INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A fast, effective scheduling framework for parallel computing systems

Blake, Ben Arthur, Ph.D.
The Ohio State University, 1990
A FAST, EFFECTIVE SCHEDULING FRAMEWORK FOR PARALLEL COMPUTING SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Ben A. Blake, B.A., M.S.

* * * * *

The Ohio State University

1990

Dissertation Committee

Ten-Hwang Lai
Karsten Schwan
Neelam Soundararajan
P. Sadayappan

Approved by

[Signature]
Adviser
Department of Computer and Information Science
To my family
ACKNOWLEDGEMENTS

First and foremost, I express thanks to my family, without whom none of this would be possible. To my parents for not teaching me that things are impossible and to my in-laws for their continuing support. Of course, I am greatly indebted to my wife for her constant reassurance and belief in my abilities. My thanks also goes to my daughter Bailey who frequently relieved my frustrations.

I would like to specially thank Dr. Karsten Schwan for his guidance throughout this research. Thanks also goes to my other committee members, Drs. Ten-Hwang Lai, Neelam Soundararajan, and P. Sadayappan for their comments and patience (particularly with the three different dates for the final defense). I would like to thank all of the graduate students who greatly assisted me during my stay namely Drs. Win Bo, Prabha Gopinath, Mike Kaelbling, and Dave Ogle. The latter two helped me realize I had the potential to graduate. I also appreciate the other graduate students involved in the noon time research (especially the summer sessions with my officemate, Mike Weintraub et. al.).

I would like to finally thank all of the CIS office staff (Chris, Ernie, Jane, Judy, Marty, and Tom) who assisted me numerous times during my endeavors.
VITA

February 2, 1961 ......................................................Born — Ravenna, Ohio

1983 .................................................................B.A. Mathematics
Wittenberg University
Springfield, Ohio

1983-present ......................................................Graduate Teaching Assistant
Computer Science Department
The Ohio State University
Columbus, Ohio

1985 .................................................................M.S. Computer Science
The Ohio State University
Columbus, Ohio

Publications


Fields of Study

Operating Systems: Prof. K. Schwan
Theory of Automata: Prof. Y. Yesha
Programming Languages: Prof. N. Soundararajan
# Table of Contents

DEDICATION .................................................. ii  
ACKNOWLEDGEMENTS ................................. iii  
VITA ................................................................. iv  
LIST OF FIGURES ........................................ ix  
LIST OF TABLES ........................................... xii  

CHAPTER  PAGE  
I INTRODUCTION ............................................... 1  
II BASIC ASSUMPTIONS ................................. 15  
2.1 Experimental hardware .......................... 15  
2.2 Migration types ........................................ 17  
2.3 Migration and message communication ....... 19  
2.4 Classes of applications ............................. 19  
   2.4.1 Minimal completion time applications ... 20  
   2.4.2 Deadline applications .......................... 21  
   2.4.3 Workload generator ............................. 24  
2.5 The complexity of static scheduling ......... 26  
2.6 The complexity of optimal dynamic scheduling 31  
III THE SCHEDULING FRAMEWORK ................. 34  
3.1 The local schedulers ............................... 39  
3.2 The global scheduler(s) ........................... 44  
3.3 Links ....................................................... 46  
3.4 Evaluation of the scheduling framework ...... 48  
3.5 The framework's efficiency .................... 52  
3.6 Summary ............................................... 53  

vi
List of Figures

1  Impossibility of dynamic optimal scheduling .................. 32

2  Two global and three local schedulers. ......................... 36

3  The offer data structure and queues. ......................... 58

4  Worst case scheduling overhead in a 10 processor average 10 bin each system. ............................................. 76

5  Worst case scheduling overhead in various systems with tasks average execution time of 100 milliseconds. ............... 77

6  Worst case scheduling overhead in various systems with tasks average execution time of 1 second. ..................... 78

7  Worst case, best case, and measured overheads: 5 processors, 10 bins, 100 millisecond average execution times. .............. 79

8  Comparison of offers to bidding using messages and shared memory. 82

9  Worst case percentages of scheduling requests meeting deadlines. 85

10 Scheduling effectiveness measured for generated workloads. .... 88

11 The ordering of the global queue affects completion time. ...... 94
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Worst case scheduling behavior for execution-time-only applications no task migration.</td>
</tr>
<tr>
<td>13</td>
<td>Average case scheduling with 4 processors and no mid-execution migration.</td>
</tr>
<tr>
<td>14</td>
<td>Average case scheduling with 100 processors and no mid-execution migration.</td>
</tr>
<tr>
<td>15</td>
<td>Sample optimal schedule.</td>
</tr>
<tr>
<td>16</td>
<td>Average case scheduling with mid-execution migration on a 4 processor system.</td>
</tr>
<tr>
<td>17</td>
<td>Average case scheduling with mid-execution migration on a 100 processor system.</td>
</tr>
<tr>
<td>18</td>
<td>Scheduling trees.</td>
</tr>
<tr>
<td>19</td>
<td>Worst case scheduling overhead for deadline schedulers with different average execution times.</td>
</tr>
<tr>
<td>20</td>
<td>Worst case scheduling overhead for deadline schedulers with varying number of bins and resources.</td>
</tr>
<tr>
<td>21</td>
<td>Worst case, best case, and measured overheads in a 32 processor system with 10 bins per processor and 1 second average execution time.</td>
</tr>
<tr>
<td>22</td>
<td>Measured effectiveness for various loads.</td>
</tr>
</tbody>
</table>
23 Measured effectiveness for various loads with multiple cube sizes . . . 129
24 Measured effectiveness for various loads with multiple cube sizes . . . 129
25 Measured effectiveness for various loads with multiple cube sizes . . . 131
26 Measured effectiveness for various loads with multiple cube sizes
   with mid-execution migration . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
27 Measured effectiveness for number of tasks with multiple cube sizes. 137
28 Measured effectiveness for number of tasks with multiple cube sizes
   with mid-execution migration. . . . . . . . . . . . . . . . . . . . . . . . . 138
List of Tables

1. Comparison of different processing systems. .............................. 16

2. Worst case percentages of deadlines met on the multiprocessor. ........ 153

3. Measured percent of deadlines met on the multiprocessor. .............. 153

4. Completion time on a 4 processor multiprocessor with processes ranging in time from 200-400 units. .............................................. 154

5. Completion time on the multiprocessor with 100 processors and processes ranging in size from 200-400 time units. ............................. 154

6. Completion time on the multiprocessor with 50 processors and processes ranging in size from 200-400 time units. ............................. 155

7. Completion time on the multiprocessor with 25 processors and processes ranging in size from 200-400 time units. ............................. 155

8. Completion time on the multiprocessor with 10 processors and processes ranging in size from 200-400 time units. ............................. 155

9. Completion time on the multiprocessor with 10 processors and processes ranging in size from 10-100 time units. .............................. 156
10 Measured percent of deadlines on the cube for various loads and
process types. ................................................. 156

11 Completion time on the 32 node cube with random subcube sizes
and task lengths between 5-20 time units. .......................... 157
CHAPTER I

INTRODUCTION

In recent years, multicomputers and multiprocessors have become increasingly popular. The processing speed of these machines potentially surpasses that of conventional computers by the equation \( P = \sum_{i=1}^{n} P_i \) where \( P \) is the speed and power of the total multiprocessor, \( n \) is the number of processors, and \( P_i \) is the speed and power of each single processor. Theoretically, each doubling of processors within a computer system should create a machine twice as fast (i.e., one that can solve a given application in half the time).

Parallel applications for such computing systems attempt to harness the computing system's increased power by concurrently executing various application components on different processors. This thesis focuses on efficiently executing parallel programs by judiciously scheduling them on target parallel machines. Such parallel scheduling concerns both the assignment of application components to processors and the scheduling of assigned components on individual processors. Issues related to the design of such parallel applications are not addressed in this
dissertation.

The scheduling framework presented in this thesis formulates schedules based on user provided constraints and objective functions, which include:

- The objective of minimizing the completion time of an entire application.

- The objective of duplicating various application components for increased reliability.

- The objective of utilizing various hardware components efficiently (e.g., using processors with floating point co-processors for software components performing floating point operations).

- The constraint of ensuring that certain segments of the application are completed by specified times.

- The constraint of upholding precedence relationships between components.

A given application may have several scheduling objectives and constraints to be satisfied simultaneously (e.g., execute component A after component B while completing A by time T with precision Q and reliability R).

The scheduling framework developed in this thesis emphasizes objectives and constraints that are time related. Namely, applications are scheduled with any number of user defined objectives and constraints subject to the restriction that one of the objectives or constraints involve time.
Assumptions. The scheduling framework defined makes the following assumptions about the criteria that drive the scheduler:

- Must be computed by the application program.
- Must contain an ordering.
- Must remain stable, that is, an application cannot alter objectives and constraints used by the scheduler during execution.

All inputs the scheduler receives are assumed to be computed by the application program. This assumption implies that any degradation of performance due to the collection and calculation of scheduling inputs is absorbed by the various schedulable components of the application. This is appropriate since the application programmer decides the inputs to the scheduling framework. Some important terms used when describing the scheduling framework are presented next.

Definitions. Throughout this text, a process represents a single schedulable element requiring a single processor for execution. A task is a single process or a group of processes that jointly define a schedulable unit. Each task contains all the scheduling information required by the scheduling framework.

Processes can either be migratable or non-migratable depending on its resource requirements and its execution time. Once a non-migratable process starts execution it may not be moved from the processor on which it started execution. A
migratable process may be moved from one processor to another after it has commenced executing.

The set of tasks to be scheduled can be static or dynamic. With a static task set, all scheduling information is available before any task starts execution. Thus an off-line scheduling algorithm can be used to construct static schedules. Dynamic task sets, however, continually generate new processing requirements (i.e., tasks to be scheduled) based on differing inputs. Such dynamic task sets require on-line scheduling algorithms. The algorithms presented in this thesis primarily concern dynamic task sets for three reasons:

- The scheduling of dynamic task sets remains ill-understood.

- Dynamic algorithms can be used to schedule both static and dynamic task sets.

- Dynamic algorithms must be time and resource efficient.

Applications. Dynamic scheduling is essential for large class of parallel applications, termed dynamic applications. Such applications include real-time programs that cannot be scheduled off-line because they may execute in highly variable operating environments. Namely, task arrival times and computational requirements are not statically known.

We show that optimal schedules do not exist for dynamic applications when
instantaneous\textsuperscript{1} process migration is not possible. This implies that dynamic scheduling requires the development of effective heuristic procedures. Furthermore, since dynamic applications operate with limited computational resources, it implies that the heuristics be efficient (i.e., not be computationally complex or otherwise resource-intensive). A simplifying assumption in our work derives from the fact that the dynamic applications considered in this thesis exhibit comparatively low bandwidths of inter-task communication, so that scheduling decisions need not explicitly consider communication.

Most dynamic applications fall into one of the three following categories:

- Applications with a single scheduling goal attached to the entire application, which we term \textit{completion-time based applications}.

- Applications where every task of the application has to be completed by a certain time, which we term \textit{deadline applications}.

- Applications containing a mixture of both.

Completion-time based applications desire to be scheduled to complete execution in minimal time. This requires that the scheduler receives values quantifying the execution times of the application's tasks (these values can either be estimates or precise values). Given a set of tasks, an optimal completion-time scheduler allocates tasks to processors enabling the application to complete execution as early

\textsuperscript{1}No processing system to date can migrate a process without consuming processing time.
as possible.

A parallel implementation of the traveling salesperson problem is the example of the completion-time based application type used in this thesis[50]. In our particular implementation, the task of deriving a solution is divided into disjoint processing problems. Each of these can be further divided when processors become idle. This process continues until the optimal solution to the problem is found.

Schedulers for the deadline type of application must receive values for task deadlines and upper bounds on the execution times for the processes of a task. An optimal deadline scheduler schedules all tasks to complete execution before their deadlines whenever possible[11].

The operating software of the Adaptive Suspension Vehicle (ASV) is an example of a deadline application. The ASV is a six-legged walking machine with operating software in which various processes which must be completed within strict time limits[33, 32, 34]. If such deadlines are not met, the machine may fail and either physically break down due to extreme stress, or roll over from lack of balance. Even though the processes produce correct values, the application fails because the mechanical components of the machine do not receive the values in a timely manner.

The performance of a particular scheduler depends on two factors: (1) its effectiveness and (2) its efficiency. The effectiveness of the scheduler is measured according to the constraints and objective functions supplied by the application
programmer. In general, we determine effectiveness by comparison of computed schedules with an optimal schedule along with the scheduler’s best and worst case performance. For example, if deadline scheduling is used, then the effectiveness of the scheduler is measured as the percentage of processes finishing execution prior to their deadlines. If minimum completion time is used, then the measure of effectiveness is the computed schedule’s closeness to the minimal processing time required by the application.

A scheduler’s efficiency is measured as the latency with which it makes scheduling decisions. For static scheduling, scheduling efficiency may be measured by the asymptotic time complexity of the scheduling algorithm. Efficiency of a static scheduler is not extremely important because the scheduler is executed to completion before the application is executed. Thus, the static scheduler’s processing does not interfere with the application’s execution. Dynamic schedulers operate during the application’s execution. Therefore, the execution time of a dynamic scheduler must be added to the execution time of the application. Here, efficiency is important because the resulting scheduling overhead should be sufficiently low so that the application benefits from using the scheduler.

Dynamic scheduling cannot be efficient unless the scheduler takes advantage of the available hardware. Specifically, dynamic scheduling for multiprocessor and multicomputer systems cannot be performed efficiently with a single, sequential scheduler. Instead, scheduling must be performed in parallel and the scheduler’s
multiple components must be mapped to the nodes of the underlying parallel hardware such that they efficiently compute effective assignments and schedules for the target parallel applications. Thus, in general, scheduling effectiveness and efficiency requires a high degree of scheduler flexibility, which is defined as its ease of (1) adaptation to different underlying parallel architectures and (2) use of different scheduling heuristics (and measures of effectiveness) for various classes of applications.

The flexibility of the scheduling framework developed in this thesis is demonstrated by mapping it to both a multiprocessor and a multicomputer architecture. Flexibility is also displayed by implementing several different schedulers for use with each architecture. Schedulers are shown efficient and effective by experimentation with sample parallel real-time (and non-real-time) applications. In addition, a probabilistic task generator is used to create sample workloads for the various classes of applications.

Flexibility of the scheduling framework can also be shown by its use for the scheduling of other resources in conjunction with CPU scheduling. We explicitly address the issue of scheduling other resources, such as disks, memory, etc., in section 4.1.5.

Finally, flexibility is exhibited by accommodating processing requests with multiple criticalities. One definition of criticality in a deadline application is that the task with the earliest deadline is the most critical. Task criticalities are considered
at two separate occasions. First, when more than one task awaits scheduling, the scheduling framework attempts to schedule the task with the highest criticality first. Second, when more than one task is eligible for execution, the framework executes the task with the highest criticality. Thus, the framework attempts to schedule and execute tasks with earlier deadlines before tasks with later deadlines.

**Related Research.** Most research regarding process assignment and scheduling for multiple computer systems concerns the use of straightforward scheduling mechanisms[26, 18] coupled with static processor assignment algorithms, where the latter attempt to balance processor loads while minimizing expected interprocess communication[43, 35, 49, 29, 30, 9]. Similarly, algorithms for group process scheduling, such as described in [39], support the co-scheduling of groups of processes that communicate highly and therefore, should execute simultaneously. Such algorithms cannot schedule arbitrary parallel applications, such as dynamic robotics applications in which low overhead scheduling decisions must be made on the fly and the amount of interprocess communication is typically small[38].

Other algorithms concerning process assignment and scheduling attempt to balance the processor[17, 62, 15, 1, 8] or system[51] loads. These algorithms are typically computationally complex and often suboptimal. In fact, it has been shown that they may yield only slightly better and sometimes worse results than less complex scheduling methods[14, 21]. Since many real-time applications have strict deadlines and require low-latency scheduling decisions, the algorithms cited
above are not suitable for dynamic applications.

Scheduling work based on value functions, reward functions, or criticalities are presented in [31], [54], and [3], respectively. In these cases, each process has an associated 'value' function. This function's range is time and domain is positive real numbers. It indicates the process' 'value' given the completion time of the process. An optimal schedule maximizes the sum of the value functions of all scheduled processes. However, a systematic method for the derivation of these functions is not available. Again, this scheduling model is useful for only a limited number of real-time applications and is not appropriate for use with applications of interest to us, because the processes of our applications have no associated value functions.

The rate monotonic scheduling system in [10, 56, 40, 24, 55, 52] is based primarily on scheduling periodic tasks. While provisions to accommodate non-periodic tasks help to enhance this system, it still lacks the generality to effectively schedule applications with few or no periodic tasks. Secondly, all tasks scheduled with the rate monotonic schedulers must have deadlines. These facts indicate that rate monotonic schedulers are useful in some instances, but are too restrictive to effectively schedule the parallel applications we are interested in scheduling.

A comparison of our dynamic scheduler's efficiency to that of bidding algorithms first developed for computer networks by Stankovic[57, 60, 7, 59, 41, 37, 6] appears in section 4.1.5. In contrast to our work, the bidding (and focused address-
(ing) scheduling methods have been shown useful for deadline-based applications; their use for other applications has not been discussed in the literature, thus these schedulers cannot be used with all application classes we describe and schedule.

Finally, implemented real-time operating systems appear in [63] and [53]. The ARTS operating system has limited use as a real-time scheduler because it lacks facilities for mid-execution task migration – an essential feature required by many real-time applications. The MARUTI operating system contains facilities for scheduling only static deadline applications. Thus, they both lack the generality to be used with real-time and parallel application classes we are interested in scheduling.

**Problem and Thesis Statement.** A single, sequential scheduler cannot perform task assignment and scheduling for multicomputers with suitable speed and overhead with a single, sequential scheduler. Instead, to take advantage of the machine's processing power, scheduling must be performed in parallel which implies that the scheduler's multiple components require mapping to the nodes of the underlying parallel hardware. In addition, dynamic applications for parallel systems cannot be scheduled optimally under reasonable assumptions (see section 2.6), such applications must be scheduled using heuristics, which must themselves be efficient since their processing time detracts from the processing time available to the application being scheduled.

The performance of a dynamic scheduler is not only determined by the efficiency
of its scheduling heuristics. In addition, the heuristics must compute effective assignments and schedules for the target parallel application programs. Ineffective schedulers poorly assign the application's components to the computer system.

It is the primary thesis of this work that dynamic schedulers cannot offer high performance unless they exhibit both high efficiency and effectiveness. Furthermore, efficiency and effectiveness cannot be attained without a high degree of scheduler flexibility, which may be measured as its ease of:

- adaptation to different underlying architectures (e.g., parallel architectures),
- use with different operating system primitives (e.g., different types of process migration and communication subsystems),
- implementation of different application-provided scheduling heuristics for and within various classes of applications, and
- incorporating different application-defined constraints (e.g., process groupings and precedence relations).

We prove this thesis experimentally by the development and evaluation of a scheduling framework that is flexible enough to use in conjunction with different system architectures, operating systems, scheduling objectives and scheduling constraints. We implement and measure novel scheduling heuristics within the framework to verify their effectiveness and efficiency. The resulting dynamic schedulers
are shown to outperform other implemented dynamic schedulers and to be highly
effective for the application classes investigated. In addition, limitations are shown
to exist regarding the benefits of dynamic scheduling due to the cost of task migra-
tion and scheduling overhead. Specific results concern the effects of mid-execution
task migration on scheduler effectiveness. Namely, the implemented schedulers
caus ed performance degradation in applications consisting of tasks with average
execution of under 100 milliseconds on the experimental multiprocessor, and tasks
with average execution of under 400 milliseconds on the hypercube.

**Thesis Overview.** We present a framework for the construction of parallel,
real-time schedulers. The framework's flexibility is demonstrated by mapping it to
both a multiprocessor and a multi-computer architecture. The multiprocessor is
an Intel 8086 Multi-bus \(^2\) based system. Each processor node contains both local
and global memory. The multicomputer is a 32 node Intel 80286-based hypercube.
Flexibility is also displayed by building scheduling systems for deadline-based and
completion-based applications. The resulting on-line schedulers are shown efficient
and effective by experimentation with sample parallel real-time (and non- real-
time) applications. Efficiency is measured as the latency with which scheduling
decisions are made. The measurement of scheduling effectiveness is dependent on
the class of target application. Scheduling effectiveness is evaluated by comparing
the algorithms' computed schedules with worst case and best case schedules.

---

\(^2\)Multi-bus is a trademark of the Intel Corporation.
Chapter 2 presents an in-depth description of the experimental hardware, further characterizations of the sample applications, arguments concerning the difficulty in developing optimal schedules, and a list of operating system issues involved with scheduling. The generic scheduling framework is described in chapter 3. Chapter 4 discusses and evaluates implementations of schedulers and task migration on the multiprocessor system along with evaluating some straightforward extensions to the scheduling framework. Chapter 5 describes and evaluates implementations of the framework and task migration on the hypercube multicomputer. Conclusions and future research are presented in the final chapter.
CHAPTER II
BASIC ASSUMPTIONS

This chapter describes the basic assumptions of this research: (1) the hardware for which the scheduling framework is developed, (2) assumptions regarding process migration, (3) message forwarding when process migration occurs, (4) the types of applications being scheduled, and (5) the complexity of scheduling dynamic applications.

