M.Sc. Thesis, Department of Computer Science, University of Toronto.

Processor Scheduling in Multiprogrammed Shared Memory NUMA Multiprocessors

by

Chee-Shong Wu

A thesis submitted in conformity with the requirements
for the Degree of Master of Science
Graduate Department of Computer Science in the
University of Toronto

Acknowlegement

I am grateful to my supervisor, Dr. Ken Sevcik. Without his support and guidance, this thesis could not have been finished. I would like to thank Dr. Songnian Zhou for being my second reader. His valuable suggestions have helped to improve this thesis to a great extend.

I would also like to thank Tim Brecht for helping me implement the scheduler. His experience with Hector and scheduling research has assisted me in completing this thesis.

My appreciation to everyone who makes the Hector/Hurricane project possible.

Many thanks to Ronnie and Wai Kau for proofreading this thesis, and Kathy for keeping me on my toes.

Cong Cong, thank you for your love and understanding, for being my lucky star.

Special thanks to the Mahjoub family, for their continuous encouragement. You always make me feel at home.

I dedicate this thesis to my loving parents, my caring sister and brother. Thank you all for believing in me.

Processor Scheduling in Multiprogrammed Shared Memory NUMA Multiprocessors

Chee-Shong Wu
Master of Science
Department of Computer Science
University of Toronto 1993

Abstract

In a multiprogrammed multiprocessor, the scheduler is not only responsible for deciding when to activate an application and when to suspend it, but is also responsible for determining how many processors to allocate to each application. In a scalable NUMA multiprocessor, it must further resolve the problem of which processors to allocate to which application since the memory reference times are not the same for all processor-memory pairs.

In this thesis, we study the problem of how to characterize parallel applications and how to apply this knowledge in scheduling for NUMA systems. We also study the performance of several scheduling algorithms in a NUMA environment. These algorithms differ in their frequency of reallocations. We propose two policies, the Static policy and the Immediate Start Static policy, that utilize application characteristics when making scheduling decisions. The performance of these two policies is compared with that of the Dynamic policy, on a NUMA multiprocessor, Hector.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this research, we study the performance issues of processor scheduling in multiprogrammed NUMA multiprocessor systems.

We examine two different models that approximate the execution time function of a given application and study their effectiveness. Scheduling policies that make use of the more effective one of the two models are derived. These policies are relatively static in terms of processor allocations, and their performance is compared with a dynamic policy on a NUMA multiprocessor.

1.1 Multiprocessors

Multiprocessor systems have received an increasing amount of attention during the past decade. They are built by integrating many relatively inexpensive, readily available components. Multiprocessor systems have the potential of satisfying the computing needs of new applications that require more computation and space.

However, this growth in computing power is accompanied by an increase in the complexity of the system software. System software which used to handle a single processor is now responsible for managing a number of processors. If the system software is not designed to manage the multiprocessor system efficiently, this increased complexity will degrade the performance of applications. Then, the overhead of running applications on multiprocessors will outweigh the performance gain; the potential benefits of multiprocessor systems will be lost.

Multiprocessor systems can be divided into two classes: shared memory multiprocessors and non-shared memory multiprocessors. In a shared memory multiprocessor, all processors have access to a single uniform virtual address space. This property holds regardless of how individual memory units are linked to form the address space. As a result, shared memory multiprocessors provide application programmers with a simple programming model, allowing easy communication and synchronization among the threads of an application. This is in contrast to non-shared memory multiprocessors where communication among threads is accomplished through explicit message passing.

Alliant, DEC, Encore, Sequent and SGI are some developers of small-scale shared memory multiprocessors. These machines typically consist of a number of processors, with local caches, connected to the global memory via a shared bus. Although the shared bus structure is simple and inexpensive to implement, it has some serious drawbacks. The performance of multiprocessors using this approach is limited by the bus capacity. It is impossible for a large-scale shared memory multiprocessor to perform well using the shared bus structure since the bus quickly becomes a bottleneck as more processors are added to the system. Although the bus can be replaced by a large switch, the cost of the switch grows rapidly with the system size, and such a Uniform Memory Access (UMA) architecture is likely to make all the memory accesses uniformly slow [ZB91].

In a shared memory Non-Uniform Memory Access (NUMA) multiprocessor, the physical memory of the system is distributed among individual processors but is still globally addressable by all processors; thus, the cost of a memory access is *dependent* on where the memory unit addressed is located relative to the processor; some accesses may be local, some may be remote. By using NUMA architectures, the scalability of multiprocessors is ensured since, as the number of processors in the system increases, only remote memory access costs will be affected.

1.2 Multiprogramming

The idea of multiprogramming was originally introduced to improve the performance of uniprocessors through an increase in processor utilization by avoiding idling of the processor. The term multiprogramming is defined, in an uniprocessor system, as the ability to execute more than one application concurrently. This can be achieved by time-slicing the processor among multiple applications in the system, giving the illusion that each application possesses its own processor (of less power than the actual processor).

Our definition of multiprogramming in multiprocessor systems is a straightforward extension of the uniprocessor definition: the ability to execute more than one application, some possibly parallel, in a system simultaneously. This can be accomplished through time-sharing (or time-slicing) the processors or space-sharing the processors among applications. Both of these terms will be explained in the next chapter.

1.3 Scheduling in Multiprogrammed NUMA Multiprocessors

In uniprocessor systems, the task of the scheduler is to decide when to activate an application and when to suspend it. However, in multiprocessor systems, additional responsibilities are assumed by the scheduler. In particular, when a new application arrives, the scheduler must also determine how many processors to allocate to the application. This decision may require revision later since an application's processor requirement may change during its life time, and the marginal utility of an additional processor varies from application to application.

With the introduction of NUMA systems, the scheduler now must also determine which processor(s) to allocate to which application. The relative positions of processors allocated to an application will affect the performance of it due to the difference in memory access times. Performance of an application may improve if all its allocated processors use only local memories. To achieve this requires coordination between the scheduler and the memory manager. Intuitively, it seems beneficial to assign the application a set of processors that are close together (i.e., memory access costs are relatively low for all processor-memory pairs assigned to the application).

Multiprogramming further complicates the task of a scheduler. Decisions must be made on whether to assign the same number and the same set of processors to an application at each reallocation point. The potential benefit of assigning the same set of processors to an application at each reallocation point is that some data that are needed by the application may remain in the associated processor caches and local memories; thus, less cache and memory reloading is required. Furthermore, fewer data reallocations are needed to maintain a high percentage of local accesses. Since different applications possess different parallelism characteristics and structures, if we could obtain this information and provide it to the scheduler, it would help the scheduler to make better allocation decisions.

Scheduling is an integral part of any computer system. In order for a multiprocessor to perform up to its potential, an effective scheduler is essential. An effective scheduler must not ignore the fact that there exist other system components, such as the memory manager, which also try to improve the overall system performance. The decisions made by one component may enhance or negate the performance improvement created by another. Cooperation among all system components may prove to be crucial to the success of any computer system.

Chapter 2

Related Work

We will present some background in multiprocessor scheduling in this chapter. However, we will first establish some terminology.

The term *process* is used to refer to both heavyweight processes (processes consisting of a single address space and a single thread of control), and lightweight processes (processes of a program concurrently and cooperatively executing within the same address space); both of which are kernel level processes that are scheduled by the *kernel-level processor scheduler*. The term *thread*, on the other hand, refers to user-level threads that are implemented and scheduled by the *thread dispatcher* of runtime thread packages which are linked with each application.

Before the introduction of two-level scheduling, parallel applications were assumed to be divided into a number of (lightweight) processes executing in parallel. Each of these processes was scheduled by the *kernel-level processor scheduler*. There was no notion of (user-level) threads at this time.

After it was introduced, policies that use two-level scheduling assume that an application is divided into small chunks of work, each of which is executed by a single (user-level) thread. These threads are dispatched onto a number of (lightweight) processes (dedicated to that application) by the *thread dispatcher*; the processes of the applications are further scheduled onto processors by the *kernel-level processor scheduler*.

Among the scheduling policies to be discussed in this chapter, the First-Come-

First-Served, the Smallest-Number-of-Processes-First, the Smallest-Service-Demand-First, the Round-Robin-Jobs and the Round-Robin-Processes policies do not use two-level scheduling; while the Equipartition, the Dynamic, and any static policies that reallocate at arrivals and/or completions, utilize two-level scheduling.

All the policies used in our experiments are two-level schedulers. Thus, in this work, applications are assumed to be divided into a number of user-level threads.

2.1 Policies from Uniprocessor Scheduling

A natural way of designing a scheduling algorithm for multiprocessor systems is to apply knowledge and experience from uniprocessor scheduling in a multiprocessor context. Researchers have extended a few traditional uniprocessor algorithms to multiprocessing systems and have evaluated their effectiveness.

Majumdar, Eager and Bunt [MEB88] and Squillante [Squ90] studied the multiprocessor version of First-Come-First-Served Policy (FCFS). In this version, when a processor becomes idle, the scheduler assigns the process (regardless of which application it belongs to) at the head of a global ready queue to the idle processor; and all the processes of a newly arrived application are placed at the end of the global ready queue. The results in these papers have shown that FCFS does not perform as well as other multiprocessor scheduling policies that are based on application characteristics. FCFS allows monopolization of the system by large applications (applications with a large number of processes and large cumulative demand). Under FCFS, small applications (applications with a small number of processes and small cumulative demand) that can potentially finish in very little time have to wait for large applications to finish before they can start executing.

To avoid the problem of large applications dominating system resources as under FCFS, the scheduler can use the Shortest-Job-First (SJF) policies. These policies give higher priority to small applications, thus allowing them to finish sooner. SJF policies have been shown to be useful in uniprocessor scheduling.

Majumdar, Eager and Bunt [MEB88], and Leutenegger and Vernon [LV90] provided an extensive study of a few multiprocessor scheduling policies based on SJF. They are Smallest-Number-of-Processes-First (SNPF), Smallest-Cumulative-Demand-First (SCDF), and their preemptive counterparts (PSNPF, PSCDF). SCDF and PSCDF perform significantly better than FCFS. But SNPF and PSNPF perform only slightly better than FCFS, unless there is a positive correlation between the number of processes in an application and the cumulative demand of an application.

Processor-Sharing (PS) is another effective scheduling policy in uniprocessor systems, particularly when there exists a high variation of service demand among applications. The processor of an uniprocessor is time-multiplexed among applications in the system, giving only a small quantum of service to an application at a time, then quickly switching to another application. Similar to SJF policies, PS prevents the total monopolization of processing power by large applications since small applications will finish sooner (as they require fewer quanta to complete). A natural extension of PS to multiprocessor systems leads to Round-Robin-Process (RRprocess) [MEB88]. In this policy, when a processor completes its quantum of service on a process, it goes to the global queue, puts the process at the end of the queue, then takes the process at the head of the queue to service for the next quantum. In this case, each *process* in the system receives an approximately equal fraction of the processing power. Majumdar, Eager and Bunt [MEB88] have studied the performance of RRprocess. They concluded that RRprocess performed poorly in comparison to policies based on application characteristics such as SNPF and SCDF, in particular when the variability in application parallelism is high and the variability in application cumulative demand is low. Another policy that extends from PS is Round-Robin-Jobs (RRjob). Leutenegger and Vernon [LV90] concluded that RRjob, which allocates an approximately equal fraction of the processing power to each *job* (or application) in the system (rather than to each *process* in the system as in RRprocess), performs well under almost all workload assumptions.

