/Virtual we provide the facilities of controlling the execution of all parts of an application with respect to a virtual time which advances under the control of keyboard directives. Thus we can pause the execution at any virtual time instant with the assurance that all temporal relationships with respect to this instant are accurately reflected in the state of the program. We use the term temporal debugging for this.

We will conduct research on the theoretical aspects of the issues of temporal debugging and consider the implications of temporal debugging. In particular, we will study how the interactions of programs executing in virtual time with external events which occur with respect to their own time line should be captured in temporal debugging. We will also study how the virtual times of several nodes in a distributed environment should be coordinated.

We will extend our implementation of temporal debugging tools in the Maruti/Virtual environment to support temporal debugging of distributed programs, and to support fine grained modification of the time line.

Dynamic Schedule Generation

We will develop the notion of time horizons to support controlled modifications of the hard real-time calendars at runtime to support programs that generate schedules dynamically. While the run-time mechanisms for modifying the calendars are already implemented, research issues relating to finding safe points to switch schedules, and scheduling the schedulers themselves, have to be studied before effective use can be made of on-line calendar generation.

8.2. Fault Tolerance

Maruti currently supports several powerful mechanisms for building fault tolerant applications:

- Maruti Configuration Language (MCL) constructs allow the application designer to specify replication of application subsystems with forkers and joiners inserted into the communication streams, as well as the allocation constraints necessary to correctly partition the replicated subsystems for the desired level of fault tolerance.
- Maruti Programming Language (MPL) allows the programming of application specific fault tolerance components such as forkers and joiners, elemental unit monitors, and channel monitors.
- The run-time system supports multiple calendars, allowing the application to switch to emergency or fault handling scenarios in real time.

We plan to extend the existing mechanisms by providing tools and new mechanisms to better automate the process of building fault tolerant applications. The new features will include:

- A library of forkers and joiners that can be incorporated into applications.
- Support for multicast messages.
- Better support in Maruti Programming Language (MPL) for EU and channel monitors.
- Automatic replication of subsystems, and analysis of fault tolerance properties through MAGIC, the graphical integrator described below.

8.3. Clock Synchronization

Currently, distributed Maruti handles clock drift at boot-up time, and thereafter time slave nodes simply adopt the time-master's clock periodically. This scheme is suitable for many applications, but is not ideal for embedded control systems that will suffer from a discontinuous time jump.

To address this problem we plan to develop and implement time-synchronization algorithms that operate concurrently with the distributed real-time program to continually adjust the clocks on all the nodes, taking into account changes in their relative drift. This will most likely involve a regular time pulse from a master clock, from which the other nodes continually measure their drift and fine-tune their tick rates. Since the clock drifts are about one order of magnitude less than the communication latency variances, a simple algorithm will not suffice here.

8.4. Heterogeneous Operation

We will extend our communications agents and boot protocol to translate typed Maruti messages between heterogeneous hosts when needed. The off-line Maruti analysis tools already collect information on the types of the channel endpoints for type-checking the connection. We will carry this information through to the run-time system for use in those channels that are connected between heterogeneous nodes.

8.5. MPL/Ada

We will incorporate Maruti Programming Language (MPL) features and analysis into the Ada 95 programming language as we did for ANSI C in the current MPL, which we will now refer to as MPL/C. Implementing MPL/Ada will involve the following tasks:

- A detailed design review studying those features of Ada which are compatible with Maruti and those that are not, and how best to proceed with the implementation of MPL/Ada.

- Port GNU Ada (GNAT) to our NetBSD development environment.

- Implement as much of the Ada run-time environment as is practicable on the Maruti run-time.

- Install hooks into GNAT to extract the resource usage information we need. We expect this work will leverage heavily from the MPL/C work, as GNAT is derived from the same back-end code base as GNU C.

- Develop and enforce within GNAT those restrictions on Ada constructs needed in order to preserve the properties needed for our hard real-time analysis.