2.1 Experimental hardware

The machines used in this research are a Multi-bus\textsuperscript{1} Intel 8086-based multiprocessor system and the Intel 80286-based hypercube. The two machines are chosen to demonstrate that the scheduling framework is effective and efficient on both a shared memory machine and a non-shared memory machine. Generalizations to other MIMD architectures are straightforward.

The effectiveness of using process migration in scheduling depends largely on the latency of process migration. The two computing systems used in this research

\textsuperscript{1}Multi-bus is a trademark of the Intel Corporation.
transmit large messages (1 kilobyte) at approximately the same rates. The multiprocessor sends short messages at a much faster rate. The reason the hypercube can send larger messages at about the same rate as the multiprocessor is that the Intel 8086 processors used within the multiprocessor are slower than the 80286 processors used in the hypercube. Comparative speeds of the processors, message transmission, and process migration are depicted in table 1.

The multiprocessor system is typical of embedded real-time machines. It includes even processors each with local and global memory. All processors share a 10 megabyte Multi-bus. The GEM[47] operating system is used. GEM, as other real-time control systems, lacks mechanisms for memory deallocation which implies that process migration can only be accomplished by process duplication. Thus, duplicate copies of each process's code, data, and stack segments are maintained on each processor. Other types of migration described in the following section can be simulated by also copying the code segment over the identical code segment when migrating a process.

The Intel hypercube consists of 32 nodes connected by 10 megabyte Ethernet
serial lines. The iPSC\textsuperscript{2} operating system, which allows for both relocatable code and deallocation of memory, is used as the kernel operating system for the scheduler. In addition to process duplication, three other types of process migration can be used in conjunction with this operating system. These three types of migration are termed full migration, partial migration, and lazy migration and are described in the following section. The flexible scheduling system described in chapter 3 allows the application to choose the appropriate method of migration on a task by task basis.

2.2 Migration types

Duplication. Process duplication is the migration method with the smallest runtime overhead. In this scheme, a copy of the process's code, stack, and data segments is duplicated on processors willing to execute the process. In this case, a process migration simply consists of copying the dynamic data segments, stack, process control block, and state information of the process from the processor it last executed on, termed the source processor, to the target processor.

This is the only type of migration possible when the operating system does not provide memory deallocation. If memory cannot be deallocated, processes retain their memory on processors after they have been migrated. Eventually, when enough processes have been migrated to and from the various processors, Version 2.0 is used.
the processors run out of memory. The following types of process migration are possible only when memory deallocation is available.

**Full Migration.** In full migration, processes are migrated by moving all of the code, stack, and data segments of the process. A new process is created at the target processor and the process at the source processor is removed by deallocating its memory. This creates a larger initial overhead for each migration than any other migration schemes because of the massive data transfer involved.

**Partial Migration.** A process can be partially migrated by moving some of its segments from the source to the target processor. This is possible when some of the process's segments already exist on the target node such as a pre-duplicated code segment. Thus, partial migration involves copying only missing segments of the process to the target processor, followed by the deallocation of some of the segments on the source processor.

**Lazy Migration.** Lazy migration entails the initial migration of solely the essential segments of the process[64]. All code, data and stack segments currently being used by the process are migrated. The process at the source node can only be partially deallocated because the segments that remain and haven't yet been migrated may be needed in the future. Each segment is individually deallocated after it has been migrated.

This method creates less initial overhead than full migration, due to the smaller initial amounts of data being transferred. Depending on the number of segments
that require transferal over the process's entire execution interval and the message latency per message size, lazy migration can outperform, perform equal to, or perform worse than full migration.

2.3 Migration and message communication

When processes have open communication channels, messages sent across such channels may have to be re-routed when processes are migrated. One solution to this problem is to leave a ghost process at the source processor that forwards the process's messages for the target processor. Another solution is to broadcast the process's new location to all processors so that messages to the migrated process will be sent to its new destination. Yet another solution is a combination of the above two solutions where (1) a ghost process is left behind to forward each message and (2) the ghost process also notifies the processor that sent the message of the process's new location. The scheduling framework presented in chapter 3 allows the programmer to choose the message forwarding method appropriate for the application.

2.4 Classes of applications

The scheduling algorithms presented in chapters 4 and 5 of this dissertation are developed for two classes of applications: (1) minimal completion time applications and (2) deadline applications. Two different classes of applications are chosen to
display that the scheduling framework is sufficiently general to be used in conjunction with many types of applications.

The difference between the two classifications is the information made available to the scheduler by the schedulable entities. While the descriptions of the scheduler implementations in chapters 4 and 5 focus on these two classes, other application-specific measures, such as value functions[31], may be used with the scheduling framework. Furthermore, the scheduling framework presented in chapter 3 allows the application programmers to fine tune the scheduler for their specific needs.

2.4.1 Minimal completion time applications

The traveling salesperson application[50] for the hypercube, referred to as the TSP application, uses a divide and conquer heuristic to find an optimal solution quickly. The application consists of processes that continually spawn new processes to assist in the search of an optimal solution. Such processes should be scheduled so as to complete the application's execution in minimum time.

In order for a scheduler to be effective, the application must provide a prediction of execution time needed by each task to it. Applications providing such estimates can further be divided into two subclasses:

1. Applications consisting of tasks for which the execution time is exactly known.

2. Applications consisting of tasks for which the execution time is only approx-
imately known.

This class of applications is called minimum completion time applications throughout the text. Since this class consists of dynamic applications, the scheduler needs to operate with small latency. Namely, a lower bound for the completion time of any application of this class is computed by (1) summing the execution time required by all processes of the application and the processing time required by the scheduler and (2) dividing this sum by the number of processors in the computer system. This bound grows linearly with the scheduler's processing time.

2.4.2 Deadline applications

The ASV (Adaptive Suspension Vehicle) application is an example of another class of applications referred to as deadline applications. The ASV is a six legged walking machine driven by an Intel 8086-based multiprocessor system [32, 34, 33] described in section 2.1. This application consists of various software components that cooperatively plan and control the vehicle's operation. The system contains sensors that measure the land grade, monitor the balance of the vehicle, and provide feedback on the position of each leg. Processes are assigned to each leg and sensor in conjunction with high-level motion planning processes. The following characteristics of the ASV and other dynamic, high-performance, real-time applications determine the basic attributes required of any scheduler to be useful for such applications:
• Hard and soft deadlines. The ASV and other realistic real-time applications require scheduling support for both hard and soft deadlines. A *hard deadline* implies that the process cannot be executed unless its completion by the deadline can be guaranteed. A *soft deadline* implies that execution should but need not be completed by the deadline. In the ASV application, a process driving an actuator (e.g., a leg actuator) has a hard deadline, since a single missed deadline might cause severe stresses on the legs and possibly damage them. However, a missed, soft deadline for a periodic process performing higher level control simply implies the unavailability of updated outputs to lower level control processes. The occasional absence of such outputs does not cause instability of the ASV robot. Thus, higher level processes may be scheduled even if the deadline cannot be guaranteed.

• Scheduling of burst, cyclic, and groups of processes. The ASV's control software consists of multiple processes, some of which should be scheduled so as to guarantee their parallel execution. Each of those processes may be burst or cyclic. *Burst processes* execute once upon being enabled, and are useful for handling exceptional conditions. *Cyclic processes* execute every $X$ time units depending on their rate. *Group processes* demand to be executed together and are further discussed in section 3.4.
In the ASV and deadline applications, each process has a known maximal amount of execution time and a strict deadline. Execution times must be bounded precisely so that deadlines of previously scheduled and subsequently arriving processes can be guaranteed. Such tight bounds may not be attainable without pre-execution of code, or control and data flow analysis. The cost of attaining the process's worst-case execution times is not included in any of the presented models.

Some deadline applications establish deadlines, execution time limits, and release times. This is the case if a process requires execution during a specific interval of time. The beginning point of the interval, termed the process's release time, represents the first time the process is ready to execute. This time indicates the moment at which the information to be used by the process becomes available and/or correct. Namely, the results of the process may be unreliable if it commences execution before its release time. The end point of the interval is the process's deadline.

In the implemented deadline scheduling systems of chapters 4 and 5, the goal of the systems is to schedule all tasks before their deadlines. Schedulers which maximizes performance an overloaded system[36, 51, 31] are useful in other paradigms, but do not satisfy the scheduling criteria examined in this thesis.
2.4.3 Workload generator

Since only a few actual applications are available for testing the scheduling framework, a workload generator has been developed. The workload generator develops tasks for both deadline and minimal completion time applications. All tasks of each application class discussed above provide the scheduler with a worst case execution time. The distribution of task arrivals and execution times are application-dependent. Deadline applications also provide the scheduler with task release time and deadlines. The following are parameters to the workload generator to produce varying sample workloads for the implemented schedulers:

- Average task execution time.

- Task execution time distribution.

- Task arrival rate.

- Percentage of total processing time used by all tasks.

Different distributions, task sizes, utilizations, and arrival rates affect the effectiveness of the scheduling system.

- Average number of deadlines.

- Distribution of deadlines.

- Average number of release times.
• Distribution of release times.

A larger number of deadlines and release times increase the information available to the scheduler and, depending on the scheduler, may influence its efficiency.

• Maximum number of processors required by each task.

• Distribution of the number of processors required by a task.

These parameters are used only with the scheduler developed for the hypercube. Varying the size and distribution of the subcubes required by tasks influences the effectiveness of the scheduler.

The processing time of the workload generator is ignored when measuring the efficiency of the scheduler. Workloads are either generated statically or dynamically. When workloads are generated dynamically, an unused processor is used for the workload generator, so that the generator’s processing time does not interfere with measurements of the scheduling efficiency.

The workload generator attempts to generate only feasible task sets for deadline applications (all task sets are feasible for minimum completion applications). When the generator calculates the next request it views all previous requests and verifies that all the requests are likely to be scheduled by an optimal scheduler. This is accomplished through two fairly simple tests. The first test verifies that the execution time of the task is smaller than the interval of time that the task is eligible to execute (i.e., the process’s deadline - release time > execution time and deadline
- current time > execution time). The second check involves the total available execution time of the hardware system. The generator uses a self built estimate of the maximal available processing time during the feasible process execution interval. It uses this estimate to ensure that there is enough processing time available for the generated process and the previously generated processes during the interval (e.g., if the processing system consisted of two processors and the generator had previously generated two processes with release times of 5, execution times of 5 and deadlines of 10, it would not generate a process with release time of 6, execution time of 2 and deadline of 9). If either test fails, then the task is not presented to the scheduling system and another task is generated randomly. Note that (1) this method removes many impossible scheduling requests and (2) never removes a feasible scheduling request.

The workload generator uses a random number generator to create the distributions. To minimize controversy over whether random numbers can be generated, experiments are executed using different random number generators as well as different seeds with the same generator. The results of the experiments showed no statistical difference using the various random number generators and seeds.

2.5 The complexity of static scheduling

Let us now examine the problem of statically scheduling concurrently executable, independent processes in a multiprocessor system where the effect of interprocess
communication time are not considered. Scheduling processes on a single processor system can be guaranteed optimal by using one of various simple strategy depending on the application's objective functions and/or constraints (i.e., earliest deadline first, execute any process, etc.). With as few as two processors, the scheduling problem reduces to the bin packing problem when process migration is not available. Bin packing is known to be NP-complete, so scheduling processes on multiple processor systems requires good heuristics.

When a no-cost mid-execution process migration facility is available, static optimal scheduling algorithms exist for minimal completion time applications[16]. This method used computes the total required execution time by summing the execution times of the application's processes and dividing this sum by the number of processors in the system. The execution time of the longest process is compared to this value. The maximum of these two numbers, referred to as $X$, represents the minimal amount of processing time required to complete the processes. Once obtaining the value for the minimal amount of processing time needed to execute the processes, the processes are scheduled in any order.

Each processor has $X$ time units available to execute processes. The first process is scheduled to execute for the first time units on the first processor. The second process is either completely scheduled during the earliest unused processing time on the first processor, or scheduled for the remaining time available on the first processor, and scheduled for the first time units on the second processor.
This continues iteratively until all processes are scheduled. The algorithm can be extended to handle the case of dynamically arriving new processing requests while continuing to operate optimally.

Similar methods have been developed for static scheduling of single process tasks with deadlines or release times[45, 44, 46, 23], while no non-exponential algorithm has been developed for the static scheduling of processes with both deadlines and release times. These algorithms provide good theoretical results and lower bounds, but in every current multicomputer, some execution time is required for process migration. When considering this factor, the above cited scheduling algorithms are no longer optimal as will be shown below.

We now consider processes scheduling when mid-execution process migration consumes execution time. To simplify subsequent computations, the amount of processing time required to migrate a process is assumed to equal one unit of processing time. In a shared memory multiprocessor system, the target processor is assumed to bear that cost. In non-shared memory machines, both processors involved in process migration bear the migration cost.

**Theorem:** Scheduling to minimize completion time is NP-complete in systems with non-zero migration cost.

**Proof:** A reduction of the equal size bin packing problem to the scheduling minimization problem is shown next. The equal size bin packing problem consists of \( N \) equal capacity \( C \) bins \( B_j \) and \( M \) elements \( E_i \) to place in these bins.
The reduction proceeds in the following order:

1. Let $SE = \sum_{i=1}^{M} E_i$ be the sum of the sizes of the elements to be packed, and let $SN = N \cdot C$ be the total capacity of the bins.

2. Let $X = SN - SE$ be the empty space available.

3. If $X < 0$, then there is no possible way to pack the bins.

4. Let $ED_i = 2 \cdot E_i - 1$ for $1 \geq i \geq M$, $ED_i = 1$ for $M + 1 \geq i \geq M + X$, and $ED_{M+X+1} = 2 \cdot C$.

5. Add a new bin $B_{N+1}$.

6. Now the problem is converted into a scheduling problem with $N + 1$ processors, $M + X + 1$ processes where process $P_i$ requires $ED_i$ units of execution time, and all processes initially reside on processor $N + 1$.

7. If the scheduling problem can be solved within $2 \cdot C$ time units, then the bin packing solution can be constructed directly from the schedule. If there is no solution to the scheduling problem, then there is no solution to the bin packing problem.

A close examination of the schedule reveals that every process in the scheduler except process $P_{M+X+1}$ is migrated exactly once. This happens for two reasons:
1. Process $P_{M+X+1}$ cannot be migrated and complete execution in $2 \cdot C$ time units (every process $P_i$ $i \neq M + X + 1$ must be migrated from processor $N + 1$).

2. If there are more than $M + X$ migrations, then the processes cannot be scheduled in $2 \cdot C$ time units (the total processing time required would exceed the amount of processing time available).

The bin packing solution is constructed from the scheduling solution by placing element $E_i$ in bin $N_j$ if process $i$ is scheduled on processor $j$. It is easily verified that there exists a bin packing solution if and only if there exists a scheduling solution for the reduced problem. □

**Corollary:** If minimal completion time scheduling with non-zero migration cost is not in P, then deadline scheduling with non-zero process migration cost is not in P.

**Proof:** Suppose deadline scheduling is in P, then there exists a polynomial time algorithm $A$ which on inputs of $E_i, D_i, M, N$ for $1 \geq i \geq M$, where $E_i$ is the execution time requires by process $P_i$ and $D_i$ is the deadline of $P_i$, $M$ is the number of processes, and $N$ is the number of processors in the computing system, will either state no solution exists (return a NULL schedule), or will present a feasible schedule. The following algorithm solves the minimal completion time scheduling problem in polynomial time using algorithm $A$. 
1. Let $MIN = M + \sum_{i=1}^{M} E_i$ compute a time by which we are assured there is a possible schedule.

2. Let $D_i = MIN$ for $1 \leq i \leq M$ set all deadlines to $MIN$.

3. Let NEW_SCHEDULE = Call A(E,D,M,N) check for a new minimal time.

4. If NEW_SCHEDULE is not NULL then $MIN = MIN - 1$; SCHEDULE = NEW_SCHEDULE; GOTO 2

5. If NEW_SCHEDULE is NULL the minimal completion time is $MIN + 1$ and SCHEDULE contains one such schedule.

\[\square\]

2.6 The complexity of optimal dynamic scheduling

The following example displays that preemptive dynamic multiprocessor scheduling has no optimal solution. Refer to figure 1 for a pictorial representation of the example. In the example, process migration is assumed to consume one processing unit at the processor to which the process is migrated. Suppose scheduling proceeds on a two processor system. At time zero, processes A and B need 4 time units each and can commence execution immediately (for reference, the beginning time is called time 0). Any scheduler must immediately schedule both processes on separate processors or the scheduler is clearly suboptimal when no other tasks arrive.
With A and B scheduled on separate processors, process C arrives at time unit 1 requiring 7 time units for execution. An optimal schedule for this situation is depicted in figure 1. Any optimal schedule for these three processes has B scheduled for at least one time unit on processor 1. Now, at time unit 5 another process, D, arrives requiring 5 units of processing time. All four processes can optimally be scheduled to complete execution by time unit 11, but no scheduler can have A, B, and C scheduled optimally before D arrives and then schedule all four processes optimally after D arrives.

The above theorems verify that optimal scheduling for dynamic applications with non-zero process migration time has no solution. The last section displayed that optimal scheduling for static applications is NP-complete. Both of these results imply that efficient and effective heuristics are needed for scheduling.
The next chapter describes a scheduling framework suitable for developing dynamic schedulers using heuristics. The framework is general enough to be used with different architectures (i.e., multiprocessors and non-shared memory machines), application classes (i.e., deadline and completion), and operating system capabilities (i.e., migration types and communication protocols). Following the framework's description, we specify scheduling heuristics to be used within the framework.
CHAPTER III

THE SCHEDULING FRAMEWORK

This chapter presents proof that a scheduling framework exhibiting the flexibility characteristics stated in the 'Thesis Statement' does exist. The described framework is shown to be sufficiently general to develop a wide range of dynamic and static schedulers that efficiently utilize multicomputers.

Two significant simplifications are made to facilitate the framework's description. They are due to the nature of many real-time and non-real-time applications, such as the ASV, robotic, and TSP applications. First, the amounts of inter-process communication are comparatively low, so that scheduling decisions need not consider ensuing communication costs. Second, infrequent communication also results in under-utilization of the communication system, so that we do not consider bus scheduling[25, 61].

In general, we define the scheduling framework as a set of components that provide a skeleton for implementation of dynamic schedulers. This skeleton consists of:
• one or more **global** schedulers that perform the assignment of scheduling requests to local schedulers, and

• one or more **local** schedulers that process scheduling requests for a single (or group of) processor(s).

Figure 2 depicts two global and three local schedulers, which interact without programmer involvement. The resulting distributed scheduler accepts scheduling requests from both the individual processors and from the host machine attached to the multiprocessor or multicomputer. Scheduling requests from the host mostly consist of activations or de-activations of entire parallel applications. In these cases, the host interacts mainly with global schedulers. Scheduling requests originating on individual processors may either be handled by a processor’s local scheduler, or by the global schedulers.

The skeleton of the scheduling system (i.e., the scheduling framework) is more precisely defined by the following informally described concurrent module. The precise implementation of the concurrent module is left to the user. It may be implemented as a set of operating system processes, a set of execution threads, or pieces of code executed by some controlling entity. A **concurrent module** shares information with the modules it encapsulates by explicit parameter passing. Concurrent modules differ from a **module** in that concurrent modules allow multiple execution threads[19] – one for each module it encapsulates. Each module represents
Figure 2: Two global and three local schedulers.
a single execution thread. A distributed type provides communication operations for modules and concurrent modules to which it is a parameter. Distributed types are used for all communication between modules. A complete description of all components and of their informal semantics used in this chapter are included in appendix A.

CONCURRENT MODULE scheduler(queue_ordering : function, multiplexor_ordering : function, task_descriptor : record, schedule_measure : function, schedule : function, relax_constraint : function, num_local : integer, arbitrate : function, num_global : integer, link_structure : record, combine : function, num_processors : integer ...

REPRESENTATION
local_queues : array [num_local] of queue(queue_ordering,...);
global_queues : array [num_global] of queue(multiplexor_ordering,...);
multiplexor : for i = 1 to num_processors
  multiplexor(...);
links : for i = 1 to num_global link(task_descriptor,
  link_structure, combine);
local_schedulers : for i = 1 to num_local
  localscheduler(task_descriptor, schedule_measure, schedule,
  relax_constraint, multiplexor, global_queues, local_queues[i],
  links);
global_schedulers : for i = 1 to num_global
  globalscheduler(task_descriptor, arbitrate, local_queues,
Global and local schedulers cooperate using distributed types in two ways: (1) through using request queues and (2) through links. All local request queues are shared, because the global schedulers must access them when assigning processes to local schedulers. The global request queues are shared so that both the application and local schedulers can send scheduling requests to the global scheduler. Both queue 'local_queues' and 'global_queues' are assumed to be globally accessible and provide the following four queue operations: (1) insert using the 'queue_ordering' function, (2) delete, (3) top, and (4) empty. The 'link' distributed type will be discussed in detail in section 3.3. Its major purpose is to coordinate the distributed scheduling information.

Queue_ordering. It is during the queuing of requests that process criticalities are considered. All request queues are ordered by process criticality using the user defined 'queue_ordering' function. Multiple scheduling requests of the same criticality are maintained in FIFO order. This insures that queued requests of higher criticality are inspected first. The different task criticalities used in this dissertation include:

- Earliest deadline first.

- Largest task processing time first.