2.2 Time-Sharing versus Space-Sharing Policies

One way of categorizing different multiprocessor scheduling policies is by the manner in which concurrency is supported. In a time-sharing policy, each processor spends a very short interval (a small quantum) executing any particular process, then quickly rotates to another process. Thus, each application in the system sees alternating periods of time where it holds many processors (possibly all) and then a few (possibly none). With space-sharing policies, processors in the system are partitioned among applications. Each process owns the processor it is on for a relatively long interval (a large quantum), or until it is completed. Thus, each application has a more constant allocation of fewer processors then it does under time-sharing.

Several studies have compared time-sharing policies and space-sharing policies. Tucker and Gupta [TG89] showed that it is beneficial to keep the number of active processes in an application no larger than the number of processors executing it. This avoids the problem of time-sharing the processors within the application's allocation. Time-sharing degrades performance because of the overhead of frequent context switches, and processor cache corruption. Also, there is a danger that a process holding a lock might be preempted.

McCann, Vaswani and Zahorjan [MVZ91] did a performance comparison between time-sharing policies and space-sharing policies. The time-sharing policy they examined is RRjob; the space-sharing policy they examined is Equipartition[1] where the scheduler tries to maintain an equal share of processors to all active applications. They concluded that space-sharing policies dominate time-sharing policies because they make more efficient use of processors. Because time-sharing policies allocate a large number of processors in short intervals, and because most parallel applications have sublinear speedups, processors are more likely to become idle under time-sharing policies. In summary, time-sharing is worse than space-sharing because of higher overhead from context switches and cache reloading.

[1]Tucker and Gupta originally called this policy Process Control [TG89]

2.3 Two-Level Scheduling

With the introduction of space-sharing policies comes the idea of two-level schedulers. Two-level schedulers split the task of scheduling processors into two parts. The kernel is responsible for allocating or partitioning processors among applications; each application (possibly with the help of a runtime thread dispatcher) is responsible for scheduling its threads onto its allocated processors.

By allowing the applications to schedule threads onto processors, the kernel scheduler (or processor allocator) is relieved of the difficulties involved in synchronizing threads and preempting threads that are executing in critical sections.

Since two-level scheduling allows the kernel processor scheduler to change the number of processors allocated to each application dynamically, this provides flexibility to the scheduler. Applications can be serviced immediately when they arrive, and, when they complete, the freed processors can be allocated immediately to other currently active applications.

From the application's point of view, two-level scheduling allows the application to execute on any number of processors. This is advantageous since, if one or more processors fail, the application can still run on whatever number of processors remain. Also, the application is more portable with two-level scheduling. If an application is written for one multiprocessor, it should be able to run on a similar multiprocessor with different number of processors without modifying its code.

2.4 Static versus Dynamic Policies

Another classification of multiprocessor schedulers is based on the frequency of reallocations. At the two extremes are the Static policy (no reallocation) and the Dynamic policy (frequent reallocations). In the Static policy, an application is allocated a fixed number of processors when it is activated, and it keeps these processors throughout its lifetime. In the Dynamic policy, the number of processors allocated to an application may vary during its execution. The scheduler in this case has the responsibility of adjusting the number of processors allocated to an application according to its time varying parallelism, as well as to the system load (new arrivals and completions).

The Static policy is simple to implement and inexpensive to use, but it fails to recognize that most parallel applications have sublinear speedups, thus cannot fully utilize all the processors allocated to them during their entire execution. Because applications hold the same number of processors until they terminate, new applications cannot be started immediately. Both of these factors could have a negative impact on the system performance, especially when the system load is high. The Dynamic policy, however, addresses this problem and adjusts the allocation to each application to maximize processor utilization. Unfortunately, this also introduces extra overhead for the scheduler since Dynamic scheduling is, in general, more complex. Moreover, because the Dynamic policy frequently switches processors from application to application, extra context switches, loss of processor cache affinity, and disruption of data locality can also degrade the system performance.

There exist policies that are between the Static policy and the Dynamic policy. These policies may reallocate processors when a new application arrives, when an active application completes, or at both occasions. The Equipartition (Process Control) policy proposed by Tucker and Gupta [TG89] is one example.

Sevcik [Sev92] identified some additional scheduling policies based on the frequency of reallocations that are between the Static policy and the Dynamic policy. They are (1) policies that reallocate at completion of an application, (2) policies that reallocate at both arrivals and completions, and (3) policies that reallocate at a phase change in parallelism of an application.

McCann, Vaswani and Zahorjan [MVZ91, VZ91, ZM91] extensively study the performance of some static policies and their Dynamic policy. Their conclusion is that, unless the overhead of context switches is quite high, the Dynamic policy will always outperform static policies.

2.5 Application Characteristics in Scheduling

Several studies have identified a number of application characteristics that can be used to improve scheduling. These characteristics can be known in advance, estimated from past runs, or observed during execution. For dynamic scheduling, it is useful to know the parallelism profile [Sev89] of the application, which shows how many processors could be used by the application in various phases of execution. In most static scheduling policies, it is helpful to know the minimum parallelism m, the maximum parallelism M, and the average parallelism A of the application [Sev89]. These numbers inform the scheduler about the range of the number of processors that each application should be allocated.

Majumdar, Eager and Bunt [MEB91] proposed a parameter $\omega(A)$ which measures the variability in an application's instantaneous parallelism. The higher the variability in the instantaneous parallelism of an application, the higher is the value of $\omega(A)$. They showed that when this parameter is used in conjunction with A, tight bounds on the optimal average response time by a static scheduler can be obtained. This will assist the static scheduler in allocating an appropriate number of processors to each application.

Dowdy described an application characterization, called *execution signature*, that has the form of:

$$\mu_j(p) = \frac{p}{C_{j1} + C_{j2}}.$$

The term $\mu_j(p)$ is the execution rate of application j on p processors where C_{j1} and C_{j2} are two constants that characterize application j [Dow88]. Using this parameter, we can derive an approximation of the execution time function of application j. Knowledge of the execution time curve can be used to improve scheduling decisions.

Sevcik proposed a function,

$$T_j(p) = \phi_j(p)\frac{W_j}{p} + \alpha_j + \beta_j p,$$

where $T_j(p)$ is the execution time of an application j on p processors [Sev92]. If estimates for the terms $\phi_j(p)$, W_j, α_j and β_j can be obtained, then the scheduler can use them to evaluate the tradeoff of taking a processor away from one application and giving it to another, thus improving scheduling decisions.

2.6 Affinity Scheduling

In a shared-memory multiprocessor with caches, it may be more advantageous to execute a process on one processor than others. This is because executing processes develop *affinity* to processors by filling up their caches. If we assign a particular process to a processor for which the process has a high affinity, execution time can be significantly reduced since many memory requests are to blocks that are already in the cache.

Affinity scheduling involves making use of this affinity when doing processor allocations. Some policies naturally use affinity when making scheduling decisions. The Static policy is an example. It allows processes to run on their allocated processors to completion, thus letting them retain their cache context. The Equipartition policy is quite successful in exploiting cache affinity by allocating a small number of processors to an application and maintaining the same set of processors through out the application's lifetime. However, the Dynamic policy, where processors are frequently switched among several applications, wastes processing power by causing frequent cache reloads. Time-sharing policies ignore cache affinity by rotating the set of processors from one application to another at each quantum. Caches would require reloading at every quantum.

Squillante and Lazowska [SL93] used a queueing network model and Mean Value Analysis plus simulation to show that exploiting even the simplest forms of cache affinity in scheduling policies can provide significant improvements over ignoring this affinity. In their experimental work, Gupta, Tucker and Urushibara [GTU91] concluded that the effect of affinity scheduling is positive on the performance of applications. However, the degree of this gain depends on the application's footprint size and complex interaction among applications running at the same time. They used the composition of execution time as a performance metric in their experiments, where the composition of execution time consists of the percentage of time an application spent on doing useful work, the percentage of time spent waiting for data fetched, and the percentage of time spent being idle due to synchronization operations and context switches. However, the experiments performed by Vaswani and Zahorjan [VZ91] showed that on current machines, considering processor affinity in their dynamic scheduler has only a limited benefit on performance. This result is confirmed in later work by McCann, Vaswani and Zahorjan [MVZ91].

2.7 Scheduling in NUMA Machines

All the work on scheduling discussed in the previous sections have been based on small-scale UMA multiprocessors. As mentioned in the last chapter, unlike NUMA multiprocessors, UMA machines are not scalable. Because of the difference in scale and structure, scheduling algorithms that are considered effective in UMA systems may not work as well in NUMA systems. To date, little research has been done on scheduling or processor allocation in large-scale NUMA multiprocessors.

Zhou and Brecht [ZB91] proposed a pool-based scheduling policy for large-scale NUMA multiprocessors in which the processors of the system are partitioned into processor pools. Processor-memory pairs within a pool are typically "close" together, and can be associated with clusters of processors in the system to reflect the architecture. In general, the processors allocated to an application are within a single pool, unless there are performance benefits for an application to span multiple pools. By doing this, the locality of data is taken into account since memory accesses within a pool is less costly than memory accesses in other pools.

Srikantiah [Sri91] used simulation to compare several scheduling disciplines in multiprogrammed NUMA multiprocessors. She concluded that it is beneficial for scheduling purposes to assign processes of an application to "nearby" processors. Allocating a set of processors that are "nearby" reduces memory access overhead. This is because from any particular processor in a set of "nearby" processors, it is

Factors	Static	Dynamic
data locality	+ can be fully exploited by scheduler	− may be sacrificed since processors are switched among applications
affinity	+ can be fully exploited by scheduler	− may be sacrificed since processors are switched among applications
context switch overhead	+ minimal (may be at arrivals/completions)	− may be high
communication overhead	+ communicate with applications only during startup	− requires constant communication with applications
processor utilization	− processors are not fully utilized due to variable parallelism	+ processors may be highly utilized
new arrivals	− may have to wait for idle processors	+ may start right away if there are more processors than applications

Table 2.1: Comparison of performance factors of static and dynamic policies

less costly to access processor-memory pairs within the set than processor-memory pairs outside the set (since they are more remote). Also, when the system load is very high, it is optimal to assign a single processor to each application.