- Add support for Maruti primitives to the language. Some Maruti primitives might be implementable directly through existing Ada facilities and thus will not require language extensions.

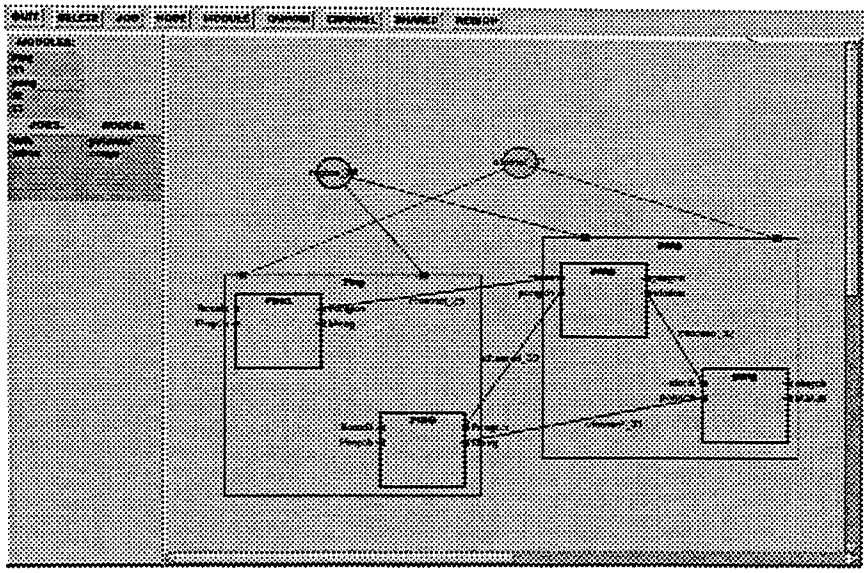


Figure 4: Prototype Graphical Program Integrator Tool.

8.6. Graphical Tools

Graphical Program Development Tools

Currently, Maruti applications are pulled together by an MCL specification, which takes the form of a procedural language whose primitive operations instantiate and bind together the parts of the application. This type of specification language is complete, allowing the specification of large, complex applications connected in arbitrary ways. However, such completeness makes MCL relatively low-level and tedious to program.

We are developing graphical program development tools which allow the application designer to pull together the modules using an entirely graphical user interface-avoiding MCL programming. The on-screen representation of modules can be interconnected with channels and grouped into hierarchical subsystems. The application designer will be able to zoom in and out to view the application at several levels.

The graphical environment will allow both the integration of existing modules and the development of the interfaces of modules that have not yet been written. The tools will generate template MPL code for those modules. In this way the graphical environment functions as a design tool and program generator as well as an integration environment.

The graphical environment will have fault-tolerance analysis built into it. Single points of failure will be identified on-screen. The user will be able to replicate entire subsystems at once, with the *forker* and *joiner* modules and allocation constraints introduced into the application automatically by the system.

This graphical style of application integration will greatly facilitate the building and deployment of reusable software components modules built to be easily customized and reintegrated into many applications. Given a suitable library of reusable component modules and the graphical integrator, it will be possible for non-programmers to build large custom applications from these parts.