- FIFO.
Task descriptor. The framework requires the application to provide the scheduling information (labeled ‘task descriptor’ in the concurrent module) used by the global and local schedulers. The local schedulers use task descriptors when attempting to schedule locally arriving scheduling requests, when forwarding requests to a global scheduler, when formulating responses to the global schedulers about global scheduling requests, and when queuing locally arriving scheduling requests. The global schedulers use task descriptors when broadcasting scheduling requests to the local schedulers and when arbitrating between favorable or unfavorable local responses. The next section examines the local schedulers in depth, section 3.2 describes global schedulers, while section 3.3 outlines the ‘link’ distributed type. All other parameters of the scheduling framework are discussed in the appropriate sections.

3.1 The local schedulers

The framework requires the existence of one or more local schedulers. Each local scheduler develops schedules for a non-empty set of subordinate processors. A local scheduler services scheduling requests arriving locally or scheduling requests being offered (explained in section 3.2) by a global scheduler. The local schedulers enqueue local request arrivals whereas requests received from global schedulers are not queued; they are simply inspected and accepted, rejected, answered. Further comments on request queue implementations appear in chapters 4 and 5.

Local schedulers inspect a global scheduler’s current request only when there are no local requests. Thus, heavily loaded processors are not affected by the
presence of unscheduled requests from the global schedulers. This implies that few interactions occur between local and global schedulers when system load is high, thereby reducing scheduling cost in a highly loaded system.

The schedule, schedule_measure and relax_constraints functions. To cooperate, the 'schedule_measures', 'schedule', and 'relax_constraints' functions share a data structure specified within the local scheduler module. The application programmer specifies both the shared data structure and the manner in which the functions manipulate it. The 'schedule' function is the only function that updates the system's schedule. The function is called by a local scheduler whenever scheduling a process. This ensures that the 'schedule_measures' and 'relax_constraints' functions use current load values.

The local scheduler uses the 'schedule_measures' function to check whether a request's scheduling constraints can be satisfied by the scheduler's subordinate processors. After inspecting a local request using 'schedule_measures', the local scheduler performs one of four actions depending on the individual request's type defined in its 'task_descriptor':

1. When the task's scheduling constraints can be satisfied locally, it schedules the task on one or more of its subordinate processors.

2. When the task's scheduling constraints cannot be satisfied locally and the task may be executed elsewhere, it submits the task to a global scheduler for scheduling.

3. When the task's scheduling constraints cannot be satisfied locally and the
task must be scheduled locally, it adjusts the task’s constraints and schedules it.

4. When the task’s scheduling constraints cannot be satisfied locally, the task’s constraints cannot be adjusted, and the task cannot execute elsewhere, it simply refuses to schedule the task and notifies the application program.

The 'relax.constraints' function is used to locally schedule task’s whose scheduling constraints cannot be satisfied. A soft deadline task that cannot be completely executed prior to its deadline is an example of such a task. A call to the 'relax.constraints' function adjusts the task’s constraints so that it can safely be scheduled locally with its new constraints. For a consistent schedule, the constraints must be adjusted in such a way that the task does not jeopardize the processing of previously scheduled tasks (i.e., in the case of a soft deadline task, the deadline must be adjusted so that all previously scheduled tasks still complete execution before their deadlines).

Multiplexor queues. To schedule a task, the local scheduler simply submits the various processes of the task to subordinate processors’ multiplexor waiting queues. The amount of information about each waiting task, ‘task.descriptor’, and the manner in which tasks are multiplexed, ‘multiplexor.ordering’, are parameters provided by the application programmer. The ‘multiplexor.ordering’ function maintains the ordering of the multiplexors’ ready queues. The application programmer must ensure that all queuing methods match the scheduling measures discipline for a consistent schedule. An example of unmatched disciplines is the
use of a deadline scheduling measure with FIFO multiplexing. In this example, the
deadline scheduling measure indicates to the application that all tasks will com­
plete execution before their deadlines, while the FIFO multiplexor will not ensure
that this happens. The application is notified by the ‘scheduling_measures’ func­
tion that all tasks will be executed before their deadlines, while actually they will
not be. Thus, when the ‘schedule_measures’ and ‘multiplexor_ordering’ functions
disciplines do not match. An inconsistency exists between the ‘schedule_measures’
function and the actual schedule (i.e., deadlines guaranteed, but not met).

A description of the local scheduler module is now displayed. A module in our
informal syntax is an encapsulation of a single execution thread. It is much like
an Ada[20] task with the exception that modules have no explicit entry points.

Modules communicate via distributed data types.

MODULE localscheduler(task_descriptor : record,
schedule.measure : function,
schedule : function,
relax.constraints : function,
multiplexor.queues : distributed type,
global.queues : distributed type,
local.queue : distributed type,
links : distributed type,
...);

REPRESENTATION
   info : task_descriptor;

PROCESS request_handler IS
   loop:
      if not_empty(local.queue) then
         info = top(local.queue);
         if schedule_measure(info,reply) then
            insert(multiplexor.queues,info);
            schedule(info);
         else
            if info.relax then
               relax_constraints(info);
insert(multiplexor_queues,info);
schedule(info);
else
  if info.global then
    insert(info,global_queues);
  else
    notify application;
  endif;
endif;
remove(info, local_queue);
endif;
else
  info = receive.down_link;
  if schedule_measure(info,reply) then
    send_up_link(info,reply);
    info = receive.down_link;
    if awarded(info) then
      insert(multiplexor_queues,info);
      schedule(info);
    endif;
  endif;
endif;
endloop;

The local scheduler first attempts to schedule any locally arriving scheduling requests. If a local request's scheduling requirements can be satisfied locally, then it is scheduled and the next scheduling request is considered. If not, the request's requirements may be relaxable (as in the case of a soft deadline task), which results in requirement adjustment followed by task scheduling. When the request's processing requirements cannot be met locally or relaxed, the request is forwarded to a global scheduler. Finally, if the request's requirements cannot be met locally, or relaxed, or forwarded to the global scheduler, the application is notified and the request is rejected.

If no local scheduling requests are pending, the local schedulers receive global scheduling requests through operations provided by the 'link' distributed type de-
scribed in section 3.3. When inspecting a global request, the local scheduler performs the 'schedule_measures' function using the request's 'task_descriptor'. After inspecting the global request, the local scheduler uses the 'link' distributed type to notify the global scheduler of the feasibility of scheduling the request. The local scheduler then receives a response from the global scheduler informing it of the global scheduler's decision. If the local scheduler is awarded the request, it schedules the request by submitting it to the multiplexor(s) ready queue(s), then proceeds to the next scheduling request. In many instances, such as using the 'offers' for the link distributed type, the notification to the global request and the response of the global scheduler may be empty (see chapter 4 for details). If the local scheduler is not awarded the request, it simply attempts to schedule the next request.

For each global request, the set of processors that are eligible for request assignment may be stated in the 'task_descriptor'. A local scheduler will check its own eligibility by verifying that it has processors eligible to execute the request. Local schedulers without eligible processors will not attempt to schedule the request. Eligibility restrictions are useful when processors differ in specific attributes, such as in associated floating-point co-processors or in the amounts of local memory.

3.2 The global scheduler(s)

The global schedulers dequeue global requests and then prepare and present them to local schedulers. All requests arriving at the global scheduler(s) may be offered to all local schedulers using links. Each global scheduling request is either (1)
rejected, (2) awarded to a local scheduler(s) accepting the request, or (3) awarded to the local scheduler with the most favorable response.

**Arbitrate.** The global scheduler uses the arbitration function provided by the application to determine the local scheduler to which the task is awarded. In many cases, the global scheduler can assign the task to the first local scheduler giving a favorable response. Thus, the average latency of assignments by the global scheduler is low. In this case, a potentially long wait for responses from all local schedulers is incurred only when the task demands that all returned task descriptors be examined by the global scheduler before assigning the task for execution.

A global scheduling request becomes infeasible when all local schedulers have inspected and rejected it or when it has not been accepted after some application-specified time interval. When this happens, the global scheduler reports the infeasible request to the application, then either rejects the request, or forces a local scheduler to schedule the request (even though the local scheduler can not fulfill the task's processing requirements). The 'arbitrate' function is used to select the local scheduler to which the task is sent. The task is then ensured scheduling on the local scheduler by setting the relax variable of the task descriptor to true. In certain applications, such as minimal completion applications, task rejection does not arise.

The informal definition of a global scheduler follows:

```module globalscheduler(task_descriptor: record,
                        arbitrate: function,
                        local_queues: distributed type,
                        global_queue: distributed type,
                        links: distributed type)```
PROCEDURE scheduling_request IS
  loop:
    if not_empty(global_queue) then
      info = top(global_queue);
      build_link(info);
      send_down_link(info);
      info = arbitrate(receive_up_link(reply));
      send_down_link(info);
      remove(info, global_queue);
      raze_link;
    endif;
  endloop;

3.3 Links

The 'link' distributed type is used for communicating offers and responses between global and local schedulers. Links are bi-directional, one to many broadcast channels from a global scheduler to the local schedulers and many to one return channels from the local schedulers to a global scheduler. Links, like queues, must be implemented using the communication primitives provided by the underlying operating system and are accessible by both the local and global schedulers. Links differ from queues in two ways:

1. Links may perform computations on their parameters passed as messages within their structure.

2. Links have an explicitly defined communication structure.

Structure. The parameter, termed 'link_structure', contains data that specifies the communication structure used by links. The application programmer is
responsible for the efficient mapping of the communication structure used by links to the underlying architecture of the computing system being used.

The 'link' distributed type that follows displays the operations available for the local and global schedulers.

DISTRIBUTED TYPE link(task_descriptor : record,
 link_structure : record,
 combine : function);

OPERATIONS
  build_link(task_descriptor) IS
    build a communication structure;
  raze_link IS
    destroy communication structure;
  send_down_link(task_descriptor) IS
    broadcast info to local schedulers;
  send_up_link(task_descriptor,reply) IS
    send scheduling information back to global scheduler using the combine function in intermediate stages;
  receive_down_link IS
    receive request info from global scheduler;
  receive_up_link IS
    receive scheduling information from local schedulers;

A 'link' distributed type builds a tree-like communication structure for each scheduling request. The root of the tree is the global scheduler and the leaves are local schedulers. Each intermediate node of the tree simply relays scheduling information from the global scheduler to the local schedulers during a 'send_down_link' operation. On a 'send_up_link' operation, the intermediate nodes relay the scheduling information from their children nodes toward the root.

Combine. During the 'send_up_link' operation, intermediate nodes not only forward information toward the global scheduler, but also coalesce the information received from children nodes. This is accomplished through the 'combine' function supplied by the application programmer. The intermediate nodes receive schedul-
ing information from each of their children nodes. After the final child replies with scheduling information, the node compacts the scheduling information using the 'combine' function and sends the modified scheduling information to its parent node. Use of this function effectively distributes portions of the global schedulers' processing among the processors of the computing system.

### 3.4 Evaluation of the scheduling framework

Chapters 4 and 5 verify the framework's flexibility by describing specific instances of scheduling systems generated with the framework. Specifically, chapter 4 describes schedulers for deadline and completion schedulers for multiprocessors, while chapter 5 describes them for a non-shared memory machine. All developed schedulers are measured for efficiency and effectiveness. In this section, we delineate the types of schedulers that can be constructed using the framework.

Any generic scheduling system is useful only if it can be applied to different applications and processing systems. The flexibility of the scheduling system above is apparent by its parameters. For each scheduling system implemented, the application programmer sets the number of local and global schedulers and the communication structures used between local and global schedulers. The programmer also defines the amount of information, the 'task_descriptor', that is passed through the communication structure. Finally, the programmer implements the 'schedule_measures' and 'arbitrate' functions that use information in the 'task_descriptor' to formulate schedules.

The 'task_descriptor' allows the user to specify both single and multiple process
Multiple process tasks can further be subdivided into many different classes:

1. Processes wishing to execute on the same processor, termed *clustered group*.

2. Processes wishing to execute on separate processors, termed *spread group*.

3. Processes that don't care.

4. Processes wishing to execute on the same processor as some processes and on separate processors as others, termed *mixed groups*.

5. Groups with other alternatives (i.e., priorities, precedence, etc.).

Multiple process tasks without placement preferences (category 3 tasks) are simply treated as separate single process tasks by the scheduling framework. Mixed groups are considered as clustered groups within a spread group which are described in the following paragraphs.

The global scheduler treats a clustered group as a single task with multiple processing requirements. This task has an eligibility restriction that all processing must be performed by a single processor. With this restriction, the local schedulers are then forced to schedule the whole task on a single processor.

A global scheduler performs the scheduling of each spread group in a sequential fashion. First, the group’s member processes are arranged by an application-specified ordering, then each group member is scheduled in turn. The first process is sent as a scheduling request to all local schedulers. The first process is then temporarily assigned to some local scheduler. The scheduling request for the second
process of the group is only available to the local schedulers not assigned to execute
the first task by attachment of suitable eligibility restrictions to the scheduling re­
quest. This continues until all of the processes of the group are scheduled. If not all
of the group can be scheduled, then all temporarily members are unscheduled, the
application is notified of the unscheduled group, and the next scheduling request
is broadcast.

Some application programmers demand the synchronization of the processes
composing a spread group. The scheduling framework presented in this disserta­
tion is not meant to be a synchronization mechanism. If application programmers
wish to use the scheduler and also requires synchronization, they must construct
a virtual clock and use the clock in conjunction with the 'scheduling_measures',
'combine', and 'multiplexor_ordering' functions to ensure synchronization of the
various tasks and processes. The accuracy of the clock depends solely on its imple­
mentation. Concurrent processes are presented to the scheduler with appropriate
constraints to ensure synchronization.

Likewise, precedence constraints among real-time processes can be enforced us­ing
release times and deadlines in conjunction with the above mentioned virtual
clock. Namely, if one process must execute prior to another process, then the dead­
line of the predecessor process precedes the release time of the successor process. In
the absence of exceptional conditions causing unexpected delays, this insures that
the predecessor process completes execution before the successor process (within
tolerances determined by the virtual clock).
The scheduling framework allows each scheduling request to provide information regarding fixed resources (e.g., memory) and schedulable resources (e.g., disk drives, robot manipulators) required by the request at the target processor. The scheduling of requests regarding more than one resource requires additional checking by local schedulers performed as part of the 'schedule_measures' function. For a fixed resource like memory, a local scheduler must check whether the processor has sufficient amounts of a fixed resource, and it must decrease the amount of the fixed resource by the request's requirements. For a schedulable resource like a disk, a local scheduler schedules the resource in a manner identical to the CPU scheduling method used.

To reduce scheduling latency for requests with resource constraints, local schedulers maintain threshold values with each resource (shown in appendix A). If a request considered by a local scheduler causes the threshold value of any one of its required resources to be exceeded, then the request may instantaneously be rejected. Thus, local schedulers quickly reject requests that desire overcommitted resources. The threshold values used for each resource are supplied by the application programmer.

Another flexible attribute of the framework is that the application programmer supplies a 'multiplexor_ordering' function. The multiplexor uses this function to decide which process to schedule for execution next when more than one process is concurrently eligible. The following are some sample functions used by the schedulers defined in chapters 4 and 5:
- Process with greatest remaining processing time first.

- Process with smallest remaining processing time first.

- Process with highest criticality first.

- Process with earliest deadline first.

3.5 The framework’s efficiency

The framework’s efficiency greatly depends on the scheduling functions provided by the application programmer. The following section lists the performance sensitive parameters excluding the actual scheduling functions (i.e., ‘schedule_measures’, ‘schedule’, ‘arbitrate’, etc.)

When the applications programmer specifies that large amounts of scheduling information be shared by local and global schedulers (i.e., a large ‘task_descriptor’), the ‘link’ and ‘queue’ distributed types must transfer large amounts of data. This results in performance degregation due to the larger amounts of data being transferred through the communication primitives of the underlying operating system.

The frequency at which local schedulers, global schedulers, and multiplexors operate are also performance-sensitive variables set by the application. For simplicity, the following parameters are omitted in our original description:

- ‘local_scheduler_frequency’,

- ‘global_scheduler_frequency’,

- ‘multiplexor_frequency’.
Schedulers that operate too frequently rob the application of processing time, while schedulers that operate too infrequently make poor scheduling decisions. The three frequency parameters, which must be implemented as functions, are easily included in the following manner prior to the end of each continuous loop of the local and global scheduler modules:

```plaintext
    loop:
      ..
      sleep(frequency_function);
    endloop;
```

The frequency parameters are included in the description of the scheduling framework in appendix A.

Finally, the application programmer selects the scheduling functions used by the scheduling framework. The time complexities of the various scheduling functions also affect the performance of the scheduling framework. When computationally complex arbitration, 'multiplexor_ordering', or 'schedule_measure' functions are used, the system's computation time increases.

The total efficiency of a dynamic scheduling system depends on two factors: (1) effective scheduling decisions and (2) efficient computation of scheduling decisions. Thus, with time consuming algorithms, the framework's efficiency declines.

### 3.6 Summary

The following is a list of important attributes of the scheduling framework described in this chapter:

- It is written specifically for dynamic applications.
- It is distributed to take advantage of parallelism available in multicomputers.

- The mappings of scheduler to multicomputer may be changed easily, thereby facilitating the scheduling system's adaptation to different architectures.

- It promotes building schedulers for several application classes.

- It permits use of various scheduling constraints and objectives.

- It supports scheduling with different operating system primitives.

The vast majority of previously written schedules are static and centralized. Centralized schedulers will eventually create a bottleneck as the number of processors and scheduling requests increase. Thus, the scheduling system must be distributed for efficiency in large multicomputers.

Like our framework, the distributed schedulers in [15, 37, 28, 41, 42] efficiently utilize a multicomputer system. Our scheduling framework is superior to these scheduling systems in the area of flexibility. Each of the above cited distributed scheduling systems is designed to schedule a single application class (single set of objectives and constraints) on a specific system architecture. In [42], provisions for dynamic scheduler substitutions (bidding or focused addressing) are described, but only one class of applications – deadline – is scheduled by the system. Our scheduling framework represents the first scheduling system that supports construction of effective and efficient schedulers for different system architectures, operating system constructs and classes of applications.
Sample implementations using our framework for varying scheduling paradigms are constructed in the following chapters. The implementations verify the framework's flexibility and reveal that it facilitates the development of efficient schedulers resulting in new scheduling algorithms.
CHAPTER IV

DYNAMIC SCHEDULING ON MULTIPROCESSORS

In this chapter, the framework is shown sufficiently general to use with different scheduling constraints and objectives on a multiprocessor. To display the flexibility of the framework, multiprocessor schedulers for the two different classes of applications - deadline and completion time - are presented. Using the framework to implement these schedulers verifies that the framework captures the essential features necessary for constructing effective and efficient on-line schedulers for different classes of applications on a multiprocessor. Efficient mapping of the 'link' portion of the framework demonstrates that we can take advantage of the special features offered by a multiprocessor (e.g., shared memory).

The implemented schedulers are timed to display their inherent efficiency. Analyses and measurements of the schedulers indicate that the use of process migration in conjunction with the framework is a viable alternative to static scheduling. Each scheduler is also measured for limits depicting when its performance ceases to benefit the application. The measurements verify that the implemented, non-
computationally complex schedulers compute effective schedules efficiently. In contrast to previous research, this demonstrates that efficient and effective schedulers do exist for a wide range of the real-time and non-real-time applications.

The description of the schedulers in this chapter begins with a presentation of parameters common to both the deadline and completion time schedulers. Next, the remaining parameters for the implemented deadline scheduler are described and timed. This is followed by a discussion of the deadline scheduling system's efficiency and effectiveness. Finally, completion schedulers are described and discussed in a similar manner.

4.1 The multiprocessor scheduling system

Numlocal and numglobal. Both the completion time and deadline scheduling systems presented below use one global scheduler for the entire application and one local scheduler per processor. The hardware used in the experimentation has too few processors (seven) to warrant the use of more than one global scheduler. One local scheduler and one multiplexor is assigned to each processor. This assignment enhances the efficiency of interactions between the local schedulers and the process multiplexors and results in a complete distribution of the scheduling system.

CONCURRENT MODULE scheduler(...
    num_local = 7;
    ...
    num_global = 1;
    ...
);

Queues. The distributed types in our scheduling framework requiring communication mechanisms are the queues and the 'link' distributed type. Communi-
cation is implemented via shared memory in the multiprocessor system. Thus, the queues are implemented as arrays located in global memory. The empty, insert, remove, and top operations are implemented in a straightforward manner.

**Links.** To efficiently utilize the shared memory available through the underlying hardware, the 'link structure' is implemented by a shared data structure termed the *offer data structure*. Thus, the offer data structure is the 'link structure' parameter to the scheduling framework. One such data structure is maintained for each global scheduler (see figure 3). Since the communication structure contains no any intermediate nodes, the 'combine' function serves no useful purpose. The 'combine' parameter supplied to the multiprocessor scheduling systems is consequently null.

The offer data structure contains both the scheduling request and information about the request returned by the local schedulers. The scheduling request information is implemented as an array with the unique identifier(s) of the process(es) being offered with each single 'request_info'. The task descriptor of each process
is contained in the PCB (process control block) of the process and is derived from the unique process identifier (see ‘task.descriptor’ in figure 3). The information returned by the local schedulers is represented by an array of records, termed return in the figure. There is one entry in this array for each local scheduler. The offer data structure also contains a bit, termed ‘scheduled’, signifying whether the offer is still available for scheduling.

Each offer data structure contains only one scheduling request at a time. This keeps the global memory required by the scheduling mechanism low. In addition to the request the offer data structure also contains a lock. Offers are only locked during offer entry by the global scheduler and scheduling the request by a local scheduler. This minimizes the time an offer is unavailable for local scheduler inspection.

Using the offer data structure, the functions ‘send_down_link’, ‘send_up_link’, ‘receive_down_link’, and ‘receive_up_link’ become simple global memory read and write operations. ‘Send_down_link’, called by the global schedulers, locks the global scheduler’s offer data structure, places the scheduling request into the data structure, and finally unlocks the offer. Since shared shared memory is available, no message sends are required. ‘Receive_down_link’ used by the local schedulers becomes an inspection of the offer data structure. The local scheduler writes scheduling information into the offer data structure using the ‘send_up_link’ operation. Finally, a ‘receive_up_link’ by a global scheduler consists of reading the scheduling information written by the local schedulers.
Thus, the scheduling framework is sufficiently general to allow efficient imple-
mentations of a scheduler in shared memory. The link data structure and queues
portions of the framework effectively operate within the shared memory of the
machine. One local scheduler is used for each processor to efficiently use the pro-
cessing power of the computer. This local scheduler placement distributes the
scheduling system's overhead across the multiprocessor.

4.2 Deadline scheduling on the multiprocessor

Measurements for the following items are made on the multiprocessor system:

**Basic costs** – the execution times of the global and local scheduler’s codes and
the cost of task migration (e.g., stack, code, data and state movement).

**Effects of load on local schedulers** - the effects of imposing certain loads on
local schedulers are analyzed by measurement of local schedulers’ execution
times as dependent on the number of deadline bins maintained.

**Group scheduling** – the cost of scheduling tasks consisting of groups of processes
is determined.

**Scheduling with constraints** – the execution times of local schedulers in the
presence of scheduling constraints (e.g., memory constraints) are determined.

All measurements are obtained using a generator of scheduling requests. The
request generator produces as output a sequence of requests with deadlines, release
times, and execution times. This sequence is forwarded to the global scheduler.
The generator's inputs are the number of tasks for which deadlines are to be generated, the latest deadline in the resulting list of deadlines, and the total desired load on the multiprocessor (1 - 100%). Task deadlines, release times, and execution times are drawn from pseudo-random distributions as follows. For each task, the generator first produces a random release time between the current time and the latest deadline. Then a random deadline is generated between the release time and latest deadline. From this it derives the task's maximum execution time as 'deadline - release time'. It then generates the task's actual execution time by multiplying 'deadline - release time' by a random number. If adding this task to its output sequence exceeds the total desired load specified by the user, then a new execution time is generated, and so on, until a request with an execution time leading to an acceptable total load is generated.

The timings listed below are the execution times of the scheduler's main components measured on a 6-processor configuration of the multiprocessor. Requests are generated by the workload generator that executes jointly with the global scheduler on one multiprocessor node, while local schedulers execute on the other five processors. Each processor is an Intel 8086 processor with an 8 MHz clock speed and a 750 ns basic instruction cycle. Its 8087 co-processor has the same cycle time. As with the other timings appearing in this thesis, the granularity of the timings is 6.5 microseconds. Timings are performed with the processors' internal clocks while no extraneous processing or I/O are being done. For comparison, execution times of typical 8086 instructions are 2 microseconds; a procedure call
without parameters requires approximately 15 microseconds. Regarding the operating system, approximately 1 millisecond is required for sending or receiving a 200 byte message from one processor to another, and a similar latency is incurred for a process switch on a processor.

The total cost of scheduling a single task arriving at a global scheduler and assigned to a processor consists of:

**Task migration latency** – which consists of the time required to move the task's state (execution stack, code and data segments), whenever the assigned processor is different from the processor on which the task is currently located.

**Scheduling latency** – the total time spent in the scheduler's code consists of (1) the time required by a global scheduler for making the initial offer, (2) the delay incurred by waiting for a response from some local scheduler, (3) the time used while choosing the local scheduler, and (4) the scheduling of the task by the local scheduler. Best case scheduling latency incurs no delay between the time an offer is made and the time it is accepted or rejected. Worst case scheduling latency incurs maximal delay between the first offering of the request and its acceptance or rejection by local schedulers followed by choosing the last local scheduler examined. In the system presented, worst case latency has an upper bound (timeout value) determined by the application programmer (set at 10 milliseconds in the tests).

Task migration cost is measured as \(0.049 + (0.00329 \cdot B)\) milliseconds, where \(B\) is the size of the task in bytes. This results in a latency of 13.5 milliseconds for
performing the migration of a 4K byte task (which is not atypical in the real-time applications explored).

4.2.1 Communication data structures

To simplify the explanation of the deadline scheduling system, we will describe an instance in which ‘task.descriptors’ contain only single process tasks scheduled for the single CPU resource. Each task has either a hard or a soft deadline and once a task has commenced execution, it may not be migrated to another processor.

The ‘task.descriptor’ shared by the local and global schedulers consists of the following information – deadline, ‘execution.time’, ‘release.time’ and type (hard or soft deadline), with an optional criticality field used when queuing the task. The ‘link.structure’ contains a ‘task.descriptor’ along with a bit signifying whether the task has been scheduled and an array of earliest possible completion times for the task (one entry for each local scheduler). The deadline, execution time, release time and task type are supplied by the global scheduler during the ‘send_down_link’ operation. In addition to supplying this information, the ‘send_down_link’ clears the array of earliest possible completion times and marks the task as not scheduled.

The implemented global scheduler requires 1.71 milliseconds for making an offer consisting of a single process task (a ‘send_down_link’ call). The array of possible completion times is updated by each local scheduler using the ‘send_up_link’ operation after inspecting the offer. A ‘send_up_link’ operation requires .13 milliseconds.

RECORD task_descriptor IS
    deadline : time_type;
    release_time : time_type;
    execution_time : time_type;

type : integer; /* hard/soft deadline */
criticality : integer; /* optional */

ENDRECORD;

RECORD link_structure IS
  task : task_descriptor;
  lock : integer;
  scheduled : integer;
  return_info : array [num_local] integer;
ENDRECORD;

4.2.2 Queuing

Tasks awaiting inspection by a local or a global scheduler are ordered by criticality. This ensures that the scheduling system attempts to schedule the most critical task first. In the implemented scheduling system, criticality is defined as the earliest deadline. Thus, the scheduling requests enqueued at the schedulers are ordered earliest deadline first and the criticality field shown in the program example is not used. Requests with identical deadlines are subordered in a FIFO manner. Both orderings are specified through the 'queue_ordering' parameter.

Whenever a process is ready to commence execution (when its release time has been reached), it is scheduled by the process multiplexor resident on each node. The 'multiplexor_enqueue' function orders the multiplexor's ready queue by earliest deadline first and processes with identical deadlines are sub-ordered by process criticality (when supplied). Earliest deadline first is used because it is an optimal scheduling method[11]. The multiplexor continually executes the uncompleted process with the earliest deadline and feasible release time. This scheduling discipline, coupled with the 'scheduling_measures' function, guarantees every scheduled process completes execution prior to its deadline.
4.2.3 The arbitrate function

Each local scheduler supplies the global scheduler with the earliest possible completion time for a soft deadline task that cannot be entirely scheduled by the local scheduler between the task's release time and deadline. This value is reported in the 'return_info' section of the offer data structure and used by the global scheduler when no local scheduler can completely execute a task between the task's deadline and release time. The implemented 'arbitrate' function simply compares the 'return_info' values and chooses the local scheduler that can complete the scheduling request at the earliest possible moment. The amount of processing time consumed by a global scheduler during the implemented 'arbitrate' function is approximately \(0.26 + (0.05 \cdot L)\) milliseconds, where \(L\) is the number of local schedulers in the system. The request's type is set to 'relaxable' and it is then placed in the chosen local scheduler's queue requiring 1.05 milliseconds.

4.2.4 The scheduling_measures, relax_constraints, and schedule functions

The scheduling measure supplied to the local scheduler by a deadline application is to guarantee the entire execution of each scheduled task between the task's release time and deadline. The 'scheduling_measure' and 'schedule' functions are implemented by using scheduling calls contained in a scheduling library. In fact, calls for all scheduling paradigms addressed in the text are included in this library. The operation of the 'scheduling_measures', 'schedule', and 'relax_constraints' functions are explained in the following text.
The implemented local scheduler first attempts to schedule tasks in its local request queue (see local scheduler description in section 3.1). When no local requests are available, the local scheduler examines the offer data structure. If the offered task has already been scheduled, it ignores the offer, does not compute the 'schedule_measure' function and proceeds to the next scheduling request. When the offered task is unscheduled and the local scheduler can complete the task between the task's release time and deadline, the local scheduler locks the offer data structure, marks the task as taken and schedules the task. This prevents other local schedulers from inspecting the scheduled offer. Note that the local scheduler performs an 'arbitrate' (to itself) and a 'send_down_link' on behalf of the global scheduler by scheduling the task and setting the scheduled bit of the offer data structure respectively. The measured processing time required by the local scheduler for both the 'arbitrate' and 'send_down_link' is 0.15 milliseconds. This is an extensive savings over the global scheduler's 'arbitrate' and 'send_down_link' operations measured in section 4.2. The difference in processing time lies in the fact that the global scheduler scans a list searching for a minimum during an 'arbitrate' and inserts the task in a local scheduler's queue while the local scheduler simply locks the offer data structure, schedules the task, updates the scheduled bit in the offer data structure, and finally unlocks the task.

The 'schedule' function maintains a list of unused processing times on each of the processors it controls. Each element of the list is considered a bin containing (1) its ending time and (2) its packing element. The packing element of a bin
is the remaining amount of available execution time between the bin’s end time and the previous bin’s end time. An explicit bin packing algorithm is used by the ‘schedule’ function, where a separate bin is maintained for each unique deadline and release time of the locally scheduled requests. Each local scheduler initially contains one bin with an end time of $\infty$ and available processing time, packing element, of $\infty$. This list of bins constitutes the shared data structure used by the ‘schedule’, ‘scheduling_measures’, and ‘relax_constraints’ functions.

RECORD bin IS
  end_time : time_type;
  available_time : time_type;
  next : integer;
  previous : integer;
ENDRECORD;

SHARED DATA /* bins could be implemented as a linked list */
  bins : array [num_bins] of bin;
  first_bin : integer;

The list of bins is used on three separate occasions. First, the ‘scheduling_measures’ function uses it when inspecting a local request or a global offer, the function scans through its bins with end times greater than the request’s release time and less than the request’s deadline. During this scan, it computes a sum expressing the total available execution in those bins between the task’s release time and deadline. A timing of the ‘scheduling_measures’ function indicates it requires $1.52 + (0.12 \cdot X)$ milliseconds where $X$ is the number of bins scanned to calculate the sum.

FUNCTION schedule_measures(request_info : task_descriptor);
...
  bin = bins[first_bin];
  while (bin <> NIL) and (bin.end_time < request_info.start_time) do
bin = bins[bin.next];
endwhile;
if (bin <> NIL) and (bin.end_time > request_info.start_time) then
    free_time = MIN(bin.end_time - request_info.start_time,
                   bin.available_time);
else
    free_time = 0;
endif;
while (bin <> NIL) and (free_time < request_info.execution_time)
    and (bin.end_time < request_info.deadline) do
    free_time = free_time + bin.available_time;
    bin = bins[bin.next];
endwhile;
if (bin = NIL) then
    free_time = free_time + (request_info.deadline
                            - last_bin.end_time);
else
    free_time = free_time + MIN(request_info.deadline
                                 - bin.previous.end_time,
                                 request_info.execution_time,
                                 bin.available_time);
endif;
if free_time > request_info.execution_time then
    return(TRUE);
else
    return(FALSE);
endif;
endschedule_measures;

The second time the scheduler uses its list of deadline bins is when scheduling a task. If the computed sum of available execution times exceeds the execution time required by the current request before its deadline, the 'schedule' function either adds the request to a bin that has an end time equal to the request's deadline or creates a new bin for this deadline. It also creates a second bin if the request's release time does not equal the end time of a previously formed bin. Finally, it adjusts the amounts of available execution time in the scanned bins, and if necessary, fills the new bins with the proper amount of available available time. The 'schedule' function requires $1.11 + (0.13 \cdot X)$ milliseconds when no new bins
are created, $1.16 + (0.13 \cdot X)$ milliseconds when one new bin is created, and $1.21 + (0.13 \cdot X)$ milliseconds when two new bins are created.

```c
FUNCTION schedule(request_info : task_descriptor);
    ...
    bin = bins[first_bin];
    while (bin.end.time < request_info.deadline) do
        bin = bins[bin.next];
    endwhile;
    if (bin = NIL) or (bin.end.time<> request_info.deadline) then
        bin = makenewbin(request_info.deadline);
        while (request.info.execution.time > 0) do
            if (bin.available_time <= request.info.execution_time) then
                request.info.execution_time = request.info.execution_time - bin.available_time;
                bin.available_time = 0;
                bin = bins[bin.previous];
            else
                bin.available_time = bin.available_time - request.info.execution_time;
                request.info.execution_time = 0;
            endif;
        endwhile;
    endif;
    bin = first_bin;
    while (bin <> NIL) and (bin.end.time < request_info.start.time) do
        bin = bins[bin.next];
    endwhile;
    if bin.end.time <> request_info.start.time then
        makenewbin(request.info.start.time);
    endif;
    endschedule;
```

The third use of deadline bins is by the 'relax_contraints' function concerning the adjustment of soft deadlines. Namely, when the computed sum of available execution times is less than the execution time required by the request and a request has a soft deadline, the request's deadline is adjusted. An algorithm similar to the one employed for the initial evaluation of the request is used to reset the request's deadline. Bins are scanned until enough execution time for the request is found.
Based on that scan, the deadline of the request is then altered to the earliest possible time at which the task can complete execution without interfering with previously scheduled tasks. The 'relax.constraints' function's computation time also depends on the number of bins scanned and is measured as $1.31 + (0.12 \times X)$ milliseconds. The request with its updated deadline is scheduled by adjustment of the available times in the appropriate bins in the same manner as other requests.

FUNCTION relax.constraints(request_info : task_descriptor)

```
free_time = 0;
bin = first_bin;
while (bin <> NIL) and (bin.end.time < request_info.start.time) do
    bin = bins[bin.next];
endwhile;
while (bin <> NIL) and (free.time < request_info.execution.time) do
    free_time = free_time + bin.available.time;
    bin = bins[bin.next];
endwhile;
if (bin = NIL) then
    request_info.deadline = last_bin + (request_info.execution.time
    - free.time);
else
    request_info.deadline = bin.deadline;
endif;
endrelax.constraint;
```

The following example of a schedule constructed by the implemented deadline 'scheduling_measures' function further illustrates the bin packing algorithm explained above. Specifically, consider a processor without processes at time $Y$. This implies the existence of a single bin (called 'bin a') with an infinite amount of execution time (in the figures below, free time in bins is unshaded, whereas used time is shown with various shadings):

<table>
<thead>
<tr>
<th>Available processing time</th>
<th>Scheduled processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If the processor's 'scheduling_measures' function accepts a process 'P1' with deadline $Y + 20$, release time $Y - 10$, and execution time of 5 seconds, three bins result, where 'bin b' is only partially filled:

```
X-----------------------bin c      bin b — -------------------1 —  bin a
X+10                     X+20
```

'Bin b' is created to record the deadline of the process, whereas 'bin c' is created to keep track of the process's release time. 'Bin b' is partially filled, whereas 'bin c' is empty.

A second process 'P2' with different deadline $Y + 12$ and release time $Y$, and an execution time of 5 seconds requires the creation of two new bins, one each for the second process' deadline and release time, which both differ from those of the first process:

```
X-----------------------bin d      bin c      bin e — bin b — -------------------1 —  bin a
X+10                     X+20
```

A third process 'P3' arriving with a deadline after that of 'P1' and 'P2' at say $Y + 16$ and a release time prior to 'P2' will result in its scheduling 'around' 'P2', and it leads to the creation of two additional bins, since bins with either its release time or its deadline do not yet exist:

```
X-----------------------bin d      bin g — bin c      bin e — bin f — bin b — bin a
X+10                     X+20
```
Note that the constructed schedule displays 'P3' as running twice in order to attain its desired total execution time. Such preemptive action actually will not occur in this case because of the manner in which the node multiplexor schedules the processes (see section 4.1.3 for a description of the 'multiplexor_enqueue' function).

A global scheduler does not itself inspect any of the deadlines, release times, or execution times of a request's processes. Instead, an offered request is scheduled by the first local scheduler able to determine that its available execution time after the task's release times and prior to the task's deadlines exceeds the request's required execution time. When all the local schedulers have responded unfavorably, the global scheduler takes one of two actions.

In the case of a hard deadline the global scheduler simply notifies the application program of the non-scheduled task and attempts to schedule the next task. For an infeasible offer with a soft deadline, the global scheduler forces the offer's acceptance by a local scheduler. Such an assignment is performed by setting the request's type to 'relaxable' and inserting the request into the chosen scheduler's local request queue and is specified by the 'arbitrate' function.

4.2.5 Measured scheduling efficiency

Below, scheduling efficiency is shown sufficient for medium-rate tasks in the ASV application (tasks with average execution times equal to or higher than 100 milliseconds). Scheduling efficiency is shown insufficient for high-rate tasks with execution times of 10 milliseconds. In fact, such tasks should be scheduled statically, which
is evident by the fact that a simple migration of an average size task consumes 13.5 milliseconds (see page 56). The processing time consumed by migrating such a task is larger than the worst case amount of processing time required by the task. This clearly demonstrates that high-rate tasks should be scheduled statically.

To display the implemented scheduling system is sufficient for scheduling medium rate tasks, the following measurements are taken with the request generator producing a fixed sequence of requests, each of which is offered by the global scheduler, then inspected, accepted, scheduled, and discarded by local schedulers. The values reported here are the averages of the actual execution times measured for a request sequence of length 1000. This also holds for the other measurements reported in this section.

The best case scheduling latency for a request entering a global scheduler is approximately 3.6 milliseconds – 1.71 milliseconds for a ‘send_down_link’ by the global scheduler and 1.89 milliseconds for a ‘scheduling_measures’ (with one bin), ‘schedule’, ‘arbitrate’, and ‘send_down_link’ by a single local scheduler. This compares very favorably with the other basic parameters of the operating system cited previously (e.g., 1 millisecond for sending a message). The critical point where the scheduler seems to first benefit the application in the current system is when all tasks of the application have an execution time of at least 100 milliseconds.

In the general case, the time taken by each local scheduler to inspect, accept, and schedule a request depends of the number of bins examined, which in turn depends on the number of tasks already scheduled on the processor. Specifically,
if the deadline and release time of the request being inspected by a specific local scheduler are the same as those of already existing bins, then the combined request inspection and schedule construction time is measured as $2.63 + (0.25 \cdot X)$ milliseconds (a 'schedule_measures' and a 'schedule' combined), where $X$ is total the number of bins examined for the request. This time does not increase much for the case in which the local scheduler must also create new bins for the request, which is $2.74 + (0.25 \cdot X)$ milliseconds. These measurements are obtained using a fixed request sequence produced by the request generator. The sequence used to obtain the first measurement consists of three requests with different deadlines offered to a single local scheduler, followed by 1000 requests with equal but later deadlines. This ensures that a fixed number of bins must be scanned for each request being scheduled. Similarly, for the second measurements, 1000 requests with different deadlines and release times follow the first three requests.

The decision-making latency for scheduling a task consisting of a group of processes (or a cyclic task) increases linearly with the number of group members (or iterations). Namely, it is the sum of the members' individual scheduling times. Thus, a group of 6 processes in an application may be scheduled in 21.6 milliseconds in the best case.

The consideration of resource constraints regarding fixed resources (e.g., memory) incurs a negligible amount of additional scheduling latency. However, decision-making latency increases linearly in the number of schedulable resources required by a task. This is due to the fact the current time available on each such resource
must be represented by a set of bins – recall that the CPU time available on each processor prior to a specific deadline is represented in the same fashion. Since bin inspection and update are the most time consuming activities in a local scheduler, each additional schedulable resource considered by a local scheduler increases scheduling cost by a factor proportional to the cost of checking the feasibility of the task’s deadline. Thus, the scheduling latency for a task requiring three of the processor’s schedulable resources is approximately 3 times larger than that of a task that does not require any schedulable resources.

More meaningful than the basic costs of scheduling presented above are analyses of scheduling overhead, which is defined as the percentage of time spent in the scheduler vs. executing application code. Below, overhead is computed using the measured latencies of request scheduling described in the previous section. First, worst case overheads are developed by assuming worst case scheduling latency. Worst case overhead is computed as a sum of the basic scheduling costs computed above and the maximum possible delay between the presentation of an offer by the global scheduler and its acceptance or rejection (set to 10 milliseconds). This sum is divided by a sum consisting of the average task execution time, the basic scheduling cost, and the maximum delay. Best case overhead is computed in the same fashion, where the term expressing maximum delay is assumed to be zero.

The results demonstrate that worst case overheads are dependent on load (the number of bins on each processor), the expected value of the execution times of the tasks being scheduled, and the number of schedulable, additional resources...
Figure 4: Worst case scheduling overhead in a 10 processor average 10 bin each system.

required by each request. These results show that measured overhead for a request sequence with execution times drawn from a uniform distribution approximates the best case rather than worst case overheads. Last, the efficiency of the scheduler and of its 'offer' mechanism are compared to that of scheduling based on bidding.

Figure 4 depicts worst case scheduling overhead in a 10 processor system, where 10 bins are assumed to exist on each processor. This corresponds to the situation likely to exist in the actual configuration of the ASV's control software, where a substantial number of processes execute on each processor. As in most of the figures below, the figure's x-axis lists the number of schedulable resources required by each request, so that the overhead’s dependence on this number can be shown.

This figure demonstrates that overhead varies with the average execution times of tasks, where average execution times of 100 milliseconds, 1 second, and larger,
experience acceptable scheduling overheads when a single schedulable resource is required by each request (i.e., the CPU). Again, tasks with average execution times of under 10 milliseconds should not employ this scheduler. For the dynamic real-time applications, these results imply that its higher-rate periodic tasks (exceeding 20 Hz) should not be scheduled dynamically with the deadline scheduler described here. For such tasks, dynamic scheduling other than local scheduling by use of a simple earliest deadline first algorithm is not advisable. Thus, their dynamic reconfiguration will be difficult.

While figure 4 depicts worst case scheduling overheads for scheduling a single resource (the CPU), figures 5 and 6 show that overhead decreases as each request requires greater processing time and overhead increases with each additional schedulable resource. Each of these figures depicts multiple curves, where
Figure 6: Worst case scheduling overhead in various systems with tasks average execution time of 1 second.

each curve plots the scheduling overhead for a number of schedulable resources given (1) a specific number of processors, and (2) an average number of bins per processor.

In figure 5, an average task execution time of 100 milliseconds is used. As can be seen, with two schedulable resources, worst case overhead varies from approximately 18% to 40% depending on the number of bins and processors involved. These overheads may be compared with overheads of 12% to 27% for a single schedulable resource shown in the same figure. Thus, a larger number of schedulable resources (more than 2) per request results in unacceptable overhead for medium-rate tasks (100 millisecond execution times). However, in figure 5, overhead is significantly less and quite acceptable for lower rate tasks (1 second execution times) and two resources (from 2% to 6%). Note that these figures also
Figure 7: Worst case, best case, and measured overheads: 5 processors, 10 bins, 100 millisecond average execution times.

demonstrate that tasks with execution times significantly larger than 100 milliseconds can be scheduled with acceptable overhead in a fairly loaded system.

Figure 7 contrasts the worst case, best case, and measured scheduling overheads for an average execution time of 100 milliseconds assuming 5 processors and 10 bins per processor. The curve in this figure displays measured overhead consisting of values obtained in a system with 6 processors, one of which runs the request generator. In these measurements, each request generated by the workload generator consists of a single process task, and the 100 millisecond average execution time of these tasks is the mean of a normal distribution of execution times used by the request generator. The measured values are averages of 50 runs of the workload generator, where desired loads vary from 60% to 100%.

As evident from the figure, the measured overhead lies between the worst and
best case results and is quite acceptable even when each task requires 3 schedulable resources. Additional measurements not shown in the figure attain similar values when an average number of 20 bins per processor is used. In that case, measured overhead for 3 schedulable resources is approximately 20%.

Scheduling latency is highest for a task with a soft deadline that cannot be met by any local scheduler. In this case, a global scheduler must wait for all local schedulers to respond to the offer, or it must wait for its maximum delay to be exceeded. After this wait, it must compare the available responses from local schedulers (i.e., the load information) and then force scheduling of the task. Thus, the exceptional case in the presented scheduling mechanism degrades to that of bidding mechanisms which can be used to create first, best, and worst fit bin packing. This type of algorithm is first developed for workload balancing in computer networks by Stankovic[41, 58, 59, 65, 37]. In this type of algorithm, either an underloaded processor seeks additional load by placing a bid for acquisition of additional tasks, or an overloaded processor emits a bid describing its surplus tasks for acquisition by other processors. All bid responses are compared in order to choose a processor to which a task should be migrated.

When the global scheduler always performs bidding, its scheduling latency increases significantly for two reasons. First, it must always wait for responses from all local schedulers eligible for an offer. Second, it must always perform the explicit comparison of local schedulers’ responses. Figure 8 shows scheduling overhead measured for a sequence of single process requests that do not require additional
schedulable resources. The measured values in the figure labeled 'bidding using shared memory' are attained by forcing all local schedulers to initially reject the request offered and then having the global scheduler assign this request to the best local scheduler. Since such measurements use the offer mechanism for the implementation of bidding, they cleanly separate overheads due to the basic notion of bidding from overheads that may arise due to the use of messages for the implementation of bidding in computer networks. Bar graphs depicting the measured overheads of bidding vs. offers for 5 and 10 processors are shown. Since the experimental multiprocessor only has 6 processors, the measurements for 10 processors are attained by having two local schedulers share a processor. Thus, the measured values for bidding and offers using shared memory are somewhat pessimistic compared to the actual performance of bidding in a 10 processor system.

Last, it can be concluded that the efficiency of offers will exceed that of bidding even when offers and bids cannot make direct use of shared memory, as can be seen by comparison of the entries labeled 'bidding with messages' and 'offers with messages' in the figure. The numeric values depicted in those entries are obtained by adding the latencies of message sending and receiving in the embedded multiprocessor to the latencies of offers and bidding. Specifically, the global scheduler issues bids or offers via messages broadcast to local schedulers, and then waits for responses to those offers or bids. The implementations of offers and bidding require the same numbers of messages in the worst case in which offers degrade to bidding. Namely, assuming a broadcast message passing mechanism in the un-
Figure 8: Comparison of offers to bidding using messages and shared memory.
derlying GEM operating system, bidding would require at least \( 3P - 2 \) message operations performed by processors: one broadcast message from the global scheduler to the local schedulers followed by \( P - 1 \) receive operations for this message, followed by \( P - 1 \) return messages from local schedulers, followed by \( P - 1 \) receive operations by the global scheduler (\( P \) = number of processors), concluded with one broadcast message announcing the processor awarded the bid. This results in the total scheduling overhead of:

\[
1.71 + (5.16 + (0.25 \cdot X)) \cdot P
\]

milliseconds.

The formula above describes scheduling overhead for bidding and also worst case scheduling overhead for offers. However, in the best case for offers, the first response from a local scheduler is taken, and additional messages from other local schedulers received later can be ignored, thereby reducing the number of messages to be received by the global scheduler compared to bidding. This results in a total of 4 message operations when the first local scheduler's response is positive. The resulting range of performance of offers using messages is indicated in the figure. Thus, the use of offers not only reduces scheduling latency due to reductions in the costs of offer collection and comparison, but also reduces the number of messages required for scheduling.

The comparison of offers and bids using messages clearly displays that offers (any fit scheduling) are preferable to bidding (best, worst, and first fit) in non-shared memory systems (e.g., computer networks), as well. Specifically, since
scheduling costs in local area networks are dominated by the costs of message transmissions required for scheduling, algorithms that reduce that cost (e.g., the any fit algorithm), should outperform other algorithms (e.g., bidding) regarding its efficiency. Furthermore, offers should also outperform bidding due to a reduction in interference with 'busy' local schedulers, since local schedulers may ignore an offer, but are usually assumed to respond to a bid.

4.2.6 Scheduler effectiveness

While the previous section demonstrates that the scheduling algorithm is efficient, its effectiveness (i.e., an evaluation of the quality of the decisions made by the algorithm) has to be determined, as well. This requires assumptions regarding typical request arrivals or detailed experimental measurements regarding the dynamic behavior of a large, multi-computer, real-time system. Since such experimental data is not available, worst case results are compared below to experimental results attained by making commonly used assumptions regarding expected request arrivals, execution times, and deadlines. The measure of effectiveness used is geared to the real-time domain: it is the average timeliness, namely the percentage of scheduled requests which finish execution before their deadline.

The worst case behavior of the scheduling algorithm can be formulated analytically if it is assumed that a fixed upper bound (called the 'termination time' of the test interval) exists for arbitrary request deadlines and the scheduling overhead is included in the execution time of each task, where:

- \( n \) = no. of requests to execute before the termination time.
Figure 9: Worst case percentages of scheduling requests meeting deadlines.

- \( N \) = no. of processors in the system.

- \( u = \text{sys. util.} = \frac{\sum \text{execution times of requests}}{N \cdot \text{interval length}} \)

For this case, simple analysis shows that the following formula holds for the worst case percentage of request deadlines met in relation to the number of processors and total system utilization:

\[
\left( \frac{\left( n - \left\lfloor (u \cdot N) - 1 \right\rfloor \right)}{n} \right) \cdot 100\%
\]

Figure 9 displays the resulting analytical relationships between the percentage of system utilization, the number of processors, and the worst case percentage of deadlines met.

A worst case occurs when several smaller requests are distributed across all processors during an interval when a request for over 50% of the processing time
for the interval arrives. For instance, consider the case of a 3 processor system in which 3 requests with common deadlines have been scheduled one on each processor. Also assume each request requires 8% of a processor's total available time before its deadline (thereby utilizing a total of 8% of each processor and of the system). At a system utilization level of 40% (see the column labeled 40% in Figure 9), a maximum of 120% (40% \cdot 3) of the execution time of a single processor is available in the system. At this point, the worst case occurs when a large request arrives with the same deadline and requires over 92% of a single processor's execution time during the interval. Since no processor can schedule the request to complete execution before its deadline, the request will not be scheduled to meet its deadline and only 75% (3 out of 4) of the requests have been scheduled.

Further inspection of this single case shows that the attainment of such worst case results depends on the sequencing and timing of scheduling actions. For instance, even with the poor execution distribution from the above example, if the large, 96% execution time task is scheduled before any of the smaller tasks then all 4 of the tasks will be scheduled. Assuming equal probability of the tasks being scheduled in any order, 94.4% of the time all tasks will be scheduled. 75% of the time the large task will be scheduled before one of the three other tasks creating a schedule of all tasks completing before their deadline. Also, even when the three smaller tasks are scheduled first, there is a probability that they will not all be scheduled on different processors and all tasks could then be scheduled. Since the algorithm is an any fit algorithm, the probability that all three small
tasks will be scheduled on different processors is 22.2%. Even when the worst case scheduling sequence occurs, a majority of the time all tasks will be scheduled to execute completing before their deadlines. Combining this with the assumption that all different sequences of requests occur with equal probability, all requests will be scheduled 94.44% of the time given this poor distribution.

To further display that worst case scheduling occurs infrequently is displayed by analyzing a 3 processor system with a 95% load. A worst case distribution for this would be 5 tasks with identical release time and deadlines, three of the tasks require 29% of a single processor between the common release time and deadline while the other 2 require 99% of a single processor between the release time and deadline. The worst case occurs when all three 29% tasks are scheduled on a single before a larger task is scheduled. Assuming each arrival sequence has equal probability, every task is scheduled through the global scheduler, and each local scheduler has an equal probability of scheduling each task the following figures are obtained for this worst case distribution: 2 tasks unscheduled - 2.22% of the time, 1 task unscheduled - 45% of the time, and 0 task unscheduled 52.78% of the time.

To compare the worst case data presented above with more realistic data, request arrivals using the request generator are used below. While both Poisson and fixed arrival rates are used, tests with the two types of request rates resulted in an insignificant difference in the percentages of deadlines met. The measured results attained for request sequences of with execution times drawn from a uniform distribution and desired loads of 50%, 70%, and 100% are shown in figure 10, in
which the average values attained for 50 runs are displayed. As can be seen, the any fit scheduling algorithm does operate at a near optimal level and its average case behavior is much better than the worst case. For example, in an experiment with 70% load on 5 processors, 99.9% of requests are scheduled, as compared to the scheduling of only 67% of the same requests in the worst case. The above experiment used the workload generator to formulate 1000 feasible schedules repeatedly. On the average, one out of each 1000 different loads is not successfully scheduled by the scheduler. These experimental results demonstrate that the scheduling algorithm should behave optimally (every task completes execution before its deadline) under low system loads and near optimally under higher loads.

The experimental results described in this section demonstrate that an efficient and effective, on-line deadline scheduler can be constructed using the scheduling
framework described in chapter 3. This verifies that the framework captures the essential features necessary for multiprocessor deadline schedulers. The multiprocessor implementation of the evaluated scheduler exhibits efficiency properties that permit the scheduler's use with medium-rate single process tasks (i.e., tasks with execution times $\geq 100$ milliseconds). Namely, the measured scheduling overhead will not exceed 15% for such tasks, even assuming a moderately heavy system load (10 different deadline bins per processor, which should correspond to at least 15-20 tasks per processor). The efficiency is due mainly to the prudent use of shared memory through the offer data structure which is shown to outperform bidding with no loss of scheduling effectiveness. Furthermore, the schedulers permit application programmers to construct complex, dynamically reconfigurable real-time systems. The next section applies the scheduling framework to completion schedulers for a multiprocessor system.
4.3 Completion time scheduling

Use of the framework for completion time schedulers verifies that the framework provides the essential features necessary for an effective and efficient scheduler on two different classes of applications. Measurements are taken to derive the implemented schedulers' efficiency and the developed algorithms are evaluated to display their effectiveness. The measurements show that efficient, effective, on-line schedulers exist for non-real-time applications.

Our schedulers differ from previously developed completion schedulers in two ways. First, most other completion schedulers operate statically and can not accommodate dynamic applications. Second, other schedulers are based on scheduling tasks given the tasks execution time. Our algorithms are general enough to schedule applications that provide application dependent estimations of task execution time or any other application dependent measure along with also scheduling applications that provide task execution times.

The parameters 'num_local', 'num_global' and the 'link' distributed type for the simulated completion time schedulers are identical to the respective parameters used for deadline schedulers. To simplify the explanation of the completion time scheduling system, we describe an instance of the scheduling framework that makes the following assumptions:

- All tasks are single process tasks.
- The only resource being scheduled is the CPU.
• Task migration requires one unit of processing time at the target node.

• The worst case execution time of each task is known (this value may be an estimate and not an actual amount).

• The worst case execution of each task is larger than one time unit.

Although the worst case execution time of each task is used in the following schedulers, they can easily be modified to allow average case measures or estimates of execution time instead of exact worst case measures. These assumptions also allow the developed scheduling system to be analyzed easily.

4.3.1 Scheduling without mid-execution migration

The simulated scheduling method without mid-execution migration schedules each newly arriving task on the processor with the smallest current load. When a scheduling request arrives at a global scheduler, the global scheduler chooses the local scheduler with the smallest 'load value'. The load value is defined as the sum of the remaining processing times of the previously scheduled tasks.

Task descriptor and link structure. The 'task_descriptor' shared by the local and global schedulers consists solely of the task's execution time. The 'link_structure' is again implemented as an offer data structure. It contains an array, termed 'load_info', with one entry for each local scheduler. Each entry in the array represents the latest possible time at which the corresponding local scheduler will complete all previously scheduled tasks. A 'receive_up_link' is composed of a global scheduler first locking the 'load_info' array and then reading information
from it. The ‘send_down_link’, called by a global scheduler, updates the ‘load_info’
value of the processor to which the task is assigned and unlocks the array. Finally,
a local scheduler uses the ‘send_up_link’ to lock, update, then unlock the ‘load_info’
array.

RECORD task_descriptor IS
  execution_time : time_type;
ENDRECORD;

RECORD link_structure IS
  load_info : array [num_local] integer;
ENDRECORD;

The arbitrate function. The global scheduler’s ‘arbitrate’ function is simply
a greedy strategy. It assigns each newly arriving task to the local scheduler with
the smallest load value contained in the ‘load_info’ array. Note that in this case,
the global scheduler does not need a ‘send_down_link’ call to receive scheduling
information from the local schedulers, it just performs a ‘receive_up_link’ because
its required scheduling information is continually available through the ‘load_info’
array.

The schedule, scheduling_measures, and relax_constraints functions.
The ‘schedule’ function inserts a task into ready queue of the processor that the lo­
cal scheduler encapsulates. When scheduling a task not from a global scheduler, the
local scheduler also must updates the load value of the ‘load_info’ array by adding
the processing time required by the task to the ‘load_info’ using a ‘send_up_link’.
The ‘send_up_link’ consists of first locking the ‘load_info’ array, then updating the
entry corresponding to the local scheduler making the ‘send_link_call’, followed by
unlocking the array. Since the ‘load_info’ array always contains the current values
used by the global scheduler, there is no need for a 'scheduling_measures' function for the local scheduler. In the implemented scheduler the 'schedule_measures' parameter is set to NULL. The 'relax_constraints' function is NULL because every task can always be scheduled by any local scheduler without changing any information in the 'task_descriptor' with completion applications. The simplicity of the 'scheduling_measures' and 'schedule' functions are depicted in the following code example:

```c
FUNCTION schedule(request_info : task_descriptor);
...
  processing_time = processing_time + request_info.execution_time;
  if not (assigned by a global scheduler)
    send_up_link(request_info.execution_time);
  endif;
endschedule;
```

Queuing. The multiplexor queues are ordered in a FIFO manner, because any execution order results in the same final completion time. When the criticality field of the task descriptor is used, the queue is arranged in a most critical task first ordering.

The global scheduler's wait queue is ordered by largest execution time task first. This enhances the scheduler's effectiveness by not allowing many small tasks to be distributed across all the processors while a large task is waiting in the queue. The enhanced effectiveness is depicted in figure 11. This figure shows schedules for a 2 processor system (labeled processor 1 and 2) with 3 tasks (labeled task A, B, and C) in the global waiting queue requiring 10, 10, and 20 units of processing time respectively. If tasks A and B are considered first (see left hand side of figure 11), they are scheduled to execute on processors 1 and 2. Task C would then be
scheduled on either processor 1 or 2 and the completion time would be 30 time units. If task C were scheduled first or second (see right hand side of figure 11, then tasks A and B would be scheduled on the same processor and the completion time would be 20 time units.

4.3.2 Measured efficiency and effectiveness

Efficiency. The scheduling overhead generated by the above algorithm is extremely small due to the simple scheduling functions applied. To assign a request arriving at a global scheduler, the global scheduler simply locks the ‘load_info’ array, scans the array finding the minimum, assigns the task to the corresponding local scheduler, and updates the array. The measured value for the entire action on the multiprocessor is $1.24 + (0.05 \cdot P)$ milliseconds.

Scheduling efficiency of the local schedulers is represented by a fixed cost. In
the worst case, scheduling a locally arriving offer consists of inserting the task into
the multiplexor queue, locking the 'load info' array, altering the corresponding load
value in the array, and unlocking the array. This action requires 0.56 milliseconds
on the experimental multiprocessor. When considering medium rate tasks with
execution times greater than 100 milliseconds, scheduling overhead will be under
7% even for 100 processor systems.

Effectiveness. With the above described scheduling system, the least efficient
schedules for various multiprocessor systems are shown in figure 12. The figure
assumes every task is migrated once before commencing execution and the task
execution time includes the time to migrate the task. A worst case occurs in a $P$
processor system when a task requesting to be executed for $P$ time units arrives
and many tasks have already been scheduled for the first $P - 1$ time units on
each processor. In this situation, the algorithm schedules the tasks to complete
execution in $2 \cdot P - 1$ time units. An optimal solution for this situation schedules
all of the tasks to complete execution in $P$ time units.

Theorem: $\frac{2 \cdot P - 1}{P}$ is the worst case scheduling performance of the above scheduling algorithm.

Proof: Suppose there exists a set of tasks (call them $T$) that can optimally be
scheduled on a $P$ processor system in $B$ time units. The total processing time of
all tasks $\leq B \cdot P$. An optimal schedule is $B$ time units, which implies that no single
task $T_i$ has an execution time of more than $B$ units. With these two facts in mind,
the following discussion proves the simulated scheduling algorithm schedules the
Figure 12: Worst case scheduling behavior for execution-time-only applications no

task migration.

Suppose a processor with \( \sum \) previously scheduled tasks \( \geq B \) is assigned a task
by the global scheduler. This implies all the processors have sums of tasks \( \geq B \), which in turn implies that the total processing time of the tasks \( > B \cdot P \),
contradicting the fact that the total sum of processing time \( \leq B \cdot P \). Therefore, the
largest sum of previously scheduled tasks on a processor able the receive another
task from the global scheduler equals \( B - 1 \) time units. Furthermore, for the worst
case to happen, every processor must have tasks scheduled for the first \( B - 1 \) time
units. Thus the largest task able to be scheduled on this processor is \( B \). This
implies the worst possible schedule by the algorithm is in \( B + (B - 1) \).

Suppose \( B < P \). The worst schedule is \( \leq \frac{B + (B - 1)}{B} \) which is \( \leq \frac{2P - 1}{P} \) for \( B < P \).

Suppose \( B \geq P \). The worst case schedule is \( \leq \frac{B + (B - 1)}{P(B - 1) + B} \) which through algebraic
manipulation is \( \leq \frac{2^P - 1}{P} \) for \( B \geq P \).

For the vast majority of task distributions the simulated scheduler performs much better than the derived worst case. The gap between worst and average case is described and discussed in the following section. To evaluate average case behavior, a workload generator creates tasks requiring various amounts of execution time, various total number of tasks, using various task arrival rates, and using various task arrival orderings. The different workloads are then scheduled. The experiments show no statistically significance between the different arrival rates and execution times. The results from the experiments verify that the average case scheduling effectiveness for different distributions fair much better than the worst case.

Because of the extreme efficiency of the implemented scheduler, the simulation assumes that scheduling overhead is included in task migration cost. Task migration is assumed to consume one time unit on the target processor (recall that task migration is measured at approximately 13 milliseconds on the experimental multiprocessor system while on the average, the scheduling system is measured at 1 millisecond to schedule the task).

Figure 13 displays the average case scheduling effectiveness by comparing the measured effectiveness to an optimal schedule. Each point plotted represents the average finish time of 1000 executions of the scheduling algorithm on a four processor system with varying numbers of task. The task sizes are chosen randomly each requiring 200-400 units of processing time. Thus, the average completion time of
the simulated scheduling system for 6 tasks on a four processor system is 589 time units.

Note that there are two different optimals depicted in figure 13, namely, (1) the optimal without mid-execution task migration and (2) the optimal with mid-execution task migration. Under the restriction that tasks may not migrate having commenced execution, the simulated scheduling method consistently stays within 10% of the optimal without mid-execution migration. Although a sparse number of tasks distributions do cause worst case performance of the scheduling algorithm, the vast majority of our experiments\(^1\) verify that the algorithm performs within 20% of the optimal using mid-execution migration.

Figure 14 displays the average case scheduling effectiveness of a 100 processor

\(^1\)See appendix B for tables of results.
Figure 14: Average case scheduling with 100 processors and no mid-execution migration.

system. The task sizes are again chosen randomly each requiring 200-400 units of processing time. As can be seen, the scheduling algorithm scales up favorably and again stays within 10% of the optimal solution without mid-execution migration.

4.3.3 With mid-execution migration

Repeated applications of a modified dynamic version of the LPT[16] optimal scheduling method may be used to schedule dynamic tasks. The problem with this approach is that it assumes that task migration consumes no processing time and disregards the processor on which each task's code, stack, and data segments currently reside. Our model assumes task migration does indeed consume processing time (one unit), so that each time a task is migrated, the task requires an extra time unit to complete execution.
A straightforward dynamic LPT implementation may cause all tasks to be migrated after each scheduling decision. In the worst case a new task could arrive at each time unit causing a scheduling decision to be made that migrates all tasks. The only processing accomplished during the next time unit would be task migrations. Enhancements to the dynamic LPT algorithm can be made to ensure that actual application processing, not solely migration processing, is accomplished. These enhancements form a more computationally complex algorithm and even with the enhancements, useless premature migrations still occur making the algorithm sub-optimal for dynamic applications.

In fact, as shown in chapter 2, there exists no optimal solution to dynamically schedule tasks when task migration time is non-zero. To overcome the problem of premature migrations, the simulated scheduling system delays migrating a task until a processor becomes idle. This instance of the scheduler classifies scheduling requests into three categories:

1. Requests for processing by tasks consisting of processes previously loaded onto the multiprocessor.

2. Requests for processing by tasks consisting of processes not previously loaded onto the multiprocessor.

3. Requests for tasks from local schedulers on idle processors.

Type 1 requests are simply placed in the ready queue of the processor on which they have previously been loaded and are handled solely by the local schedulers.
Type 2 requests are scheduled by the global schedulers in a manner identical to the previously described completion time scheduling system without mid-execution migration.

Type 3 requests are submitted by a local scheduler when a subordinate processor of the local scheduler becomes idle. The handling of a type 3 request is the gradient method presented in [28] on a fully connected topology. The local scheduler sends a request for work to a global scheduler. Upon receiving a request for work, the global scheduler broadcasts the request to all other local schedulers. The local schedulers reply to the global scheduler with tasks eligible for migration to the requesting local scheduler. An eligible task is the largest task in the ready queue of a processor with more than one task in the queue. The global scheduler chooses the largest eligible task returned by a local scheduler and initiates the task's migration to the requesting local scheduler.

Task_descriptor and link_structure. The 'task_descriptor' record and the 'link_structure' record are similar to the previous simulation with the exception of two extra fields, termed type and 'return_info', in the 'link_structure'. The 'link_structure' is simulated as an offer data structure. The type field is used to distinguish the type of scheduling request being made. For type 2 requests the 'send_down_link', 'send_up_link', and 'receive_up_link' link all operate identically to the previously described scheduler. For type 3 requests, the 'send_down_link' consists of the global scheduler broadcasting a request for work from the local schedulers. The local schedulers respond to the request by placing information in
the 'return_info' array using a 'send_up_link' call.

RECORD task_descriptor IS
  execution_time : time_type;
ENDRECORD;

RECORD link_structure IS
  load_info : array [num_local] integer;
  type : integer;
  return_info : array [num_local] integer;
ENDRECORD;

The arbitrate function. The 'arbitrate' function for a type 2 request is identi-
tical to the 'arbitrate' function described in the previous section. The 'arbitrate'
function for a type 3 request simply compares the 'return_info', then chooses the
maximum value returned by a local scheduler. This value represents the local
scheduler with the largest task available for migration.

The scheduling_measures, schedule, and relax_constraints functions.
The 'scheduling_measures' function, used only for type 3 requests, computes the
largest migratable task. This function simply chooses the largest task in the ready
queue provided the queue contains more than one task. The 'schedule' function
is used to schedule locally arriving tasks and is also called to unschedule a mi-
grated task. Unscheduling a task is accomplished by removing it from the mul-
tiplexor ready queue. The 'relax_constraints' function is NULL. Sample code for
the 'scheduling_measures' and 'schedule' functions follows:

FUNCTION schedule_measures(request_info : task_descriptor);
  ...
  if (ready_queue_size > 1)
    ex = task with greatest execution time in ready queue;
    return(ex);
  endif;
endschedule_measures;
FUNCTION schedule(request_info : task_descriptor);
...
if (request_info.type = unschedule)
    remove_from_queue(request_info);
else /* request_info.type = schedule */
    if (request_info.type = locally_arriving)
        lock(load_info);
        load_info[my_id] = load_info[my_id]
            + request_info.execution_time;
        unlock(load_info);
        ready_queue_size = ready_queue_size + 1;
    endif;
endif;
endschedule;

Queuing. The multiplexor queues are ordered in a largest remaining execution
time first manner. Each time unit the multiplexor schedules the task with the
greatest remaining execution time for the next time unit. The global scheduler’s
wait queue is ordered by in a largest execution time first. These requests are
followed by idle processor requests in a FIFO ordering.

4.3.4 Measured efficiency and effectiveness

Efficiency. Scheduling overhead is extremely low in comparison to task migration
time. All measures for type 2 tasks remain equivalent to the measures derived in
section 4.3.2. Type 3 requests must be broadcast by the global scheduler which
requires 1.81 milliseconds of processing time. Each local scheduler must respond
to the global request requiring 0.79 milliseconds. After collecting all responses
from the local schedulers, the global scheduler chooses the task to migrate using
1.16+(0.05·P) milliseconds of processing time, where P is the number of processors.
Then the corresponding load values are updated by the global scheduler requiring
0.56 milliseconds. Finally, the task is migrated which requires over 13 milliseconds. Thus, the scheduling overhead for a medium rate task (100 millisecond), medium size (4K) is 16.7% which is acceptable.

Effectiveness. No tasks are migrated unless a processor becomes idle in the simulated scheduler. The dynamic nature of this class of applications makes it unwise in many situations to prematurely migrate a task as demonstrated by the following example. Considered is a three processor system that initially receives four tasks to schedule. Three of the tasks require six units of processing time and each of the three tasks' code and data segments reside on a separate processor. The fourth task requires seven time units and its code and data segments are stored on processor 1. An optimal schedule for this scenario completes all tasks in nine time units and one such solution is illustrated in figure 15. Suppose the
Figure 16: Average case scheduling with mid-execution migration on a 4 processor system.

depicted optimal schedule is developed by the scheduler and at time unit four, two more tasks arrive each needing 7 units of processing time and on processors 2 and 3 respectively. The above scheduling scheme prematurely migrates a task and cannot perform optimally as compared to an algorithm that delays task migration.

All workloads generated for the scheduling system not using mid-execution migration are also scheduled using mid-execution migration. Likewise, task migration is assumed to consume one unit of processing time at the target processor. The scheduling overhead is assumed to be included in the task migration time. The results from the different arrival rates and task execution time distributions are again statistically insignificant.

Figure 16 compares the simulated scheduling algorithm with mid-execution task migration to the optimal schedule on a four processor system. The figure
Figure 17: Average case scheduling with mid-execution migration on a 100 processor system.

displays that the simulated scheduler remains within 10% of the optimal on the average for the plotted distributions. Tables for various distributions, included in appendix B, display that the scheduler performs consistently above the worst case values.

Another fact ascertained from figure 16 is that as the number of tasks grows the simulated algorithm approaches the optimal. In fact, all of the experimentation with a fairly reasonable number of tasks (over 3 times the number of processors) showed our simulated scheduling system performs within 2% of the optimal.

Figure 17 shows the average case scheduling compared to the worst case, and scheduling without mid-execution migration for a 100 processor system. The figure indicates that the results of the 4 processor experiments scale up to larger multiprocessor systems. For results from other load distributions and systems with
other numbers of processors see appendix B.

The simulation results of the completion schedulers verify that efficient and effective dynamic schedulers can be implemented for completion time applications. Describing the schedulers in terms of the framework displays that the framework captures the important features of such efficient and effective schedulers.

4.4 Summary

In this chapter we developed efficient, effective scheduling systems for real- and non-real-time applications. Measurements indicate that the implemented schedulers are efficient and effective for medium rate (100 millisecond) and slower tasks.

The schedulers in this chapter are developed using the framework developed in chapter 3. It is clear that the framework describes the essential elements of flexible dynamic scheduling systems. Specifically, one interesting result is that a single scheduling framework may be used for the generation of schedulers that optimize entirely different objectives and constraints – completion time and deadlines. Also, the framework permits the application to supply either estimates or actual values of execution time when scheduling completion time applications. Additional objectives and constraints (i.e., memory requirements, other resources, etc.) are shown to be simple modifications to the implemented schedulers. In addition, the link data structure of the framework may be implemented on the multiprocessor such that the resulting scheduler can take advantage of the multiprocessor's shared memory.

Since on-line schedulers are desirable for dynamic applications, we develop
efficient heuristic schedulers as opposed to the computationally complex static algorithms in [13, 27]. The presented algorithms are measured and shown more efficient than other algorithms such as bidding [41, 58, 59, 65, 37]. The algorithms' worst and best case overheads are compared to the measured overheads. Measured overheads consistently approximate best case efficiencies, thereby further demonstrating the schedulers' efficiency. Comparisons of the implemented schedules to optimal schedules prove the heuristics used perform very near optimal schedulers.

The next chapter develops dynamic schedulers for deadline and completion applications for a non-shared memory machine. This demonstrates the scheduling framework may be used with a different class of parallel architectures.
CHAPTER V

DYNAMIC SCHEDULING ON NON-SHARED MEMORY MULTICOMPUTERS

This chapter contains implementations of the scheduling framework for use with a non-shared memory multicomputer. These implementations confirm that the scheduling framework contains the necessary features for developing efficient and effective on-line schedulers for real-time and non-real-time applications on a shared as well as a non-shared memory architectures. Again, two application classes—deadline and completion time—are used to demonstrate the framework's flexibility.

The 'link' structure of the framework is shown to be easily mapped to the underlying architecture of the non-shared memory machine. In conjunction with the mapping, computationally non-complex schedulers are implemented and evaluated.

Scheduling effectiveness is demonstrated by measuring the schedulers' average case behaviors and comparing them to either optimal schedules or to lower bounds on schedules (since developing an optimal schedule becomes computationally intractable as shown in chapter 2). Using the efficiency and effectiveness measures, limits depicting when schedulers become ineffective due to task migration and
Scheduling latencies are ascertained.

Scheduling with mid-execution migration is compared to scheduling without mid-execution migration. This comparison demonstrates that the use of mid-execution migration does not always increase a scheduler's effectiveness.

The schedulers presented in this chapter represent the initial dynamic and distributed scheduling systems which schedule tasks requiring execution on different sized substructures of a target machine. To efficiently compute these schedules, a novel scheduler to machine mapping is used to effectively utilize the processing power of the target machine. Finally, this is the first implemented scheduling system where the information flow within the scheduling system may be dynamically altered to enhance scheduling effectiveness (as displayed below in figure 18).

The non-shared memory machine explored in depth is the Intel hypercube. The reason for experimentation on the hypercube, as opposed to another non-shared memory machine, is the availability of the machine for experimentation. Specifically, a 32 node Intel 80286-based processor hypercube running a modified version of the Intel iPSC/1 operating system is used for all experimentation and measurements. All presented schedulers are transferable to other non-shared memory machines by simply altering the 'link' parameter to match the underlying architecture.

The applications for which the scheduler is designed consist of dynamically arriving tasks that request various size subcubes. The tasks are assumed to be highly independent and, therefore, never block and wait for communications from
other tasks. All inter-task communication is assumed to be performed at the start and end of a task. Processes within a task are scheduled concurrently by the application programmer. Scheduling processes within a task exhibiting communication constraints is an important research issue, but, as stated previously, the issue is not addressed in this thesis. Accordingly, the application programmer is assumed to be responsible for intra-task scheduling, while the scheduling system performs the inter-task scheduling.

For deadline applications, each task supplies the scheduling system with its worst case execution time. This value represents the maximum amount of processing time the task could possibly require to complete execution. Along with the worst case execution time, a task also provides the scheduler with the size of the subcube required to efficiently execute the task. For completion time applications, only an estimate of the task's execution time and required subcube size are needed.

With these assumptions in mind, the parameters to the scheduling framework common to both deadline based and completion implementations are described. The most striking modification from the scheduling systems described in chapter 4 is to the link distributed type portion of the framework. The alterations are due to the different underlying architecture. The subsequent sections describe specific implementations of the framework for both deadline and completion applications on the iPSC/1 hypercube. After each description, evaluations of the resulting scheduling efficiency and effectiveness are presented.

**Numlocal and Numglobal.** Both of the implementations use one global
scheduler and 32 local schedulers (one per processor). Once again, the number of nodes in the experimental hardware is small enough to warrant the use of only a single global scheduler. One local scheduler and one multiplexor are assigned to each processor to evenly distribute the scheduling overhead and to minimize the local scheduler to multiplexor communication overhead.

**Queues.** The local and global scheduler queues are implemented in a straightforward manner on the same processor as the owner of the queue (i.e., the queue for a local scheduler resides on the same node as the local scheduler). This placement minimizes queue access time.

There are two major differences between the multiprocessor scheduling systems and the non-shared memory scheduling systems. The most obvious difference is caused by the lack of shared memory. This factor prevents the efficient use of global shared memory to broadcast and arbitrate scheduling decisions. The second difference is the manner in which groups of processes composing a single schedulable task are handled. In the multiprocessor scheduling system, a task that makes a request for X processors to the scheduling mechanism is provided with any set of X processors. The multiprocessor scheduling systems assume that each task executes with equal efficiency on any set of processors. A task requesting a set of processors on our version of the hypercube architecture is assumed to implicitly request that the processors be part of the same subcube to minimize intra-task communication delays. Only sets of processors sharing a common subcube of the size signified by the scheduling request are eligible to execute the request.
Links. Due to the absence of shared memory, the 'offer' data structures used for the multicomputer cannot be implemented on the hypercube. Instead, a global scheduler builds a full binary tree of logarithmic depth to communicate scheduling requests to local schedulers and uses the same tree to receive scheduling information from the local schedulers. The processor on which the global scheduler executes is the root of the scheduling tree. Each leaf node of the tree represents a local scheduler. The intermediate nodes of the tree represent the combine portion (explained in section 5.1.4) of the link distributed type. Figure 18 shows an example of three scheduling trees of an 8 processor hypercube with roots at node 0. This communication structure is used by the 'send_down_link' and 'send_up_link' operations when communicating a scheduling offer to the local schedulers and when receiving scheduling information from the local schedulers respectively.
Given a specific scheduling tree, the `send_down_link` operation sends a scheduling request down the tree, effectively broadcasting it to each local scheduler. This is the fastest method of broadcast on the iPSC/1 hypercube. The local schedulers respond to the offer through the `send_up_link` operation sending scheduling information toward the global scheduler. The intermediate nodes of the link distributed type receive scheduling information from their child nodes and use the `combine` function to remove extraneous scheduling information. After applying the `combine` function, the node forwards the compacted scheduling information to its parent node. Once the scheduling information describes a subcube identical in size to the scheduling request’s required size, the information is sent directly to the global scheduler.

Measurements for both task migration costs and scheduling overhead are evaluated on the hypercube. The measurements are derived in a manner identical to the measures taken on the multiprocessor. The granularity of the node clocks on the hypercube is 5 milliseconds. All timings of scheduling code are accomplished by executing the code 1000 times while no other processing or I/O is being done. The scheduling overhead can once again be broken down into two distinct categories:

**Task migration latency** – which consists of the time required to move the task’s state (execution stack, code and data segments), whenever the assigned subcube is different from the subcube on which the task is currently located.

**Scheduling latency** – the total time spent in the scheduler’s code consists of

1. the time required by a global scheduler for making the initial offer, (2)
the delay incurred by waiting for a response from some local scheduler, (3)
the time used by the global scheduler for choosing the local scheduler, and
(4) the scheduling of the task by the local scheduler. Best case scheduling
consists of one broadcast message, an immediate positive response to the
scheduling request, followed by a broadcast of the scheduling decision and
the actual scheduling of the request. Worst case scheduling latency incurs
maximal delay between the first offering of the request and its acceptance or
rejection by local schedulers. In the system presented, worst case latency has
an upper bound (timeout value) determined by the application programmer
(set at 30 milliseconds in the tests).

Task migration latency is measured at \((3.8 \cdot K) + (3.335 \cdot K \cdot H)\) milliseconds,
where \(K\) is the maximum number of kilobytes of memory required by the task on a
single processor and \(H\) is the number of hops the process is migrated. This results
in a latency of 41.868 milliseconds for performing the parallel migration of a 4K
task to a subcube two hops away.

5.1 Deadline scheduling on the hypercube

The ‘task_descriptor’ for the non-shared memory deadline schedule contains the
task’s release time, deadline, worst case execution time, type (hard or soft deadline),
and required subcube size. A copy of this record is included in the
‘link_structure’ along with an array of possible execution intervals. During a
‘send_down_link’ the scheduling information is broadcast from the global sched-
uler to the local schedulers using a scheduling tree. This broadcast requires 9.3 milliseconds.

The local schedulers respond to the global scheduling request via the scheduling tree using the `send_up_link` function. The latency of a scheduling decision is broken down into four distinct parts: (1) the broadcasting of the scheduling request, (2) the initial response of each local scheduler (the `scheduling_measures` function), (3) the communication time required to forward information toward the root, and (4) the combining of the information by intermediate nodes on the scheduling tree (the `combine` function).

5.1.1 The arbitrate function

For hard deadline tasks the `link_structure` supplies the global scheduler with possible subcubes that can execute the scheduling request between its release time and deadline. The `arbitrate` function for a hard deadline task chooses the first favorable response received from the `link_structure`. A favorable response is one that can entirely execute the task guaranteeing completion before its deadline. All local schedulers are notified of the global scheduler's decision. After notification, the task is migrated and scheduled and any remaining scheduling information within the scheduling tree is discarded.

For soft deadline tasks, the `arbitrate` function chooses any subcube responding to fully execute the task before its deadline. When no subcube responds favorably to the task, the subcube able to complete the task at the earliest possible time is chosen. The subcube awarded the task is broadcast to all local schedulers and
the local schedulers awarded the task initiate the task's migration and schedule it. After scheduling a task, or being notified that it is not awarded a task, the local schedulers examine the next scheduling request.

5.1.2 The scheduling_measures, relax_constraints, and schedule functions

The local scheduler performs a 'receive_down_link' to receive a scheduling request from the global scheduler, when no local requests are available. The local scheduler then performs a 'scheduling_measures' function on the globally offered task and sends scheduling information up the scheduling tree toward the global scheduler. The 'scheduling_measures' function requires the local scheduler to check if a hard deadline task can possibly be scheduled between its release time and deadline without preemtping any previously scheduled task. In the case of a soft deadline task, the scheduler simply returns scheduling information. After sending scheduling information, the local scheduler waits for a response from the global scheduler. Upon receiving an unfavorable response, the local scheduler simply examines the next scheduling request. When awarded the request, the local scheduler calls the 'schedule' function to schedule the task.

Bins are again used to maintain the unused processing time on a processor (see section 4.2.4). Instead of a single set of bins coupled with an earliest deadline first multiplexing strategy, two sets of bins are maintained per processor. The first set of bins, termed 'unused_bins' in the program example, contains unused processing time and is used by the 'scheduling_measures' function. The second set
of bins, termed 'used_bins' in the program example, contains the schedule which is executed by the multiplexor. The intersection of the 'usedbins' and 'unused_bins' is continually empty while their union represents a complete time line. Both sets of bins are maintained and updated by the 'schedule' function. Initially the 'usedbins' are empty, while the 'unused_bins' consist of one bin with a start time of zero and an end time of \( \infty \).

Upon receiving a scheduling request, the local scheduler calls the 'scheduling_measures' function. This function scans the unscheduled set of bins between the request's release time and deadline. While collecting information to send toward the global scheduler, the function creates a third set of bins representing the available processing found in the 'unused_bins' between the request's release time and deadline for a hard deadline request. For soft deadline requests, the created bin set represents all unscheduled processing time after the request's release time. The reason this extra information is returned with the soft deadline request is that, by definition, soft deadline requests may be scheduled to execute after their deadlines. The newly created set of bins is communicated to the global scheduler through a 'send_up_link' call. After passing the scheduling information towards the global scheduler, the local scheduler waits for a response from the global scheduler.

FUNCTION schedule_measures(request_info : task_descriptor);
   ...
   bin = unused_bins[firstUnused_bin];
   while (bin.end_time < request_info.start_time) do
      bin = unused_bins[bin.next];
   endwhile;
   if (bin.end_time > request_info.start_time) then
      add_bin_to_reply(bin);
   endif;
   ...
while (bin.end.time < request_info.deadline) do
    add_bin_to_reply(bin);
    bin = bin.next;
endwhile;
if (bin.start.time > request_info.deadline) then
    add_bin_to_reply(bin);
endif;
if (request.info.type = SOFT.DEADLINE) then
    while (bin != last_unused.bin)
        add_bin_to_reply(bin);
        bin = unused_bins[bin.next];
    endwhile;
endif;
endschedule_measures;

If the response from the global scheduler indicates that the local scheduler has been awarded the request, the local scheduler schedules the request. Accompanying the response from the global scheduler is a set of bins representing the intervals during which the request is to be scheduled. This represents a subset of the list sent from the node in response to the scheduling request. The 'schedule' function uses the bins received from the global scheduler to update both the 'unused_bins' and the 'used_bins'. The updates are accomplished by decreasing the appropriate amounts of available time in the unscheduled bins (possibly creating more bins) and increasing the number of scheduled bins by adding the request to the schedule. Tasks are scheduled to complete execution in the earliest possible time when the intervals returned by the global scheduler are greater than the total execution time required by the task.

FUNCTION schedule(request_info : task_descriptor,
    time_list : array);
...
bin = unused_bins[first_unused_bin];
i = 1;
while (i <= time_list.end)
while (bin.end.time < time_list[i].start.time)
    bin = unused_bins[bin.next];
endwhile;
update_bin(bin, time_list[i].start.time);
i = i + 1;
endwhile;
bin = used.bins[first_unused.bin];
i = 1;
while (i <= time_list_end)
    while (bin.end.time < time_list[i].start.time)
        bin = used.bins[bin.next];
    endwhile;
    add_bin(used_bins, time_list[i].start.time, request_info.task.Id);
i = i + 1;
endwhile;
endschedule;

The 'schedule' function ignores the request's deadline when scheduling it. The function uses information solely from the global scheduler. Since no deadlines require adjustment by local schedulers, no 'relax.constraints' function is needed. Soft deadline processes are always scheduled to not interfere with previously scheduled tasks by scheduling them only during previously free intervals of times.

5.1.3 Multiplexor ordering

The multiplexor uses the schedule queue when arbitrating what to schedule for the next time unit. The schedule queue consists of a series of bins, termed 'used_bins' above, each contains a start time and end time representing an interval of time. Also associated with each bin is the task to execute during the interval. The multiplexor checks the the first element in the list of scheduled bins for possible execution each time unit while the processor is idle. Once the first element becomes eligible for execution (the start time of the bin has been reached), the task specified by the bin is executed for the next (end time - start time) units of time. After the task is
executed, the bin is removed from the scheduled list and the multiplexor idles until the next bin's start time has been reached. Thus, tasks are scheduled for specific intervals and once scheduled they will never be preempted. The multiplexor, in combination with the 'scheduling_measures' and 'schedule' functions, ensures that any deadline guaranteed by the 'scheduling_measures' function is met.

5.1.4 The combine function

The 'combine' function of the link distributed type receives two lists of possible execution intervals for the global request from two independent subcubes smaller than the requested subcube size. The global scheduler requires information specifying when both subcubes can simultaneously schedule the request. To offload some of the global scheduler's processing time, the 'combine' function intersects the two lists and forwards the result toward the global scheduler. If the correct subcube size for the request is present after the combine, the information is sent directly to the global scheduler. Otherwise, depending on the scheduling tree used, either more information is received and intersected, or the information is sent to another intermediate node to be further intersected. This relieves the global scheduler of list intersecting and distributes the intersecting of lists across the system.

5.1.5 Measured efficiency and effectiveness

Efficiency. The overhead generated by the local and global schedulers is extremely small compared to the communication overhead caused by using the operating system's (iPSC/1) communication facility. The cost of sending a scheduling request
or scheduling reply from one node to another is measured at 1.8 milliseconds. This is by far the largest contributing factor to the scheduling overhead as is shown by the following measurements of the implemented scheduler's overhead.

A global scheduling request must first be broadcast to all the local schedulers. Preparing the message for broadcasting requires .07 milliseconds of processing time by the global scheduler. A broadcast tree is the fast method to transmit information to all nodes on the experimental hypercube system and requires 9.3 milliseconds to broadcast information to all nodes. After a local scheduler receives the scheduling request, it requires \(0.20 + (0.15 \cdot B)\) milliseconds to perform the 'scheduling_measures' function (where \(B\) is the number of bins examined) and prepare the scheduling information to be forwarded toward the global scheduler. Sending information back to the global scheduler requires 1.8 milliseconds for each hop. The 'combine' functions, performed to coalesce scheduling information on the way to the global scheduler, require \(0.04 + (0.08 \cdot B)\) milliseconds of processing time. Thus, the total scheduling decision latency with an average of a hard deadline task requesting a subcube of size \(S\) with 10 bins examined by each 'scheduling_measures', 'arbitrate', and 'combine' function is \(20.0 + (0.84 \cdot \ln(S))\) milliseconds with 18.3 milliseconds due to the iPSC/1 communication facility.

The scheduling decision is followed by actually scheduling the task. Again a broadcast tree is used to notify the local schedulers of which scheduler(s) have been awarded the task. The scheduling of the task by the local scheduler requires from .76 to 1.03 milliseconds depending on the number of new 'used_bins' and
'unused_bins' that require creation and/or combination. Thus, in the worst case it requires 10.33 milliseconds to schedule a task after making a scheduling decision—9.3 milliseconds again due to the iPSC/1 communication facilities.

A best case scheduling scenario occurs when a hard deadline request is broadcast and immediately answered positively by a single subcube. It is then scheduled before any other local schedulers examine the request. Therefore, in the best case for a task of size \( S \), the scheduling system first performs a scheduling request broadcast. This is followed by an immediate response from one set of local schedulers representing the request's required subcube size, and finally a scheduling decision broadcast. The two broadcasts each require 9.3 milliseconds. Each concurrently calculated scheduling measures function requires .35 milliseconds. The concurrently performed 'combine' functions require \( \ln(S) \cdot .12 \) milliseconds and also represent a message send requiring 1.8 milliseconds. Finally, the 'schedule' function performed simultaneously by each local scheduler requires .76 milliseconds. Thus, the total latency is 

\[
19.36 + \ln(S) \cdot 1.92 \text{ milliseconds.}
\]

The worst case scheduling overhead occurs when a soft deadline request cannot be scheduled before its deadline. In this case, all local schedulers reply to the scheduling offer and all replies are examined by the global scheduler. The 'arbitrate' function of the global scheduler chooses the subcube reporting the earliest possible completion time for the request. The additional latency incurred by this type of scheduling request lies in the fact that the global scheduler must wait for all local schedulers to respond to the request. In fact, this overhead is identical to the
overhead caused by using bidding as described in [41]. Giving the global scheduler a maximum allowable wait helps to cure this problem, but does not alleviate all of the extra latency. A timing of this case shows that there are still only \( \ln(S) \) serial message sends causing the identical communication latency as the best case and global scheduler's processing overhead is only slightly larger due to comparing all responses when arbitrating. As stated earlier, the increase in overhead is minimal, while the increase in latency due to waiting for responses from all local schedulers is unbounded.

With the above timing information in mind, the scheduling system is easily shown insufficient for high-rate tasks with execution times of under 20 milliseconds. This is because the processing time consumed by migrating a 3K task to a neighboring node requiring under 20 milliseconds of execution time exceeds the processing time required to execute the task. Thus, the processing time used to migrate the task could be used to entirely execute the task. On top of that, the scheduling overhead also surpasses the execution time of the task. Thus, migration of such tasks should not be considered. The critical point at which the application begins benefitting from the implemented scheduler is when tasks require at least 400 milliseconds of processing time.

The following measures are taken with the global scheduler operating on a single node along with a local scheduler and multiplexor. All other nodes each contain a local scheduler and multiplexor. Different task arrival rates and processing time distributions again caused no significant deviations in the scheduling efficiency or
The scheduling overhead for different average execution times is shown in figure 19. Overhead, as defined previously, is the percentage of time spent scheduling vs. executing application code. This displays the scheduling overhead is unacceptable for tasks average task size of 100 millisecond and under due to the fact that the scheduling system utilizes over 40% of the processing time. The scheduling system uses a reasonable amount of processing time for applications with average tasks execution time of above 400 milliseconds.

Figure 20 consists of two graphs depicting the worst case scheduling overhead of applications consisting of tasks with average execution time of 1 second. One figure displays the increased scheduling overhead when the local schedulers require scanning different numbers of bins, while the second graph depicts the additional
overhead for each successive resource required. Both figures display that for a reasonable number of bins per processor (under 40) and a reasonably number of schedulable resources (under 5) the additional scheduling overhead is minimal. The relatively small increases in scheduling overhead are due to the fact that the dominant contributor to the scheduling overhead is the communications overhead. Therefore, the communications overhead is still the dominant overhead when scheduling on systems with 4 schedulable resources and 40 bins per processor.

The measured scheduling efficiency lies between the best case and worst case overhead and is displayed in figure 21. The measures are taken by generating tasks with random subcube sizes from a normal task execution time distribution with mean at one second and a standard deviation of 50 milliseconds. The generator maintained desired system loads from 60% to 100%. Additional measurements
Figure 21: Worst case, best case, and measured overheads in a 32 processor system with 10 bins per processor and 1 second average execution time.

reported in appendix B verify the measured scheduling overhead is acceptable for different numbers of schedulable resources and average numbers of bins scanned.

Effectiveness. For ease of explanation and evaluation, all tasks are considered to be of approximately the same size so that each task migration requires the same amount of time. The migration of a task is considered to consume one unit for processing time on both the home and target nodes. Each task requests an integer amount processing time in terms of this time unit. For example, if a task requesting ten units of processing time, then the amount of execution time it requires is ten times the amount of processing time required to migrate it. Effects that task migration has on intermediate nodes in its migration path are dismissed due to routing chips available for more recent hypercubes.

The measured effectiveness for various loads is displayed in figure 22. The loads
Figure 22: Measured effectiveness for various loads.

are generated using an average task execution time from a normal distribution with mean of 10 units and standard deviation of 5 units. The subcube size requested by each task is the same while the system load varies. In this case, the proposed scheduling algorithm performs admirably scheduling 100% of the processes in a 70% loaded system 99.8% of the time. The figures are attained by generating 1000 loads and using the system to schedule them.

Figure 23 displays the scheduling overhead with the cube size requested by a task chosen from a random distribution. Although the proposed scheduling algorithm operates optimally for lightly loaded systems (i.e., systems under 50% loaded), the diagram clearly displays the algorithm sub-optimal for systems loaded over 70%. The main reason for the sub-optimal performance is that cube fragmentation that occurs.
Figure 23: Measured effectiveness for various loads with multiple cube sizes.

Figure 24: Measured effectiveness for various loads with multiple cube sizes.
This is especially pointed out in figure 24 where the identical tasks as scheduled in figure 23 are ordered earliest deadline first subordered by largest subcube required. In this case the scheduler performs extremely close to optimal, 99.5% of the time, even when the load reaches 90%. These are extremely good results, but they are under the unrealistic assumption that the application program presents the scheduler with ordered scheduling requests.

The cube fragmentation suggests that the scheduling system not only bases scheduling on whether a deadline can be met, but also on whether cube fragmentation can be avoided. Cube fragmentation can be partially avoided using computationally complex algorithms described in [12, 5]. Both of these systems rely on a single centralized scheduler. Since an on-line, efficient, distributed scheduler is desirable for the applications being considered, another method to reduce fragmentation is employed.

To avoid cube fragmentation, another layer on the 'send_up_link' portion of the scheduling system is added. This final combine node receives information from its children representing the subcube of the same size as the request. The lists received designate the possible times that the task can be scheduled on the subcubes. It then exclusive ORs each list with the complement of the other. The results of the exclusive OR's represent moments at which the subcube is free and the opposing subcube is busy.

The additional layer views the pairs of subcubes as a single subcube and attempts to keep the entire subcube busy and idle during the same intervals. The
node returns to the global scheduler the result of the exclusive ORs along with the intervals at which both subcubes are idle.

A favorable response to the global scheduler is a reply that signifies that a subcube can entirely schedule the task while its opposing subcube is busy executing other tasks. When no such favorable response occurs the global scheduler chooses the subcube that can execute the task before its deadline and has the greatest amount of time which its opposing subcube is busy.

The scheduling overhead in this system is slightly higher than the earlier described system that does not combat cube fragmentation (actual figures are included in appendix B). The minor drop in efficiency is due to the global scheduler not being able to immediately accept as many offers. The gain in effectiveness outweighs the loss in efficiency as can be ascertained by figure 25. The results
indicated in the figure are using identical loads used to derive the values for figure 23.

5.1.6 Deadline scheduling using mid-execution migration

In this section we explore if mid-execution task migration significantly increases the effectiveness of the scheduler. A simple change to the local and global schedulers allows the system to schedule tasks for various time intervals on different subcubes using mid-execution process migration. This section begins with a description of the modifications required, then the loss of efficiency and gain of effectiveness are discussed.

The arbitrate and schedule functions

The modified 'arbitrate' function checks not only for a subcube that can entirely execute a task between its release time and deadline while its opposing subcube is busy, but also maintains two lists of unscheduled intervals returned by subcubes. When a subcube is found that can entirely execute a process while its opposing subcube is busy, then the subcube is awarded the task.

Otherwise, the response is added to the two lists. One list, termed list 1, represents free intervals from each responding subcube when that subcube's opposing subcube is busy. The other list, termed list 2, represents free intervals when the opposing subcube is not busy. Both lists 1 and 2 begin empty and each time a new pair of list is received from a subcube, each received list is intersected with the appropriate list (1 or 2). The original received lists are then exclusively ORed
with the result of the respective intersections. The results of these operations are added to the respective current list. The net result of these operations is that the intervals of free time returned to the global scheduler not in lists 1 and 2 are added to the lists.

When all subcubes have responded and no subcube can entirely execute a task while its opposing subcube is busy, the global scheduler checks if a single subcube can entirely schedule the task. If more than one such subcube exist, then the global scheduler awards the request to the subcube that can schedule the task with the greatest amount of time where its opposing subcube is busy.

If no single subcube can guarantee the task’s deadline, the global scheduler checks if the task can be scheduled using mid-execution migration. This check is made by first scanning the list 1 and summing the available processing time. If this sum is enough, the task is scheduled on the appropriate intervals in an earliest interval first manner. If their sum does not represent enough processing time to completely schedule the task then list 2 is intersected with list 1 and exclusively ORed with the result of the intersection. If the sum of the free intervals in list 1 and the revised list 2 is enough to entirely schedule the request, the global scheduler schedules it. The schedule first uses all the intervals from list 1 then intervals from list 2 in an earliest first manner. Each time the request is to be migrated, time units are added to the execution time of the request to compensate for the migration processing. Finally, when a hard deadline request’s deadline can not be guaranteed, the scheduling system refuses to execute the task and notifies the
When a task can be scheduled using mid-execution migration, a list of bins defining when it is scheduled on the various subcubes is sent to all local schedulers. When scheduling tasks in this system, the local scheduler must examine the list and remove inappropriate entries (i.e., entries representing intervals on other subcubes).

This is accomplished by modifying the 'schedule' function in the following manner:

```c
FUNCTION schedule(request_info : task_descriptor,
                 time_list : array);
...
    bin = unused_bins[first_unused_bin];
    i = 1;
    if (time_list[i].subcube != my_subcube)
       i = i + 1;
    endif;
    while (i <= time_list_end)
       while (bin.end_time < time_list[i].start_time)
          bin = unused_bins[bin.next];
       endwhile;
       update_bin(bin, time_list[i].start_time);
       i = i + 1;
       if (time_list[i].subcube != my_subcube)
          i = i + 1;
       endif;
    endwhile;
    bin = used_bins[first_unused_bin];
    i = 1;
    if (time_list[i].subcube != my_subcube)
       i = i + 1;
    endif;
    while (i <= time_list_end)
       while (bin.end_time < time_list[i].start_time)
          bin = used_bins[bin.next];
       endwhile;
       add_bin(used_bins, time_list[i].start_time, request_info.task_id);
       i = i + 1;
       if (time_list[i].subcube != my_subcube)
          i = i + 1;
       endif;
    endwhile;
endschedule;
```
This method slightly slows the ‘schedule’ function of the local schedulers ranging from .97-1.56 milliseconds which is still very reasonable compared to the communication latencies. The method also affects the global scheduler’s efficiency in two ways. First, when a subcube finds a task unschedulable in the non-mid-execution migration system, the lack of feasible scheduling is reported to the global scheduler. Now every subcube reports all unused intervals between the requests release time and deadline to the global scheduler. The intersection and exclusive ORing of the two lists .04 + (.08 \cdot B) milliseconds and is performed \(\frac{\text{cub\text{size}}}{\text{subcube size}}\) times. The ‘arbitrate’ function requires .06 + (.12 \cdot B) milliseconds to scan the list and develop the schedule to be sent to the local schedulers. Thus, the efficiency of the scheduler is slightly lower and the scheduling latency is slightly higher, but still compares favorably to the communication and task migration latency (precise measured overhead is included in appendix B).

All sample workloads presented to the scheduler not using mid-execution migration are also scheduled using mid-execution migration. The increase in effectiveness is shown in figure 26. This displays that on the average adding mid-execution migration to the scheduler only slightly enhances the scheduler’s effectiveness.

5.2 Completion scheduling on the hypercube

The completion scheduling not using mid-execution migration operates identically to the deadline scheduler in which every task has an unattainable soft deadline. In this case the global scheduler receives scheduling information from all the subcubes and picks the subcube that can complete the tasks execution at the earliest time.
The scheduler need not build lists 1 and 2 described in the previous section.

When more than one subcube can complete the task at the same time it chooses the subcube with the busiest opposing subcube. This choice is made to reduce cube fragmentation. Since a practically identical scheduling system is used, the scheduling overhead created by this system is virtually the same as the previous scheduler's and the efficiency measurements are included in appendix B.

The effectiveness of the scheduler is compared to the optimal schedule derived by in [22]. Although optimal scheduling may be two time units more than the polynomial algorithm described in [22] due to the cost of migration, the solution gives a reasonable estimate of the optimal.

Figure 27 compares completion times for sample loads to the pseudo-optimal value. The pseudo-optimal value is the maximum of the following two values:
Figure 27: Measured effectiveness for number of tasks with multiple cube sizes.

1. The largest task execution time.

2. The sum of the task execution times divided by the number of processors.

The each task of a load has a random execution time between 5 and 20 time units and requires a random size subcube. The figure indicates that for applications generating a reasonable number of requests (over 50) the scheduling algorithm performs within 18% of the optimal.

5.2.1 Completion scheduling with mid-execution migration

As without mid-execution migration, the global scheduler waits for responses from all local schedulers before making a scheduling decision. The global scheduler's 'arbitrate' function treats every task as a soft deadline task with an unattainable deadline by a single processor. In this case, the global scheduler forms both lists
Figure 28: Measured effectiveness for number of tasks with multiple cube sizes with mid-execution migration.

1 and 2 as described above. It then intersects the lists and removes the result of the intersection from list 2. After removing the result of the intersection from list 2, it then combines the lists into a single chronological list. Finally, the request is scheduled at the earliest portion of the list.

The same generated tasks that are used with the system that did not employ mid-execution migration are also scheduled by this system. Figure 28 compares the optimal to the schedulers with and without migration. The scheduler with mid-execution migration only slightly outperforms the scheduler without mid-execution migration.
5.3 Summary

This chapter displays that efficient and effective schedulers can be developed for real- and non-real-time applications on a non-shared memory machine. The schedulers are described using the framework to verify that the framework contains the essential features for developing such scheduling systems. Task migration is timed to ascertain limits on when such dynamic schedulers are feasible.

A novel mapping of the 'link' data structure onto the hypercube is described. Along with the mapping, the implemented schedulers use non-computationally complex constraints and objectives as in chapter 4. Timings of the schedulers indicate that the non-complex algorithms and the 'link' data structure effectively utilize the hypercube architecture.

The schedulers are shown effective by using them to schedule sample workloads and comparing the developed schedules to optimal scheduling and estimates of optimal scheduling. Incorporating mid-execution task migration into the scheduling system is shown to only slightly enhance scheduling effectiveness on the average case while at the same time slightly decreasing the scheduler's efficiency. Thus, in some cases, using mid-execution migration needlessly complicates the scheduler.

Finally, all the implemented schedulers of chapters 4 and 5 are based on the framework described in chapter 3. This supports the postulate that the framework is a sufficient foundation from which to build efficient and effective schedulers for multicomputers.
CHAPTER VI

Conclusions and Future Research

Applications written for concurrent processing systems require effective scheduling to efficiently utilize the underlying architectures. The scheduling framework presented in this dissertation is shown to be efficient and effective for different classes of applications and multiple parallel architectures.

6.1 Conclusions

There are two different types of results in this thesis:

- We developed a common scheduling framework general enough to be used with all types of dynamic applications.

- We design, implement and measure efficient and effective scheduling algorithms for two important classes of dynamic applications.

The framework presented in this dissertation is postulated as a sufficient foundation on which to specify distributed schedulers. It consists of modules distributed across the computing system to efficiently utilize the underlying architecture. The
flexible framework description represents the initial encapsulation of both scheduler to machine mapping and efficient scheduling heuristics.

One method of displaying the framework's flexibility is through using the framework in conjunction with different operating systems constructs. The framework is shown to operate efficiently with various message and process migration facilities.

Extensions to the scheduling framework, such as scheduling multiple resources and using different 'scheduling_measures', 'schedule' and 'relax_constraints' functions further confirm that it is a sufficiently general foundation for building dynamic schedulers with varying constraints and objectives.

The flexibility of the framework is further demonstrated by building schedulers for two different classes of applications - deadline and completion time. This represents the first use of a single scheduling framework for two entirely different sets of constraints and objectives.

Finally, the framework's flexibility is displayed through mapping it to two different multicomputers - a multiprocessor and a hypercube. Flexibility of the implemented schedulers is also shown through describing extensions to the system's hardware constraints (e.g., adding memory constraints and scheduling other resources).

The novel 'link' distributed data type is shown to be sufficiently general for the scheduler to multicomputer mappings. In regards to specific implementations, offers are shown to be an efficient use of 'links' in a multiprocessor computer system, while scheduling trees are shown efficient in the hypercube multicomputer.
Thus, the 'link' distributed data type along with the 'combine' function allow the framework to effectively be mapped effectively to arbitrary target architectures.

Comparing the schedules constructed by the implemented schedulers' to optimal schedules verify that our implemented non-computationally complex algorithms are highly effective (usually performing within 5% of an optimal scheduler). The effectiveness measures, along with the efficiency timings, indicate that high performance schedulers can feasibly be developed for parallel computing systems. In fact, the schedulers perform well enough to be run in conjunction with the application, thereby making dynamic scheduling a viable alternative to static scheduling. Limits on average task execution time are investigated to derive when our dynamic schedulers are non-beneficial on the various computing systems (e.g., tasks with average execution time under 100 milliseconds on our experimental multiprocessor and under 400 milliseconds on the Intel iPSC/1 hypercube).

Through our experiments, we established that, in many instances, scheduling using mid-execution task migration is only slightly more effective than not using mid-execution migration. We also discovered that using mid-execution migration leads to more involved scheduling computations dropping the scheduler's efficiency. Thus, in some cases using mid-execution migration did not enhance the scheduler's net effectiveness.

In addition, the following theoretic results are proven to further establish the need for good scheduling heuristics:
• Scheduling to minimize completion time is NP-complete in systems with non-zero migration cost.

• If minimal completion time scheduling with non-zero migration cost is not in P, then deadline scheduling with non-zero process migration cost is not in P.

6.2 Future Work

One area of future work is investigating the framework's mappings onto different non-shared memory architectures to strengthen the hypothesis that the framework is a sufficient foundation for scheduler generation. Parameterizing the link distributed type so that the scheduler to underlying architecture mapping can be automated would aid in this task and also simplify implementing schedulers for various systems. This, combined with a syntax for a scheduler description, would allow the framework to generate a scheduler for a specific parallel machine and class of applications. A system such as the 'topology' compiler, described in [4] could be altered to map the link distributed type, local schedulers and global schedulers to the target architecture. Furthermore, the scheduler's communications system implemented using a 'topology' operating system construct[48], which would enhance the scheduler's efficiency.

Another issue involved in scheduler generation is incorporating communications schedulers into the scheduling system. These schedulers would ensure that highly communicating processes within tasks execute at the appropriate times. Including intra-task communication into the scheduling framework would require an addi-
tional layer of schedulers. This additional level of the scheduling system would work within the described framework and would be responsible for:

- deciding the number of processors a task requires,
- computing the maximum amount of processing time required by the task (for the inter-task schedulers of this dissertation),
- synchronizing the appropriate intra-task processes during task execution, and
- reporting any unused processing time to the inter-task scheduler.

To assist in scheduler generation, all implemented schedulers and future scheduler implementations for other classes of applications should be included in a scheduler library. This library will be used for subsequent scheduler generations. Thus, an applications programmers would specify the type of inter-task scheduler, intra-task scheduler, and system architecture, and from the specifications, a scheduler would be generated for their application. Using effectiveness and efficiency trade-offs, the programmers would have the opportunity to tailor the scheduler to meet their performance criteria.

Performance comparisons between highly effective and highly efficient scheduling algorithms in different scheduling paradigms can be made. These results would further verify that in many real-time instances heuristics outperform computationally complex optimal (or near optimal) scheduling algorithms.

Finally, dynamic substitutions of scheduling functions as proposed in [42] and implemented in [2] could be incorporated into the framework. Using such a scheme
allows the scheduler to dynamically change its functionality to the changing requirements of the application as discussed at the end of chapter 3.
Appendix A

Full language description

The scheduling system is more precisely defined by the following informal semantics. The entire scheduling framework consists of a single concurrent module. A concurrent module contains multiple execution threads (modules) that communicate via intermodule communication mechanisms (distributed types). A concurrent module uses explicit parameters when sharing information with encapsulated modules and distributed types. Data is accessible to a concurrent module only by use of explicit parameters. Thus, a concurrent module consists of a section declaring its own parameters and a REPRESENTATION section declaring its encapsulated modules and distributed types. A concurrent module differs from a module only in the fact that it allows multiple threads of execution – one for each module it encapsulates.

Modules are much like Ada[20] tasks in that they encapsulate a single execution thread. In contrast to Ada, modules do not have explicit entry points. Instead, a thread begins execution when the module is created. Furthermore, modules communicate with each other through shared distributed types, which are specified as
parameters. The multiplexors, global schedulers, and local schedulers are modules. The structure of a module is determined at compilation. A module consists of a parameter list, a REPRESENTATION section declaring local variables, and a PROCESS section. Within the PROCESS section the module may call operations supplied by a distributed type.

A distributed type provides operations to modules for which it is a parameter. The internal structure of each distributed data type is determined at compilation time. The link and queue data structure used for communication between global and local schedulers are distributed data types. A distributed type consists of a parameter section, local data section, and a set of operations available to specified modules.

Complete descriptions of the concurrent modules, modules, and distributed types used in this thesis appear below:

```plaintext
CONCURRENT MODULE scheduler
  queue_ordering : function,
multiplexor_ordering : function,
task_descriptor : record,
schedule_measure : function,
schedule : function,
relax_constraint : function,
num_local : integer,
local_sleep : function,
thresholds : function,
notify : function,
arbitrate : function,
num_global : integer,
global_sleep : function,
wait_sleep : function,
link_structure : record,
reply : record,
combine : function,
num_processors : integer,
multiplexor_sleep : function);
```
REPRESENTATION

local_queues : array [num_local] of
  queue(task_descriptor : record,
         queue_ordering : function);

global_queues : array [num_global] of
  queue(task_descriptor : record,
         queue_ordering : function);

multiplexor_queues : array [num_processors]
  queue(task_descriptor : record,
         multiplexor_ordering : function);

links : for i = 1 to num_global link(task_descriptor,
                                     link_structure,
                                     reply,
                                     combine);

multiplexor : for i = 1 to num_processors
  multiplexor(multiplexor_sleep : function,
              multiplexor_ordering : function,
              multiplexor_queues[i] : distributed type);

local_schedulers : for i = 1 to num_local
  localscheduler(task_descriptor,
                 schedule_measure,
                 schedule,
                 relax_constraint,
                 multiplexor_queues,
                 global_queues,
                 local_queues[i],
                 thresholds,
                 local_sleep,
                 notify,
                 reply,
                 links);

global_schedulers : for i = 1 to num_global
  globalscheduler(task_descriptor,
                  arbitrate,
                  local_queues,
                  global_queues[i],
                  global_sleep,
                  wait_sleep,
                  reply,
                  links);

MODULE localscheduler(task_descriptor : record,
                      schedule_measure : function,
                      schedule : function,
                      relax_constraints : function,
                      multiplexor_queues : distributed type,
                      global_queues : distributed type,
                      local_queue : distributed type,
thresholds : function,
local_sleep : function,
notify : function,
reply : record,
links : distributed type;

REPRESENTATION
info : task_descriptor;

PROCESS request_handler IS
loop:
    if not(empty(local_queue)) then
        info = top(local_queue);
        if thresholds(info) then
            if info.relax
                relax_constraints(info);
                insert(multiplexor.queues,info);
                schedule(info);
            else
                if info.global then
                    insert(info,global.queues);
                else
                    notify(info);
                endif;
            endif;
        else
            if schedule_measure(info,reply) then
                insert(multiplexor.queues,info);
                schedule(info);
            else
                if info.relax
                    relax_constraints(info);
                    insert(multiplexor.queues,info);
                    schedule(info);
                else
                    insert(info,global.queues);
                endif;
            endif;
        endif;
    else
        info = receive_down_link;
        if not(thresholds(info)) then
            if schedule_measure(info,reply) then
                send_up_link(info,reply);
                info = receive_down_link;
            endif;
        endif;
    endif;
endif;

endif;
endif;
endif;
endif;
endif;
local_sleep;
endloop;

MODULE globalscheduler(task_descriptor : record,
                        arbitrate : function,
                        local_queues : distributed type,
                        global_queue : distributed type,
                        global_sleep : function,
                        wait_sleep : function,
                        reply : record,
                        links : distributed type);

REPRESENTATION
    info : task.descriptor;

PROCESS scheduling_request IS
loop:
    if not(empty(global_queue)) then
        info = top(global_queue);
        build_link(info);
        send_down_link(info);
        wait_sleep;
        info = arbitrate(receive_up_link(reply));
        send_down_link(info);
        remove(info, global_queue);
        raze_link;
    endif;
    global_sleep;
endloop;

DISTRIBUTED TYPE link(task_descriptor : record,
                       link_structure : record,
                       reply : record,
                       combine : function);

REPRESENTATION
    info : task_descriptor;

OPERATIONS
    build_link(task_descriptor) IS
        build a communication structure;
    raze_link IS
        destroy communication structure;
    send_down_link(task_descriptor) IS
        broadcast info to local schedulers;
    send_up_link(task_descriptor, reply) IS
        send scheduling information back to global scheduler
using the combine function in intermediate stages;
receive_down_link IS
receive request info from global scheduler;
receive_up_link(reply) IS
receive scheduling information from local schedulers;

MODULE multiplexor(multiplexor_sleep : function,
multiplexor_ordering : function,
my_queue : distributed type);

PROCESS multiplex IS
loop:
multiplexor_ordering;
multiplexor_sleep;
endloop;

DISTRIBUTED TYPE queue(task_descriptor : record,
queue_ordering : function);

REPRESENTATION
linked list of task_descriptor; /* or array */

OPERATIONS
insert(task, queue) IS
insert task in queue using queue_ordering;
empty(queue) IS
returns true if queue is empty, false otherwise;
top(queue) IS
returns the index of the element currently on queue top;
remove(task, queue) IS
removes task from queue;
Appendix B

TABLES OF SCHEDULING RESULTS

This appendix contains all raw data used in the figures along with additional data not depicted within the text.
Table 2: Worst case percentages of deadlines met on the multiprocessor.

<table>
<thead>
<tr>
<th>Number of processors and system utilization</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>20%</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>86</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>80</td>
<td>83</td>
<td>75</td>
<td>78</td>
<td>80</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>40%</td>
<td>100</td>
<td>100</td>
<td>75</td>
<td>80</td>
<td>83</td>
<td>75</td>
<td>78</td>
<td>73</td>
<td>75</td>
<td>77</td>
</tr>
<tr>
<td>50%</td>
<td>100</td>
<td>100</td>
<td>75</td>
<td>80</td>
<td>71</td>
<td>67</td>
<td>64</td>
<td>67</td>
<td>64</td>
<td>71</td>
</tr>
<tr>
<td>60%</td>
<td>100</td>
<td>67</td>
<td>75</td>
<td>67</td>
<td>71</td>
<td>67</td>
<td>64</td>
<td>67</td>
<td>64</td>
<td>67</td>
</tr>
<tr>
<td>70%</td>
<td>100</td>
<td>67</td>
<td>60</td>
<td>67</td>
<td>63</td>
<td>60</td>
<td>64</td>
<td>62</td>
<td>60</td>
<td>63</td>
</tr>
<tr>
<td>80%</td>
<td>100</td>
<td>67</td>
<td>60</td>
<td>57</td>
<td>63</td>
<td>60</td>
<td>58</td>
<td>57</td>
<td>56</td>
<td>59</td>
</tr>
<tr>
<td>90%</td>
<td>100</td>
<td>67</td>
<td>60</td>
<td>57</td>
<td>56</td>
<td>55</td>
<td>54</td>
<td>53</td>
<td>53</td>
<td>56</td>
</tr>
<tr>
<td>100%</td>
<td>100</td>
<td>67</td>
<td>60</td>
<td>57</td>
<td>56</td>
<td>55</td>
<td>54</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 3: Measured percent of deadlines met on the multiprocessor.

<table>
<thead>
<tr>
<th>approx. system load</th>
<th>number of processors</th>
<th>average % deadlines met</th>
</tr>
</thead>
<tbody>
<tr>
<td>30%</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>40%</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>70%</td>
<td>3</td>
<td>99.9</td>
</tr>
<tr>
<td>100%</td>
<td>5</td>
<td>99.9</td>
</tr>
<tr>
<td>100%</td>
<td>3</td>
<td>90.2</td>
</tr>
<tr>
<td>100%</td>
<td>5</td>
<td>90.3</td>
</tr>
</tbody>
</table>
Table 4: Completion time on a 4 processor multiprocessor with processes ranging in time from 200-400 units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>best without migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>357</td>
<td>357</td>
<td>357</td>
<td>357</td>
</tr>
<tr>
<td>5</td>
<td>381</td>
<td>473</td>
<td>413</td>
<td>549</td>
</tr>
<tr>
<td>6</td>
<td>442</td>
<td>521</td>
<td>466</td>
<td>589</td>
</tr>
<tr>
<td>7</td>
<td>523</td>
<td>574</td>
<td>540</td>
<td>644</td>
</tr>
<tr>
<td>8</td>
<td>596</td>
<td>634</td>
<td>613</td>
<td>682</td>
</tr>
<tr>
<td>9</td>
<td>673</td>
<td>728</td>
<td>689</td>
<td>813</td>
</tr>
<tr>
<td>10</td>
<td>747</td>
<td>800</td>
<td>760</td>
<td>885</td>
</tr>
<tr>
<td>11</td>
<td>823</td>
<td>878</td>
<td>835</td>
<td>937</td>
</tr>
<tr>
<td>12</td>
<td>897</td>
<td>928</td>
<td>907</td>
<td>992</td>
</tr>
<tr>
<td>13</td>
<td>969</td>
<td>1011</td>
<td>975</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>1037</td>
<td>1083</td>
<td>1048</td>
<td>1169</td>
</tr>
<tr>
<td>15</td>
<td>1116</td>
<td>1161</td>
<td>1125</td>
<td>1248</td>
</tr>
<tr>
<td>16</td>
<td>1198</td>
<td>1229</td>
<td>1207</td>
<td>1318</td>
</tr>
</tbody>
</table>

Table 5: Completion time on the multiprocessor with 100 processors and processes ranging in size from 200-400 time units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>best without migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>393</td>
<td>403</td>
<td>399</td>
<td>504</td>
</tr>
<tr>
<td>200</td>
<td>598</td>
<td>629</td>
<td>620</td>
<td>776</td>
</tr>
<tr>
<td>400</td>
<td>1199</td>
<td>1216</td>
<td>1208</td>
<td>1414</td>
</tr>
<tr>
<td>600</td>
<td>1797</td>
<td>1812</td>
<td>1804</td>
<td>1978</td>
</tr>
<tr>
<td>800</td>
<td>2398</td>
<td>2409</td>
<td>2405</td>
<td>2520</td>
</tr>
<tr>
<td>1000</td>
<td>2997</td>
<td>3010</td>
<td>3005</td>
<td>3115</td>
</tr>
</tbody>
</table>
Table 6: Completion time on the multiprocessor with 50 processors and processes ranging in size from 200-400 time units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>best without migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>393</td>
<td>411</td>
<td>393</td>
<td>506</td>
</tr>
<tr>
<td>100</td>
<td>601</td>
<td>634</td>
<td>618</td>
<td>773</td>
</tr>
<tr>
<td>150</td>
<td>904</td>
<td>928</td>
<td>919</td>
<td>1087</td>
</tr>
<tr>
<td>200</td>
<td>1194</td>
<td>1215</td>
<td>1206</td>
<td>1387</td>
</tr>
<tr>
<td>500</td>
<td>2992</td>
<td>3005</td>
<td>3000</td>
<td>3211</td>
</tr>
</tbody>
</table>

Table 7: Completion time on the multiprocessor with 25 processors and processes ranging in size from 200-400 time units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>best without migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>392</td>
<td>423</td>
<td>392</td>
<td>503</td>
</tr>
<tr>
<td>50</td>
<td>595</td>
<td>641</td>
<td>623</td>
<td>739</td>
</tr>
<tr>
<td>75</td>
<td>897</td>
<td>922</td>
<td>911</td>
<td>1074</td>
</tr>
<tr>
<td>100</td>
<td>1207</td>
<td>1225</td>
<td>1218</td>
<td>1394</td>
</tr>
<tr>
<td>250</td>
<td>2979</td>
<td>2992</td>
<td>2987</td>
<td>3177</td>
</tr>
</tbody>
</table>

Table 8: Completion time on the multiprocessor with 10 processors and processes ranging in size from 200-400 time units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>best without migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>385</td>
<td>453</td>
<td>393</td>
<td>500</td>
</tr>
<tr>
<td>21</td>
<td>629</td>
<td>692</td>
<td>654</td>
<td>795</td>
</tr>
<tr>
<td>31</td>
<td>936</td>
<td>971</td>
<td>953</td>
<td>1076</td>
</tr>
<tr>
<td>41</td>
<td>1220</td>
<td>1243</td>
<td>1231</td>
<td>1399</td>
</tr>
<tr>
<td>51</td>
<td>1518</td>
<td>1540</td>
<td>1531</td>
<td>1680</td>
</tr>
<tr>
<td>76</td>
<td>2278</td>
<td>2298</td>
<td>2286</td>
<td>2429</td>
</tr>
<tr>
<td>101</td>
<td>3022</td>
<td>3037</td>
<td>3031</td>
<td>3181</td>
</tr>
</tbody>
</table>
Table 9: Completion time on the multiprocessor with 10 processors and processes ranging in size from 10-100 time units.

<table>
<thead>
<tr>
<th>number of processes</th>
<th>best with migration</th>
<th>actual with migration</th>
<th>actual without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>92</td>
<td>92</td>
<td>96</td>
</tr>
<tr>
<td>21</td>
<td>114</td>
<td>120</td>
<td>159</td>
</tr>
<tr>
<td>31</td>
<td>170</td>
<td>175</td>
<td>215</td>
</tr>
<tr>
<td>41</td>
<td>226</td>
<td>231</td>
<td>273</td>
</tr>
<tr>
<td>51</td>
<td>281</td>
<td>285</td>
<td>325</td>
</tr>
<tr>
<td>101</td>
<td>556</td>
<td>562</td>
<td>609</td>
</tr>
<tr>
<td>201</td>
<td>1100</td>
<td>1110</td>
<td>1160</td>
</tr>
</tbody>
</table>

Table 10: Measured percent of deadlines on the cube for various loads and process types.

<table>
<thead>
<tr>
<th>approximate system load</th>
<th>Average percent deadlines met</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single cube size</td>
</tr>
<tr>
<td>50%</td>
<td>100</td>
</tr>
<tr>
<td>60%</td>
<td>100</td>
</tr>
<tr>
<td>70%</td>
<td>99.9</td>
</tr>
<tr>
<td>80%</td>
<td>99.5</td>
</tr>
<tr>
<td>90%</td>
<td>98.2</td>
</tr>
<tr>
<td>100%</td>
<td>90.7</td>
</tr>
</tbody>
</table>
Table 11: Completion time on the 32 node cube with random subcube sizes and task lengths between 5-20 time units.

<table>
<thead>
<tr>
<th>number processes</th>
<th>best</th>
<th>with migration</th>
<th>without migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>102</td>
<td>130</td>
<td>138</td>
</tr>
<tr>
<td>20</td>
<td>162</td>
<td>203</td>
<td>213</td>
</tr>
<tr>
<td>30</td>
<td>209</td>
<td>241</td>
<td>244</td>
</tr>
<tr>
<td>40</td>
<td>295</td>
<td>348</td>
<td>353</td>
</tr>
<tr>
<td>50</td>
<td>395</td>
<td>439</td>
<td>452</td>
</tr>
<tr>
<td>100</td>
<td>1093</td>
<td>1130</td>
<td>1141</td>
</tr>
<tr>
<td>150</td>
<td>1620</td>
<td>1658</td>
<td>1664</td>
</tr>
<tr>
<td>200</td>
<td>1887</td>
<td>1931</td>
<td>1938</td>
</tr>
</tbody>
</table>
Bibliography