2.8 The Goals and Motivation

The work in this thesis is motivated by the need to understand the performance of different scheduling disciplines in NUMA multiprocessors. Although it has been shown that dynamic policies perform better than static policies in UMA systems, the same conclusion cannot be drawn with confidence in NUMA systems. Table 2.1 compares the performance factors of static and dynamic policies. Note that these factors do not affect the performance of the system and the applications to the same degree, and some of these factors may dominate others depending on the types of system. Data locality, which is not a factor in traditional UMA systems, is more favorable to static policies. The processor cache affinity factor, which is more significant in NUMA systems than UMA systems (since it is more costly to reload caches), also favors static policies. We believe that by utilizing application characteristics, it is possible for a static policy to outperform a dynamic policy in NUMA systems.

The goals of this thesis are the following. We will study the effectiveness of Sevcik's model [Sev92] and Dowdy's execution signature model [Dow88] in approximating the execution time function using a set of parallel applications. Through experiments on an existing NUMA multiprocessor, Hector, we will compare the performance of a set of policies based on the frequency of reallocations that range from static policies to dynamic policies, applying workloads created from the parallel applications. These policies, which make use of application characteristics in making scheduling decisions, conform to the set of policies considered by Sevcik [Sev92].

2.9 Thesis Organization

The following chapters are organized as follows. We will give a brief description of the system on which our experiments were performed in Chapter 3. In Chapter 4, we will discuss the applications that were chosen for our study, their characteristics and their approximated execution time curves that were obtained using Sevcik's and Dowdy's models. Chapter 5 consists of a description of the three policies to be studied and the details of their implementations. The experimental results will be presented in Chapter 6, followed by our concluding remarks in Chapter 7.

Chapter 3

System Description

In this chapter, we will describe the system on which all our experiments were performed. First, some basic characteristics of NUMA machines are discussed. Then, a description of the actual system is presented.

3.1 NUMA Machine Properties

A Non-Uniform Memory Access (NUMA) shared memory multiprocessor consists of a set of memory units connected by hardware to form a globally shared address space. The memory units are connected in such a way that the cost of a memory access depends on the distance between the processor and the memory module involved, thus creating the Non-Uniform Memory Access pattern. A common topology for NUMA multiprocessors is a hierarchical structure in which each memory unit is coupled with a processor, and the cost of a memory access depends on which level of the hierarchy must be reached before it can be completed. As mentioned in Section 1.1, NUMA multiprocessors are an important class of multiprocessors since they possess the scalability that is absent in Uniform Memory Access (UMA) multiprocessors (in which all memory accesses have the same cost) [Unr93]. Scalability allows the architecture and the operating system structure of a small scale multiprocessor to be easily extendable to a large scale multiprocessor. Large scale multiprocessors offer much greater potential in their capacity for supporting parallel applications by realizing the performance potential of applications with high degrees of parallelism, and by allowing multiple parallel (or sequential) applications to be efficiently executed on such systems concurrently.

In traditional UMA multiprocessors, two factors have a definite impact on the performance of parallel applications: load balancing and processor-cache affinity. Load balancing affects the performance since, if an application cannot divide its computation among the processors evenly, some processor would take longer than others to finish its share of the computation. The overall performance is less than optimal in this case. In shared memory multiprocessors with caches, processes develop "affinity" to processors by filling their caches with data and instructions during execution. Hence, it may be more efficient to assign a returning process a processor with which it has cache affinity [VZ91].

With the introduction of NUMA, data locality becomes an important factor because memory accesses have different costs depending on which memory unit the request addresses. The scheduling module in the operating system must cooperate with the memory management module in such a way that an application ideally only requires local memory accesses to complete its execution. The effects of processor-cache affinity also become more significant in NUMA machines. It is more costly to refill caches in NUMA machines since some of the data requested may reside in remote memory units.

In this work, we limit our attention to NUMA shared memory multiprocessors that are homogeneous and symmetrical . A homogeneous NUMA system is one where all processors in the system are equal in speed and processing power. A symmetrical NUMA system is one where the costs of accessing various levels of memory are the same from every processor's point of view.

3.2 Hector

The experiments in this research are performed on **Hector**. Hector is a NUMA shared memory multiprocessor with a hierarchical structure built at the University of

Toronto. It is a homogeneous, symmetrical NUMA system according to the definition in the last section.

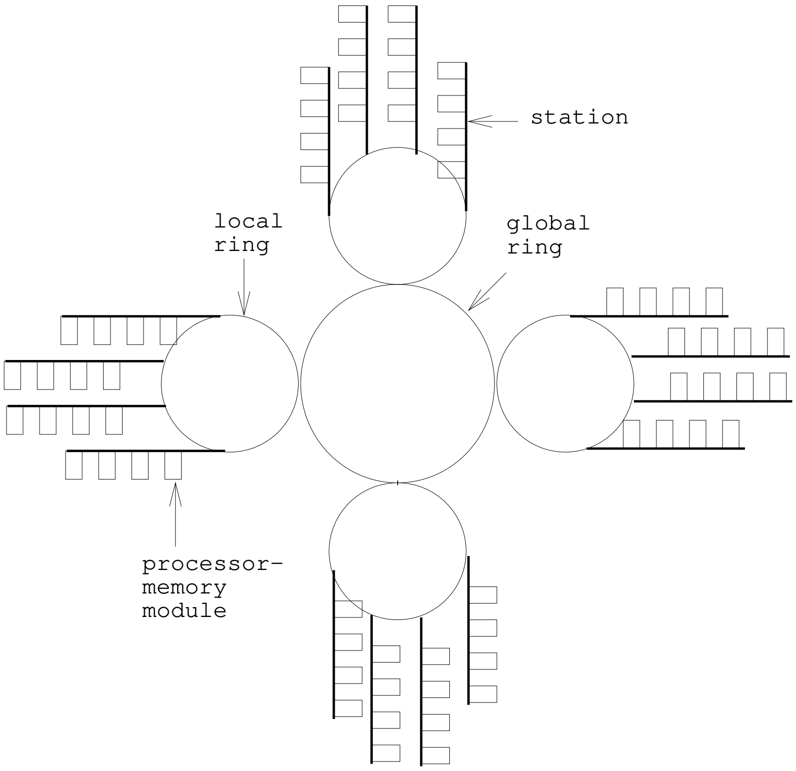

Figure 3.1: Hector with 1 global ring, 4 local rings, 16 stations and 64 processors

Hector consists of a set of stations connected by a hierarchy of rings. Each station consists of a set of processor-memory modules (PMs) connected to a bus. These stations are then interconnected by local rings, and local rings can be further joined by global rings (see Fig. 3.1) [SVW+92].

In the current prototype of Hector, a station consists of four PMs. A local ring is used to connect four of these stations, making up a total of 16 PMs. We can say that Hector possesses three-level NUMAness, i.e., there are three different costs for memory accesses. Local accesses are least expensive; on-station accesses are in the middle; and across-station (or off-station) accesses are most costly.

Each PM contains a 16.67MHz Motorola 88100 microprocessor, a 16Kbyte instruction cache, a 16Kbyte data cache, and 4MByte of on-board local memories. The Page size is 4Kbyte. Putting these on-board local memories together creates a contiguous global physical address space for all the processors in the system. Thus, any

Level	Read	Write
Local Memory	10	10
On Station	15	8
On Local Ring	19	14

Table 3.1: Memory access times at different level on Hector (in machine cycles)

particular PM has access to any memory location in any of the memory modules on other PMs. However, due to the hierarchical structure, the costs of accessing memory modules at different levels of the hierarchy are different. The cost increases as we go up the hierarchy. Table 3.1 summarizes the memory access cost at each level [SVW+92].

Memory access costs increase as we move from local memory accesses to more remote memory accesses. In terms of read accesses, off-station local ring accesses are most costly at 19 cycles, while local accesses are least costly at 10 cycles. Note that on-station writes are faster than local writes since on-station writes do not wait for memory accesses to complete while local writes do.

Chapter 4

The Applications And Their Characteristics

As mentioned in Chapter 2, application characteristics can be used by the scheduler to improve the overall performance of the system and applications. There are many types of application characteristics that could be used. We choose to use the execution time function, $T_j(p)$, of an application, j, given the number of processors, p. This function, $T_j(p)$, informs the scheduler about how efficiently application j can use the p processors allocated to it. In a multiprogramming context, with the execution time function of each application known, the scheduler can then use $T_j(p)$ to examine the tradeoff in overall system performance at reallocation points between

- taking a processor away from an application j and giving it to another application k, or
- maintaining the same allocation for each application.

However, it would be impractical to provide the scheduler the execution time of each application running on every possible number of processors in the system. Simple mathematical models that approximate and give good abstraction of an application's execution time function would be useful in this regard. Both Sevcik [Sev92] and Dowdy [Dow88] have proposed functions that approximate the execution time function $T_j(p)$ of an application j.

In this chapter, we will study the accuracy of these two models using three different applications. We will first present these three applications and discuss their parallelism structures.

4.1 The Applications

The applications we have chosen are all applications that were previously written for shared memory multiprocessors. Each of these applications has a different parallelism structure and is representative of a class of applications with similar structure. In order to provide realistic workloads for experimentation, we have chosen our applications so that not all of them provide very good speedup.

The first application that was chosen is Matrix Multiply, MM, a parallel implementation of the matrix multiply algorithm. It is a highly parallel application that performs a basic fork-join as indicated in Fig. 4.1.

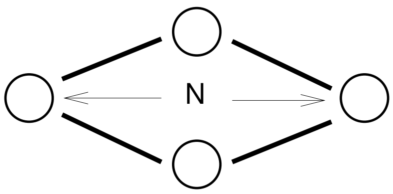

Figure 4.1: MM parallelism structure

The second application, MVA, is a parallel version of the Mean Value Analysis solution for queueing networks with two classes, each of which has N customers. Tasks for this application have precedence. In each iteration, task(i,j) cannot start until both task$(i-1,j)$ and task$(i,j-1)$ are completed with the exception of task$(1,1)$, which has no predecessor. As can been seen in its task precedence graph in Fig. 4.2, the potential parallelism of MVA slowly grows from 1 to $N+1$, then slowly reduces back to 1.

The last application, GRAV, is the Barnes and Hut clustering algorithm [BH86] for simulating the gravitational interaction of N gravitational objects (stars or particles) over time. The algorithm of GRAV is iterative, and at each iteration (or time step) , there are fifth distinct phases of execution that can be done in parallel. At the

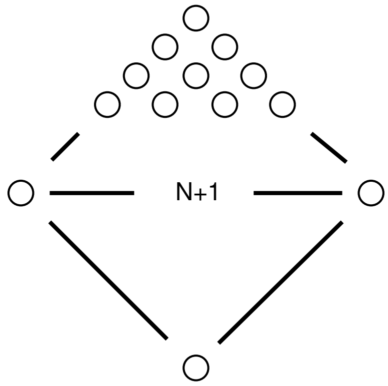

Figure 4.2: MVA parallelism structure

end of each phase, all the threads from that phase are barrier-joined. Thus, GRAV experiences a variable fork-join parallelism structure (see Fig. 4.3). In our implementation, the user can specify the parallelism in each phase during runtime. We used 20 threads for the first, fourth and five phases, 10 threads in the second phase and 1 thread in the third phase for our experiments. This particular parallelism structure corresponds to the minimal execution time on an 8-processor system in a uniprogramming mode.