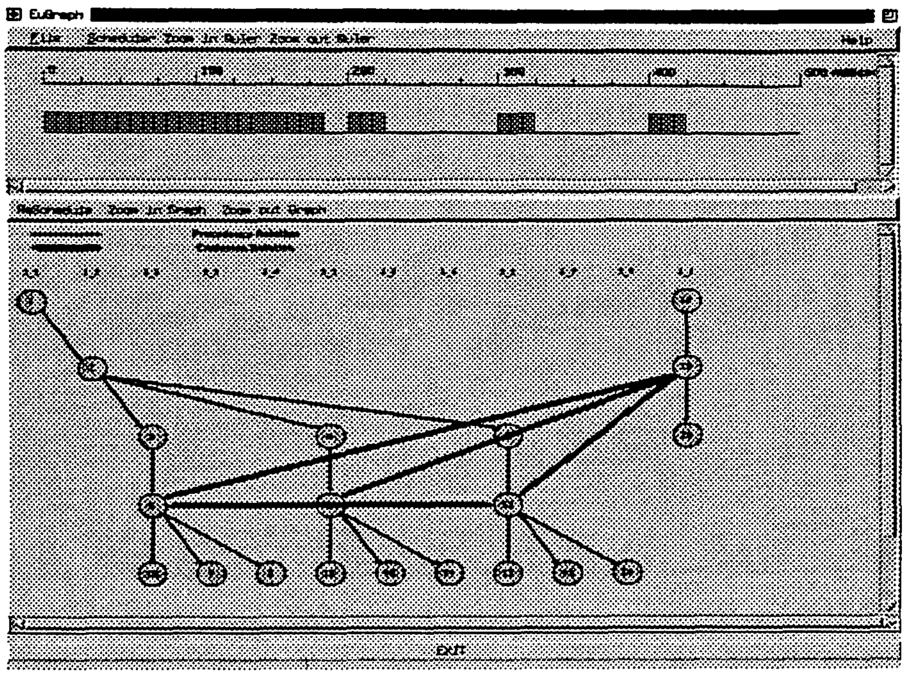


Figure 5: Prototype Graphical Resource Scheduling Tool

Graphical Resource Management Tools

Along with the graphical software development tools, we are pursuing graphical resource management tools. These are a non-programmer's interface into the advanced Maruti scheduling technology. The Maruti allocator/scheduler works with the abstract concepts of schedulable entities, available resources, and various types of constraints on the placement of entities and resources. In the Maruti operating system, the scheduling entities are EUs, and the resources are CPUs, network, memory, and devices - but in fact any type of entity or resource can be manipulated by the allocator/scheduler.

A graphical resource management tool will allow the specification of these entities, resources, and constraints on screen in a way more oriented towards the general user. With this tool users should be able to use Maruti scheduling technology to schedule classes, busses, or projects, for example.

We have built a small prototype of the graphical resource manager. The prototype displays the EU graph input to the scheduler as well as the calendar output of the scheduler. The user can edit the EU graph and its constraints and reschedule with the click of a button. The resulting resource calendar is redisplayed.

9. Availability

We are pleased to announce the availability of the Maruti 3.1 Hard Real-Time Operating System and Development Environment.

With Maruti 3.1, we are entering a new phase of our project. We have an operating system suitable for field use by a wider range of users, and we are embarking on the integration of our time-based, hard real-time technology with industry standards and more traditional event-based soft- and non-real-time systems. For this, we are greatly interested in the feedback from users as to the direction of evolution of the system.

For the Maruti 3.1 project, we will be pursuing the integration of a POSIX interface for soft and non-real-time applications, the use of Ada for Maruti programming, support for asynchronous events and soft/non-real time schedulers within the time-based framework, and heterogeneous Maruti networks.

For this user-oriented phase of the project we will be making regular releases of our software available to allow interested parties to track and influence our development. To begin this phase we are making our current base hard real-time operating system and its development environment available. This is an initial test release.

Maruti 3.1 will be made available to interested parties on request, via Internet ftp. Please send electronic mail to maruti-dist@cs.umd.edu for details. More information about the Maruti Project, as well as papers and documentation, are available via the World Wide Web at:

http://www.cs.umd.edu/projects/maruti/l

9.1. Runtime System

The Maruti 3.1 embeddable hard real-time runtime system for distributed and single-node systems includes the following features:

- The core Maruti runtime system is small - 16 KB code for the single node core, 30 KB code for the distributed core.

- The core provides a calendar-based scheduler, threads, distributed message passing using Time Division Multiplexed Access (TDMA) over the network, and tight time synchronization between network nodes.
- Also included in the runtime system is a graphics library suitable for system monitoring displays as well as simulations.