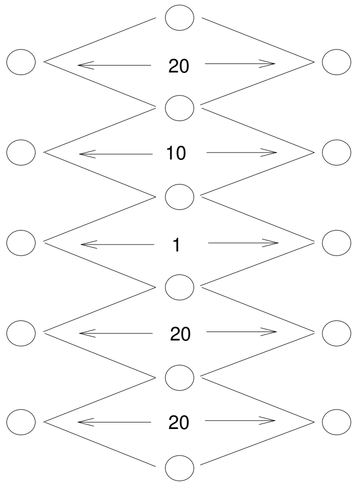

Figure 4.3: GRAV parallelism structure

For each of our applications, we have chosen two problem sizes, a large one (with high service demand) and a small one (with low service demand), providing us with two instances of the application. Choosing real applications of different sizes provides us with workloads of different service demands. This is helpful in giving us a realistic model for our experiments.

The following is a description of each of the six applications:

MM_l: two 400x400 element matrices

MM_s: two 200x200 element matrices

MVA_l: 200 service centers, 40 customers per class and 50 iterations

MVA_s: 200 service centers, 10 customers per class and 50 iterations

$GRAV_l$: 200 stars and 20 iterations

$GRAV_s$: 100 stars and 10 iterations

In the following two sections, the term $T_j(p)$, is used to represent the measured execution time function of application j, while the term $\mathcal{T}_j(p)$ represents the estimated execution time function of application j using one of the two models.

4.2 Sevcik's Model of Execution Time Function

The function,

$$\mathcal{T}_j(p) = \phi_j(p)\frac{W_j}{p} + \alpha_j + \beta_j p,$$

described by Sevcik [Sev92] approximates an application j's execution time function.

In this function, $\frac{W_j}{p}$ represents the ideal division of application j's basic work across p processors; $\phi_j(p)$ is the ratio of the maximum work assigned to any of the p processors to the average work per processor; α_j represents the amount of work per processor increases due to parallel processing; and $\beta_j p$ includes the communication and congestion delays among processors that grow with the number of processors. For values of p that are greater than 1, we have decided to ignore the dependence of ϕ_j on p. So $\phi_j(p) = \phi_j$ for $p > 1$ and $\phi_j(1) = 1$.

If we can obtain approximations for these four parameters, ϕ_j, α_j, β_j and W_j, then they can be provided to the scheduler at runtime, and can be used to approximate the execution time function of application j. For our experimentation purposes, the approximations for these parameters were obtained quite easily. We ran each application j, using 1 to 16 processors, and measured its actual execution times obtaining 16 data points. Then, we took the data points at 2, 4, 8, 12 and 16 processors[1] and applied a least squares approximation on the data points using the function,

$$F_j(p) = \frac{Z_j}{p} + A_j + B_j * p.$$

Note that the data point using one processor is not included because we assume ϕ_j is constant for p greater than 1, and when $p = 1$, $\phi_j = 1$.

In this function, for $p > 1$ processors, B_j approximates β_j, A_j approximates α_j, while Z_j approximates the product of ϕ_j and W_j, and $F_j(p)$ approximates $\mathcal{T}_j(p)$. However, the above procedure has given us approximations for only α_j and β_j. Our next logical step is to try to break the approximation Z_j of $\phi_j * W_j$ into two factors C_j and D_j so that C_j approximates ϕ_j and D_j approximates W_j.

We can achieve this by observing that by definition, when $p = 1$, $\phi_j = 1$ (as a result $C_j = 1$), so

$$\mathcal{T}_j(1) \approx D_j + A_j + B_j.$$

However, if we use $F_j(1)$ to approximate $\mathcal{T}_j(1)$, we get,

$$F_j(1) = Z_j + A_j + B_j = C_j * D_j + A_j + B_j.$$

Note that $F_j(1) > \mathcal{T}_j(1)$ since $D_j > 0$ (the approximate service demand) and $C_j \geq 1$ (from definition). The difference between these two values is due to $F_j(1)$ ignoring the fact that when there is only one processor, naturally work is divided evenly (i.e., $\phi_j = 1$). Taking the data point that was measured at one processor $T_j(1)$ (rather

[1]The least squares approximation using all data points from 2 to 16 processors produces similar results as using only 2, 4, 8, 12 and 16 processor data points, thus the latter is used. Since Hector uses four processors for a station, by including the 4, 8, 12 and 16 processor data points, we can reflect the machine architecture of Hector by taking into account “steps” in execution time curves (due to across station memory accesses).

Application j	ϕ_j	α_j in ms	β_j in ms	W_j in ms	$T_j(1)$ in ms
MM_l	1.01	3394	125	176937	180456
MM_s	1.01	1132	43	22299	23474
MVA_l	1.10	249	1772	59034	61055
MVA_s	1.10	3218	692	4325	9375
GRAV_l	1.16	9657	262	50857	60776
GRAV_s	1.14	3108	62	7666	10836

Table 4.1: Parameters of Sevcik's approximated execution time functions

than using $\mathcal{T}_j(1)$ since D_j is still unknown), we can use the formula,

$$\frac{F_j(1) - A_j - B_j}{T_j(1) - A_j - B_j} = \frac{C_j * D_j}{D_j} = C_j$$

to obtain C_j, an approximation of ϕ_j; and D_j can be obtained easily by using,

$$\frac{Z_j}{C_j} = D_j.$$

Table 4.1 gives the set of parameters obtained for each application's approximated execution time function by applying Sevick's model and using the parameter fitting method described above. The average execution time of each application using one processor is also included for comparison. Note that the aggregate overhead of parallel processing (i.e., the difference between $T_j(p)$ and W_j) for the small applications is greater than the aggregate overhead of their large counterparts. This is because for small applications, the overhead of creating a number of processes (capatured by the term α_j) is relatively large compared to their basic work.

The averaged measured execution time curves together with the estimated execution time curves from Sevcik's model for the six chosen applications are presented in Figure 4.4 to Figure 4.9. Notice from the figures that the applications we have chosen have very distinct execution time curves. The two instances of Matrix Multiply MM_s, MM_l and the large version of Gravity GRAV_l are highly parallel; their execution time continuously decreases as the number of processors increases. The small instance of Gravity GRAV_s's execution time curve begins to curve up as the

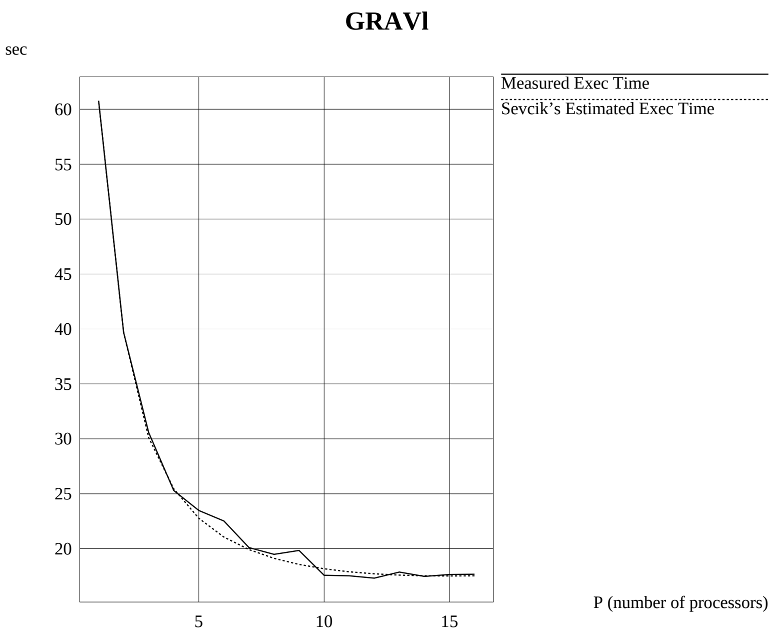

Figure 4.4: Averaged execution time versus Sevcik's estimate for GRAV_l

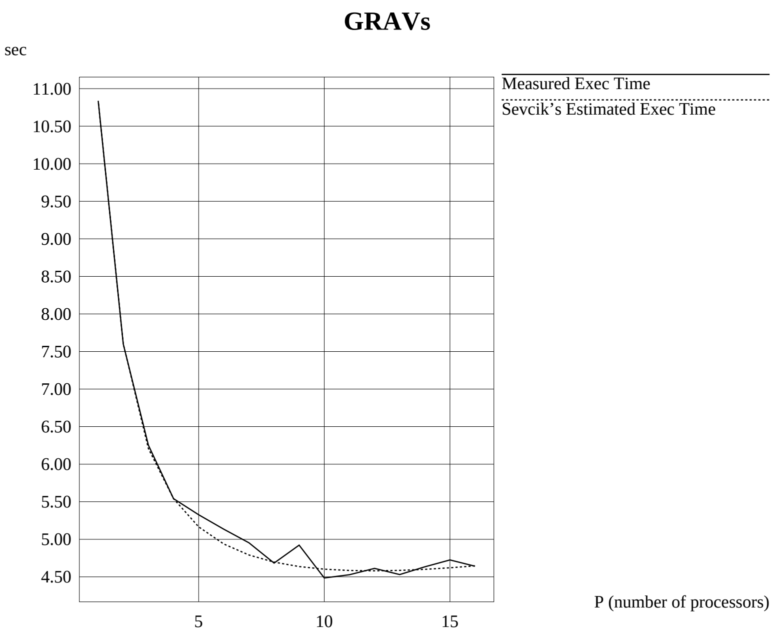

Figure 4.5: Averaged execution time versus Sevcik's estimate for GRAV_s

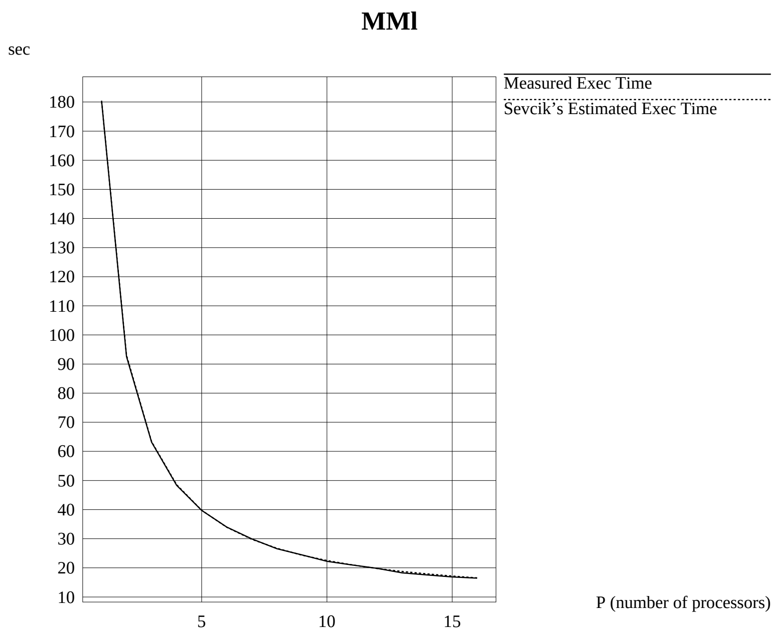

Figure 4.6: Averaged execution time versus Sevcik's estimate for MM_l

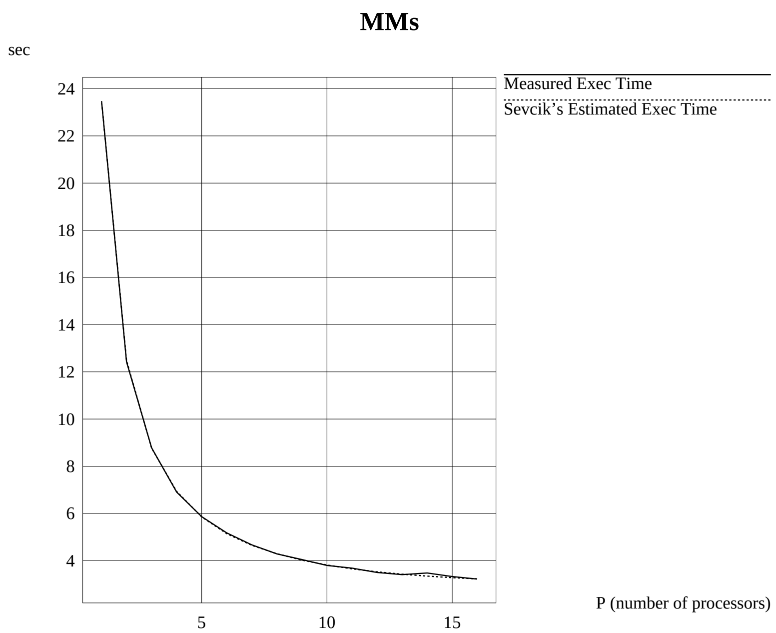

Figure 4.7: Averaged execution time versus Sevcik's estimate for MM_s

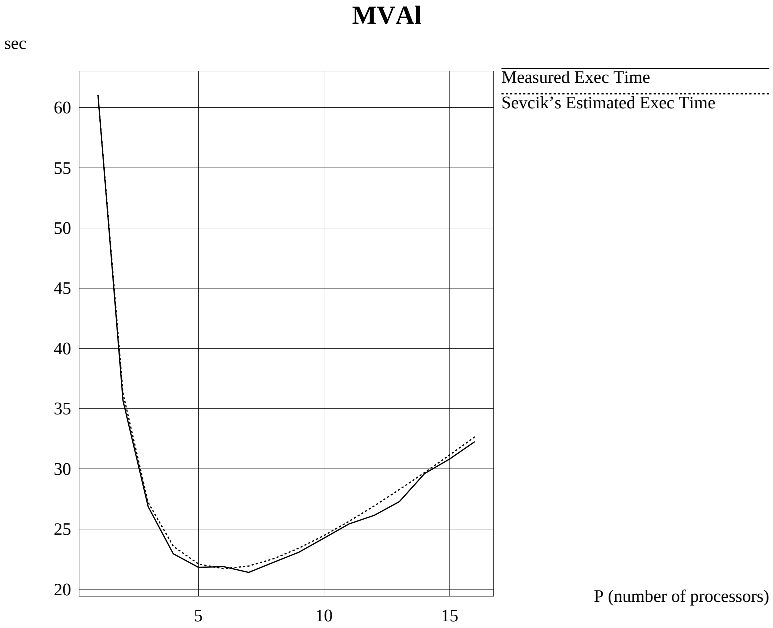

Figure 4.8: Averaged execution time versus Sevcik's estimate for MVA_l

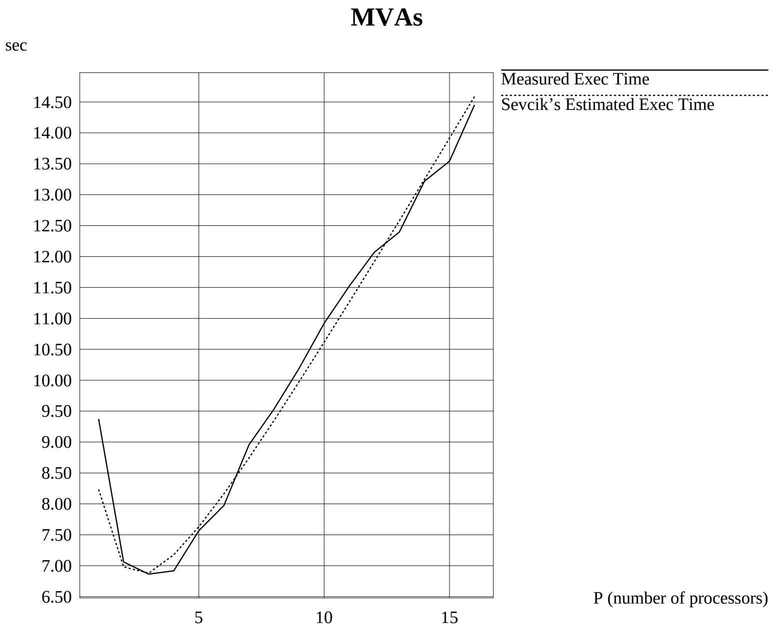

Figure 4.9: Averaged execution time versus Sevcik's estimate for MVA_s

number of processors approaches 16. MVA$_l$'s execution time stops decreasing when the number of processors reaches 6. The worst of all is MVA$_s$, which speeds up only up to two processors. As can be seen from the figures, Sevcik's model accurately approximates each application's execution time function regardless of the application's speedup property.

4.3 Dowdy's Model of Execution Time Function

We can also use Dowdy's *execution signature*,

$$\mu_j(p) = \frac{p}{C_{j1}p + C_{j2}},$$

of an application j to obtain an approximation for its execution time function. In this equation, $\mu_j(p)$ indicates the execution rate of an application j, and C_{j1} and C_{j2} are two constants that characterize application j. When $C_{j1} = 0$, application j possesses linear speedup. As more processors are allocated to application j, it speeds up linearly by a factor of $\frac{1}{C_{j2}}$. However, when $C_{j2} = 0$, application j executes at a constant rate of $\frac{1}{C_{j1}}$. In this case, the execution time of application j is independent of the number of processors allocated to it. Dowdy suggested in his work that the terms C_{j1} and C_{j2} will normally be positive values.

By taking the reciprocal of $\mu_j(p)$, we get $1/\mu_j(p)$ which is the execution time of application j running on p processors. Thus, the function,

$$\mathcal{T}_j(p) = \frac{1}{\mu_j(p)} = C_{j1} + \frac{C_{j2}}{p},$$

can be used as an approximation of the execution time function of application j. If we compare the above model with the model proposed by Sevcik, we can alternatively interpret C_{j1} as the overhead of parallel processing, and C_{j2} as the service demand of application j.

Again, we took the measured execution times of an application j using 1 to 16

Application j	C_{j1} in ms	C_{j2} in ms
MM_l	4993	175311
MM_s	1678	21645
MVA_l	21123	33940
MVA_s	11162	-4746
$GRAV_l$	13985	47669
$GRAV_s$	4023	6813

Table 4.2: Parameters of Dowdy's approximated execution time functions

processors[2], then applied least squares approximation. However, function of two terms,

$$F_j(p) = A_j + \frac{B_j}{p},$$

is used this time where A_j approximates C_{j1}, B_j approximates C_{j2}, and $F_j(p)$ approximates $\mathcal{T}_j(p)$.

Table 4.2 gives the set of parameters obtained for each application's approximated execution time function by applying Dowdy's model. Notice that $C_{j2} < 0$ for MVA_s. Under Dowdy's model, MVA_s will not be considered as a normal application. The effect of a negative C_{j2} will become more apparent as we examine the approximated execution time curves produced by these C_{j1} and C_{j2} values.

The graphs of the averaged measured execution time curves together with estimated execution time curves using Dowdy's approximation for the six chosen applications are presented in Figure 4.10 to Figure 4.15. Notice that for applications such as MM_l, MM_s and $GRAV_l$, which continuously speed up as the number of processors increases (in this case up to at least 16 processors), Dowdy's model provides a very accurate approximation. However, if an application fails to speed up after a certain number of processors, such as $GRAV_s$, MVA_l and MVA_s, his model approximates poorly. The worst case is with MVA_s, Dowdy's model even fails to approximate the shape of its measured execution time curve (as a result of the negative C_{j2} for MVA_s).

The poor behavior of Dowdy's model in approximating applications with poor

[2]This time all 16 data points were used for the least squares approximation because using data points in 2,4,8,12,16 gives significantly different results than using all 16 points.