- Maruti runs on PC-AT compatible computers using the Intel i386 (with i387 coprocessor), i486DX, or Pentium processors. Distributed operation currently requires a 3Com 3c507 ethernet card. The graphics library supports standard VGA and Tseng-Labs ET-4000-based Super-VGA. Support for other SVGA chipsets is forthcoming soon.

9.2. Development Environment

Maruti 3.1 includes a complete development environment for distributed embedded hard real-time applications. The development environment runs on NetBSD Unix and includes the following:

- d The Maruti/Virtual debugging environment - simulates the Maruti runtime system within the development environment. The system clock in this environment tracks virtual time, which can be sped up, slowed down in relation to the actual time, or single-stepped or stopped. This allows temporal debugging of the application. Within Maruti/Virtual traces of the application scheduling and network traffic can be monitored in the debugging session.

- The ANSI-C based Maruti Programming Language (MPL/C). MPL adds modules, message passing primitives, shared memory, periodic functions, message-invoked functions, and exclusion regions to ANSI C. MPL is processed by a version of the GNU C compiler which has been modified to recognize the new MPL features, and to output information about the resources used by the MPL program.

- The Maruti Configuration Language (MCL). MCL allows the system designer to specify the placement, timing constraints, and interconnections of all the modules in an application. MCL is a powerful interpreted C-like language, allowing complex, hierarchical configuration specifications, including replication of components and installation-site specific sizing of the application. The MCL processor analyses the application graph for completeness, and type-checks all connections.

- The Maruti Allocator/Scheduler. The Maruti allocation and schedulingtool analyses the information generated by the MPL compiler and the MCL integrator to find an allocation and scheduling of the tasks of a distributed application across the nodes of a Maruti network. All relative and global timing, exclusion, and precedence constraints are taken into account in finding a schedule, as are the network speed and scheduling parameters.

- The Maruti Timing Trace Analyzer. The Timing Analyzer calculates worst-case computation times from timing files output by the runtime system. Computation times are calculated for each scheduling unit in the application, and these times can be fed back into the Allocator/Scheduler for more precise scheduling analysis.

- The Maruti Runtime Binder (mbind). One of the features of Maruti is the late binding of an application to a particular runtime system. The same application binaries can be combined with different system libraries to build a binary customized for a particular application in a particular setting. Only those portions of the system library needed for that binding are included. Mbind manages this final step.

- The Maruti Application Builder (mbuild). Mbuild automates the process of building an application by generating for the programmer a customizable makefile that manages the complete process of compiling, configuring, scheduling, and binding an application.

DISTRIBUTION LIST

AUL/LSE Bldg 1405 - 600 Chennault Circle Maxwell AFB, AL 36112-6424	1 cy
DTIC/OCP 8725 John J. Kingman Rd, Suite 0944 Ft Belvoir, VA 22060-6218	2 cys
AFSAA/SAI 1580 Air Force Pentagon Washington, DC 20330-1580	1 cy
PL/SUL Kirtland AFB, NM 87117-5776	2 cys
PL/HO Kirtland AFB, NM 87117-5776	1 cy
Official Record Copy PL/VTS/Capt Jim Russell Kirtland AFB, NM 87117-5776	2 cys
PL/VT Dr Hogge Kirtland AFB, NM 87117-5776	1 cy

DEPARTMENT OF THE AIR FORCE
PHILLIPS LABORATORY (AFMC)

30 Jul 97

MEMORANDUM FOR DTIC/OCP

FROM: Phillips Laboratory/CA
3550 Aberdeen Ave SE
Kirtland AFB, NM 87117-5776

SUBJECT: Public Releasable Abstracts

1. The following technical report **abstracts** have been cleared by Public Affairs for unlimited distribution:

PL-TR-96-1126, Pt 1	ADB222369	PL 97-0685 (clearance number)
PL-TR-96-1126, Pt 2	ADB222192	PL 97-0685

2. Any questions should be referred to Jan Mosher at DSN 246-1328.

Janet E. Mosher

JANET E. MOSHER
Writer/Editor

cc:
PL/TL/DTIC (M Putnam)