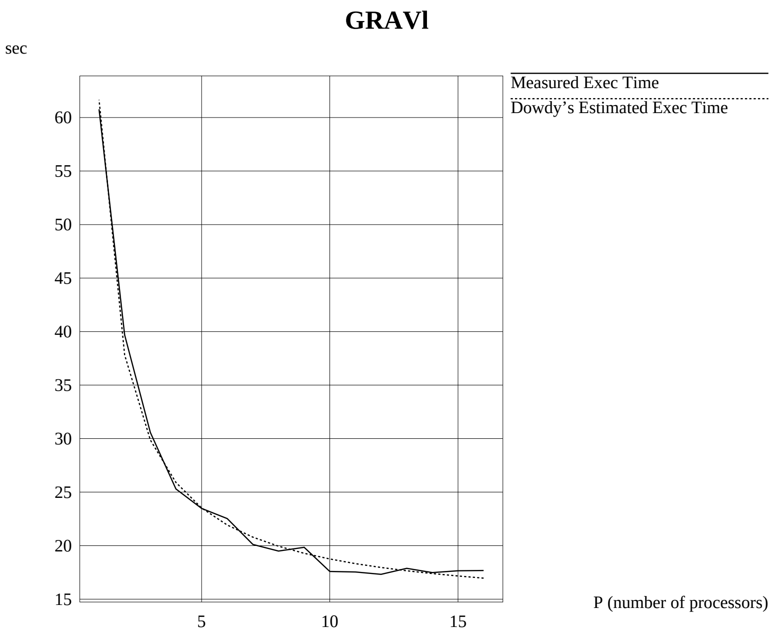

Figure 4.10: Averaged execution time versus Dowdy's estimate for $GRAV_l$

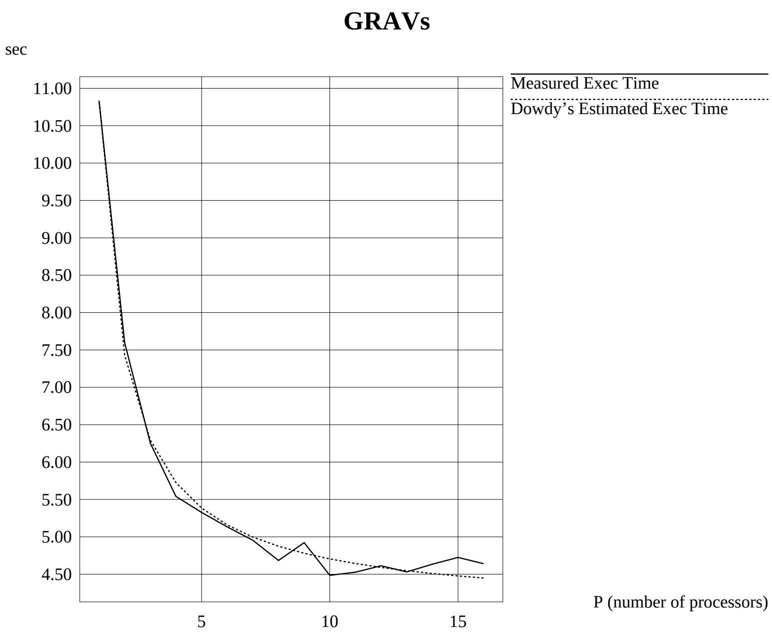

Figure 4.11: Averaged execution time versus Dowdy's estimate for $GRAV_s$

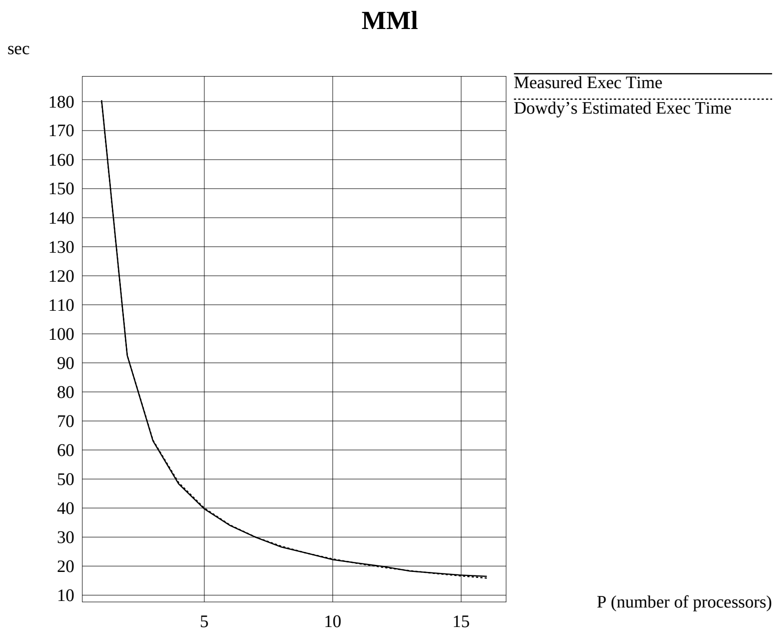

Figure 4.12: Averaged execution time versus Dowdy's estimate for MM_l

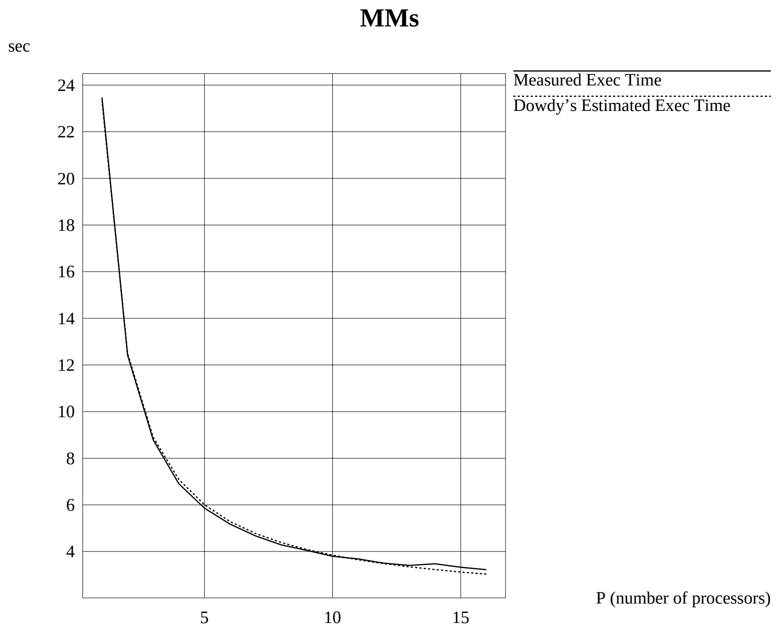

Figure 4.13: Averaged execution time versus Dowdy's estimate for MM_s

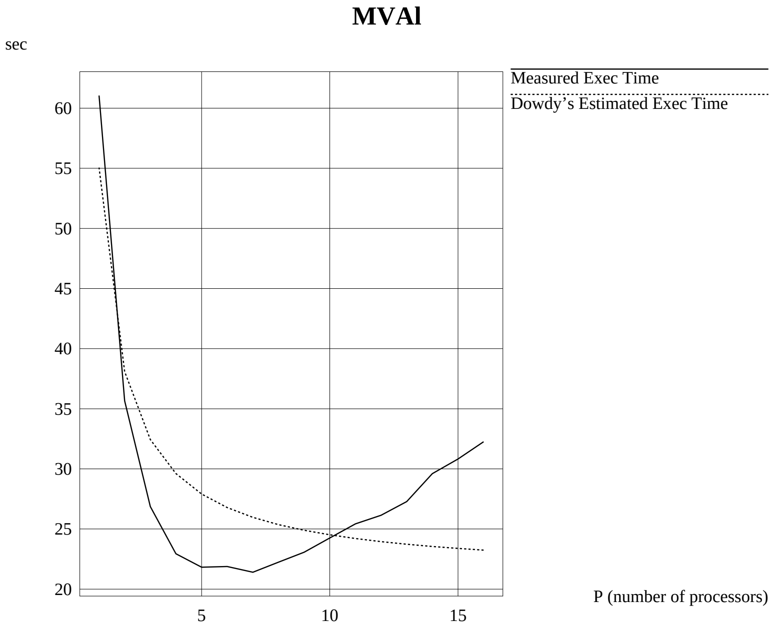

Figure 4.14: Averaged execution time versus Dowdy's estimate for MVA_l

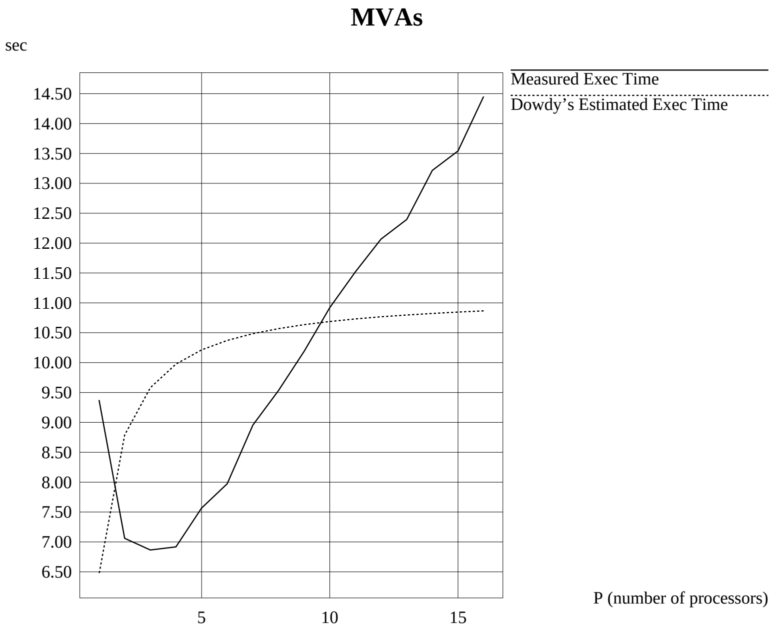

Figure 4.15: Averaged execution time versus Dowdy's estimate for MVA_s

speedup, such as MM_s, is because his model fails to include a factor that accounts for the communication and congestion delays, which grow with the number of processors. Since Dowdy's model for approximation of execution time functions is a decreasing convex function, only applications that continuously speed up can be accurately represented. Note that in Sevcik's model, the communication and congestion delays are represented by the term β_j, thus his model approximates these types of applications accurately.

In next chapter, we will present two static policies which make use of the approximated execution time function of applications using Sevick's model. Their performance will be compared with a dynamic policy in a later chapter.

Chapter 5

The Scheduling Policies

In this chapter, the three policies that we studied, the Static, the Immediate Start Static and the Dynamic policies, are introduced in the first section. Then, some issues that should be considered when making reallocation decisions are discussed. In the section that follows it, we will describe how application characteristics are applied to our policies. Detailed descriptions of each of the three policies are presented in three later sections. Finally, implementation details of each policy will be given.

5.1 Policies and Considerations

The scheduling policies studied in this work are all space-sharing policies that differ in their frequency of reallocations. They are the Static (STA) policy, the Immediate-Start-Static (ISS) policy, and the Dynamic (DYN) policy. In the Static policy, reallocations are performed only when some application completes. If there are no processors available, new arrivals are not started until some active application terminates and releases its set of processors. Applications hold the same number of processors from the beginning until the end of their execution without any readjustments. The Dynamic policy, on the other hand, reallocates frequently in response to the changes in the parallelism of each application. The hope is to avoid processor idling by quickly switching a processor from an application that does not require it to an application that can immediately utilize it. The Immediate Start Static policy, which performs processor reallocations at application arrivals as well as at completions, is a modified static policy that tries to start a new application as soon as possible. The Immediate Start policy falls between the Static policy and the Dynamic policy, in terms of frequency of allocations.

There are some options available for a kernel scheduler when it tries to reallocate at application arrivals or at application completions.

At an application arrival, the kernel scheduler can choose among the following actions:

1. If some processors are idle, then these idle processors may be allocated to the new application. Otherwise, queue the new application and wait for processors to become available.

2. If some processors are idle, then these idle processors may be allocated to the new application. Otherwise, one or more processors are preempted from active applications and allocated to the new application to start it up.

3. Readjust the current allocations to the set of active applications including the new one (regardless of whether there are idle processors) by preempting processors from active applications.

When an application completes, these are the options for the scheduler:

1. Leave the processors idle until the next arrival (if there are no applications in the waiting queue).

2. Assign the released processors to various active applications.

3. Give the first application in the waiting queue all the released processors.

4. Give each application in the waiting queue at least one released processor, then use the remaining released processors for readjustment.

5. Readjust the current allocations including all waiting arrivals by allocating the released processors and/or preempting processors from other active applications.

When doing allocation readjustments, the scheduler must follow some rules or criteria. For example, in Equipartition, the guiding principle is to assign an equal number of processors to all applications. When using the Dynamic policy, the rule is to reflect the current parallelism of each individual application. In the static policies we study, STA and ISS, the rule is to assign an available processor to the application which will benefit the *most* from an extra processor, and to require the application which will suffer the *least* to release a processor when one is needed.

In order to facilitate reallocations, an application must be able to execute on any number of processors. Two-level scheduling is useful in this regard. With two-level scheduling, the kernel scheduler's responsibility is to allocate processors among competing applications according to their needs and priorities, while taking into account the overall system load; the application level thread dispatcher (or thread package) is responsible for distributing threads of the application onto the system-allocated processors. This allows any application to be able to run on any number of processors. By using two-level scheduling, the kernel scheduler can adjust the processor allocations among applications freely. Another benefit of using a thread package is that programmers do not have to worry about communicating with the scheduler when writing parallel programs; the thread package would handle it.

5.2 Applying Application Characteristics

In the last chapter, we studied two models for approximating the execution time function of an application. Since Sevcik's model is more accurate than Dowdy's model, it was chosen as a basis for our scheduling policies. The four parameters in Sevcik's model are used as application characteristics, and are provided to the scheduler by the thread package at runtime.

In our static policies, STA and ISS, processors are reallocated either at application completions or at both application arrivals and completions. At these reallocation points, the scheduler will use the approximated execution time function parameters to calculate the marginal gain of having an extra cpu and the marginal loss of having one

P	MM_l	MM_s	MVA_l	MVA_s	$GRAV_l$	$GRAV_s$
2	87458.8	10995.0	24793.3	1254.2	21097.9	3234.4
3	29659.4	3710.7	9050.9	100.9	9570.4	1394.5
4	14767.2	1833.8	3639.5	-295.5	4654.2	666.3
5	8810.3	1083.1	1474.9	-454.1	2687.7	375.0
6	5831.9	707.7	392.6	-533.4	1704.5	229.3
7	4129.9	493.2	-225.9	-578.3	1142.6	146.1
8	3066.2	359.2	-612.4	-607.0	791.5	94.1
9	2357.0	269.8	-870.1	-625.9	557.4	59.4
10	1860.6	207.2	-1050.5	-639.1	393.5	35.1
11	1499.6	161.8	-1181.7	-648.8	274.3	17.5
12	1228.8	127.6	-1280.1	-656.0	184.9	4.2
13	1020.6	101.4	-1355.7	-661.5	115.2	-6.0
14	856.9	80.8	-1415.2	-665.9	62.1	-13.9
15	726.0	64.3	-1452.8	-669.3	18.9	-20.4
16	619.6	50.8	-1501.4	-672.2	-16.2	-25.6
P_{max}	37	22	6	2	15	12

Table 5.1: Marginal difference in estimated execution time and p^{max}

less cpu for all active applications. The marginal gain of an application j currently with p processors is

$$T_j(p) - T_j(p+1);$$

and similarly the marginal loss of an application j currently with p processors is

$$T_j(p-1) - T_j(p).$$

The parameters in Sevcik's model are also used to calculate a number p_j^{max} for each application j. This number, p_j^{max}, is the maximum number of processors that an application j can effectively use without causing slowdown and is derived from Sevcik's model [Sev92]. We can calculate p_j^{max} using the following equation,

$$p_j^{max} = \sqrt{\frac{\phi_j W_j}{\beta j}}.$$

In our static policies, p_j^{max} is used to indicate the maximum number of processors an application should ever be allocated.

Table 5.1 presents the values of marginal gain and marginal loss at different number of processors of all six applications. Their respective p^{max} values are also included. In any particular row p, the entry represents the marginal gain of the corresponding application with $p-1$ processors and the marginal loss with p processors.

5.3 The Static Policy

Under the Static policy, the scheduler allocates a fixed number of processors to an application when it first arrives in the system, and the application holds the same number of processors until completion. This number is decided by the scheduler, and it is either p_j^{max} [Sev92], the maximum number of processors that an application j can effectively use without causing any slowdown, or the total number of unallocated processors in the system. Under this policy, processors are unallocated only when all active applications have been allocated their respective p_j^{max} number of processors, and there are no arrivals waiting for processors; and processors can become available only when some active application completes and releases its processors.

Thus, at application arrivals,

- If there are unallocated processors, the scheduler will allocate all of them to arriving application (subject to the p_j^{max} limit).
- Otherwise, the arrival is queued until some processors become available. The arrivals are placed in a Smallest-Service-Demand-First (SSDF) queue.

When an active application completes,

- The released processors are allocated one at a time to the applications at the waiting arrival queue.
- If there are remaining free processors, they are allocated one at a time to the application(s) that can benefit most from an extra processor (the application(s) with the largest marginal gain).

5.4 The Immediate Start Static Policy

Most parallel applications do not have linear speedup. Allocating an extra processor to an application that has sublinear speedup will cause some delay to all subsequent applications that use the same processors, even though it may reduce the execution time of the application with the extra processor. Thus, it is important for a scheduler to be sensitive to the number of applications ready for execution and the overall system load. As the queue of ready applications becomes longer or as the system load increases, fewer processors should be allocated to each application in order to minimize overall average response time. A simple approach to making the scheduler sensitive to the length of the ready application queue and to the system load is to start a new application as soon as possible (ideally immediately) if there are active applications with more than one processor. This will likely require preempting processors from active applications at application arrivals.

A major disadvantage of the Static policy is that it fails to start up new arrivals quickly when there are applications with more than one processor (even if they cannot use all their processors effectively). The Immediate Start Static policy avoids this problem by requiring an active application (with more than one processor) to give up a processor to a new arrival when there are no idle processors. Hence, in this policy, in addition to during application completions, a processor can also become available when an application is requested to release a processor by the scheduler.

In the Immediate Start Static policy, when a new application arrives,

- If there are idle processors, they are allocated to the new application (subject to the p_j^{max} limit).
- Otherwise, the application is placed in a SSDF waiting queue. An active application that uses its last processor least effectively (the application with the smallest marginal loss of losing a processor) is identified and asked to give up a processor.

When a processor is given up by an active application due to the system scheduler's request,

- The processor is allocated to the first application in the waiting queue.

At application completions, the actions are the same as in the Static policy.

5.5 The Dynamic Policy

Our Dynamic policy is based on the dynamic policy defined by McCann, Vaswani and Zahorjan [MVZ91]. Under this policy, processors are reallocated to reflect the changes in an application's parallelism. The runtime thread dispatcher of an application continuously notifies the system scheduler of how many extra processors it could use. This number corresponds to the number of ready but not executing threads of the application. When the thread dispatcher of an application realizes that there are no more ready threads for execution and a processor is available, it notifies the kernel scheduler that a processor is available for reallocation (it is *ok to yield*). When an application needs more processors, the scheduler tries to satisfy its request by using the least valuable processors:

- First, it allocates the application all unallocated processors.
- If the request is not satisfied yet, *ok to yield* processors are preempted and allocated.
- If the request is still unsatisfied, an equipartition allocation is enforced by requiring some active applications to give up processors to the one making the request.

The Dynamic policy requires applications to acquire and release processors frequently. In a NUMA environment, this frequent change of allocations may disrupt the data locality of each application and affect the performance of each individual application as well as the overall system.

5.6 Implementation Details

Our kernel scheduler which implements the different scheduling policies is written as a user-level process. It runs on a designated processor and waits for requests from the applications. All our applications are linked with a modified runtime thread package called SP[1], which is the University of Toronto variation of the Presto thread package developed at the University of Washington [BLL88]. The kernel scheduler exchanges information with the applications through the help of the thread package. For instance, the parameters that characterize the applications are sent to the kernel scheduler from the thread dispatcher at the start of each application. The pseudo code of the kernel scheduler and the application level thread dispatcher is given in the appendix.

Unlike the dynamic policy described by McCann, Vaswani and Zahorjan [MVZ91], all the policies we study avoid forced preemptions. When the kernel scheduler wants an active application to release one of its processors, it simply asks the application to release a processor at its next convenient point. For our implementation, this point corresponds to the next thread completion for the application. With our assumption that the applications are written in relatively fine grains (i.e., using a large number of threads), this approximates immediate preemptions.

There are several reasons for avoiding forced preemptions:

1. Forced preemption requires an application to block a running thread in order to give up a processor. If that thread is in a critical region, then the progress of the entire application will be affected since other threads of the application could waste processor cycles by continuously testing and trying to enter the critical section.

2. We could prevent the problem of preempting processes with critical threads by using a scheme called scheduler activations proposed by Anderson [And91]. However, this method requires a number of thread and process reschedulings,

[1]The name SP is a mystery even to the author.

which is costly.

3. Also, in a NUMA environment, these reschedulings of threads and processes will disrupt the data locality of applications. As a result, performance will be affected.

4. A scheduler that performs forced preemptions is typically more complex and requires more communication between the processor scheduler and the thread dispatcher than one that does not force preemptions.

For the reasons above, we have chosen not to implement forced preemptions in our scheduler.

Chapter 6

Experiment Results

The results of our experiments will be presented in this chapter. However, we will first describe the workload mixes that were used in our experiments.

6.1 Workload Mixes

The applications that we have chosen in Chapter 4 are combined to create workloads from Poisson arrival streams with average interarrival time η[1]. For our experimentation purposes, we have chosen a range of values for η to study the performance of the scheduling policies under various system loads. The chosen values of η are 3, 5, 10, and 20 measured in seconds. The value of η determines the overall load on the system, so our workloads represent a range of loading factors to the system. Notice that the smaller the value of η, the heavier the corresponding workload is to the system. Poisson arrival streams for all four η values were created for 30 jobs. These four streams are shown in Table 6.1. In any particular row r, entry 1 gives the order of the job arrival relative to other jobs; entry 2 indicates the application used for the job in row r; entry 3 to entry 6 give the interarrival time between the job in row r and the job in row $r-1$ with different η. Note that entry 3 to entry 6 are linearly dependent.

[1]Note that $\eta = \frac{1}{\lambda}$, where λ is the arrival rate.

Job Order	Assigned Job	η=3	η=5	η=10	η=20
0	MVA_s	0.0	0.0	0.0	0.0
1	GRAV_s	1.13	1.89	3.79	7.58
2	MVA_s	9.89	16.48	32.96	65.93
3	MM_l	1.22	2.03	4.06	8.12
4	GRAV_s	0.87	1.45	2.91	5.81
5	MM_s	2.53	4.21	8.42	16.83
6	GRAV_s	0.26	0.44	0.88	1.75
7	MM_s	2.03	3.38	3.38	6.76
8	MVA_s	8.57	14.38	28.57	57.14
9	MM_s	2.37	3.95	7.90	15.81
10	GRAV_s	1.12	1.89	3.79	7.57
11	GRAV_l	0.72	0.12	0.23	0.47
12	GRAV_s	0.15	0.25	0.49	0.99
13	MVA_s	4.43	7.38	14.77	29.54
14	MM_s	2.30	3.82	7.63	15.27
15	MVA_s	5.98	9.38	18.76	37.53
16	MVA_s	12.56	20.93	41.86	83.73
17	MVA_l	3.62	6.04	12.09	24.17
18	GRAV_s	0.36	0.60	1.20	2.40
19	GRAV_s	1.11	1.85	3.69	7.39
20	MM_s	1.72	2.87	5.75	11.50
21	GRAV_s	1.20	2.00	4.00	8.00
22	MVA_s	7.91	13.18	26.35	52.70
23	MVA_s	6.43	10.71	21.42	42.84
24	MM_s	2.45	4.08	8.17	16.34
25	MM_s	1.54	2.56	5.12	10.24
26	GRAV_s	0.80	1.33	2.65	5.30
27	GRAV_s	0.63	1.05	2.11	4.22
28	MM_s	1.45	2.41	4.82	9.65
29	MM_s	1.60	2.66	5.32	10.63

Table 6.1: Poisson stream interarrival times generated for each η

Application j	$T_j(1)$
MM_l	180.456s
MM_s	23.474s
MVA_l	61.055s
MVA_s	9.375s
GRAV_l	60.776s
GRAV_s	10.836s

Table 6.2: Execution time of each application using one processor

How the applications are associated with the order of arrivals (and hence the arrival times) is determined using the following method. First, an application pool is created from which applications would be drawn randomly. Assuming that small applications are submitted more frequently than large applications, we put six small applications in the application pool for every large application[2]. Then, we draw, in order, 30 applications from the pool. Each application is assigned, corresponding to its drawn order, the interarrival time from the Poisson arrival streams generated earlier for each η (as listed in Table 6.1). This creates a total of four workloads, each generated from a different η-Poisson arrival stream but with the same order of arrivals among the six chosen applications.

In order to determine how intensive our workload is to the system, we will calculate the offered load intensity, ρ, when each of the four workload streams is submitted to the system. The offered load intensity, ρ, is defined as

$$\rho = \frac{1}{\eta} * \frac{1}{p} * \frac{1}{N} * \sum_{j=1}^{N} T_j(1)$$

where p is the number of processors in the system, N is the number of jobs in the workload, and $T_j(1)$ is the execution time of application j using one processor. Table 6.2 gives the execution time of each of our chosen applications using only one processor. These values are taken from Table 4.1 and are used to calculate the load intensity of our workload at each η, which is presented in Table 6.3.

For each given η, the workload was executed under each scheduling policy of

[2]The number 6 is the ratio of the application's execution time running on 1 and 8 processors.

η	ρ
20	9.7%
10	19.4%
5	38.7%
3	64.5%

Table 6.3: Load intensity of the workload at different η

interest. The performance of the scheduling policies for each workload is summarized in the next few sections.

6.2 Experiment Details

We performed all our experiments on Hector using all 16 processors. Of all 16 processors, four of them that reside on one station were dedicated to system programs: one is used to execute the scheduler; one is used for a "launch" program which submits jobs according to the workloads; one is used for all thread package related processes; and one is used to execute a program which monitors the behavior of the scheduler. These four processors are called system pool processors. The other 12 of the 16 available processors were used to run the applications. We decided to use this scheme because:

- We want to minimize the overhead of system programs on the performance of applications.

- We do not want the processors to be time-shared among system processes and application processes. Using this scheme, when an application is allocated a processor, its process will be the only process running on it.

For each of our measurements, eight runs were performed and the average of mean response times among the eight runs was calculated and presented. Confidence intervals of 90% were also included in each measurement.

η	STA	ISS	Difference
∞	11.76 ± 0.21	12.67 ± 0.24	-7.2%
20	17.61 ± 0.54	15.23 ± 0.18	15.6%
10	20.76 ± 0.52	17.28 ± 0.37	20.1%
5	32.85 ± 1.45	23.39 ± 0.34	40.1%
3	33.69 ± 1.80	24.10 ± 0.57	39.8%

Table 6.4: Mean response time under STA and ISS

6.3 STA versus ISS

One of the drawbacks of the Static policy is that it fails to start an application soon after it has arrived. The Immediate Start Static policy, however, addresses this drawback and starts a newly arrived application almost immediately. To determine the effect of this change in the Static policy, and the benefit of reallocating both at arrivals and completions, the performance of STA and ISS were studied and compared.

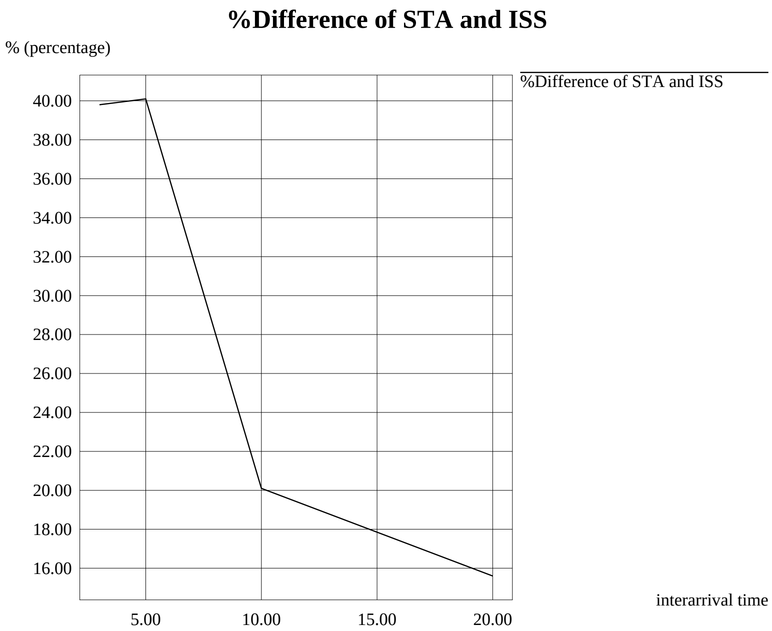

Figure 6.1: Percentage Difference of Performance for STA and ISS

Table 6.4 presents the performance of STA and ISS on 12 processors. A workload of $\eta = \infty$ is also included for comparison. This workload represents the minimum average response time without multiprogramming. Notice that this minimum average response time is 7.2% larger for the ISS policy than for the STA policy because the implementation of ISS is slightly more complicated than STA.

In all four workloads we studied, ISS outperforms STA by 15-40%. The STA policy prevents the completion of small applications by not starting a new application soon after it arrives, even though there may be other applications with more than one processor. This delay in completion of one application will lead to further delay to subsequent applications in the workload. As a result, the mean response time of STA is much worse than ISS. The benefit of immediately starting a new application becomes more apparent as the system load increases (see Table 6.4). As η decreases in our workload (i.e., as the system load increases), the percentage difference in mean response time between ISS and STA increases from 15.6% to 39.8%[3] (see Figure 6.1). This is because when using the STA policy, the waiting queue is longer when the system load is high. The longer the waiting queue, the poorer the performance is since remaining applications in the workload are subject to more subsequent delays.

Since STA is a policy that reallocates only at completions, and ISS is a policy that reallocates both at arrivals and completions, the result of this experiment also indicates the importance of reallocating at *both* arrivals and completions.

6.4 Performance using FCFS queue

We would like to study the effects of different queueing methods in our static policies. In our original STA and ISS policies, Smallest-Service-Demand-First (SSDF) queues were used for the new arrival waiting queues. We performed the same set of experiments for STA and ISS policies as the ones in the last section, but this time using First-Come-First-Served (FCFS) queues instead. The performance of the two static policies is expected to be significantly worse in the new FCFS versions than in the original SSDF versions.

As shown in Table 6.5, the FCFS versions of STA and ISS perform worse than the SSDF versions. However, the difference in performance is not as significant as expected. Notice that in order to see a difference in performance using SSDF and

[3] with $\eta = 5$ slightly higher than $\eta = 3$

η	STA-SSDF	STA-FCFS	Difference	ISS-SSDF	ISS-FCFS	Difference
20	17.61 ± 0.54	18.23 ± 0.46	0.04%	15.23 ± 0.18	15.10 ± 0.20	-0.01%
10	20.76 ± 0.52	21.06 ± 0.57	0.14%	17.28 ± 0.37	17.46 ± 0.46	0.01%
5	32.85 ± 1.45	33.14 ± 0.91	0.01%	23.39 ± 0.34	23.84 ± 0.28	0.02%
3	33.69 ± 1.80	33.98 ± 2.51	0.01%	24.10 ± 0.57	25.08 ± 1.33	0.04%

Table 6.5: Mean response time under STA and ISS using SSDF and FCFS queues

FCFS, the following conditions would need to be satisfied:

1. For most of the time, the waiting queue should be reasonably long. Otherwise, SSDF and FCFS would be essentially the same since SSDF would have very few applications to choose from.

2. The time between starting two new applications is relatively long. If applications in the waiting queue are started in groups at a time, again SSDF and FCFS would be essentially the same since either policy would attempt to start many applications at a reallocation point.

3. The service demands of the queued applications are significantly different. This would make the benefit of SSDF more apparent since SSDF prevents large applications from starting first and keeps them from dominating the system.

In the STA policy, waiting applications are started only when some active application completes and releases its processors. This typically would cause most (if not all) of our applications in the waiting queues to be started regardless of which queueing method we use. Thus, the benefit of SSDF is not large. In the ISS policy, because we are starting newly arrived applications almost immediately, the waiting queue is usually very short. In our experiments, the length of the waiting queue is never greater than two[4], while on average, there are six to seven applications running in the system concurrently. For this reason, the effect of different queueing methods on mean response time is also insignificant in the ISS policy.

[4]If forced preemptions were used, the waiting queue would have been empty at any particular instance.

η	ISS	DYN	Difference
∞	12.67 ± 0.24	13.09 ± 0.33	3.3%
20	15.23 ± 0.18	16.21 ± 0.29	6.4%
10	17.28 ± 0.37	18.17 ± 0.38	5.2%
5	23.39 ± 0.34	24.83 ± 1.25	6.2%
3	24.10 ± 0.57	28.76 ± 0.85	16.2%

Table 6.6: Mean response time under ISS and DYN

6.5 ISS versus DYN

In their work [MVZ91, VZ91, ZM91], McCann, Vaswani and Zahorjan showed that in UMA systems, their Dynamic scheduler outperforms more static policies such as Equipartition. However, as mentioned in Chapter 2, in the NUMA environment that we are interested in, it is not clear that the Dynamic policy will produce better mean response time for applications than a more static policy. Moreover, in the static policies that McCann, Vaswani and Zahorjan studied, no application characteristics were used to improve scheduling decisions. We believe that utilizing application characteristics is vital to the performance of applications and the overall system, and could possibly lead to a static policy that outperforms the Dynamic policy.

In the static policies STA and ISS we proposed, knowledge about the the execution time of applications using different number of processors was provided to the scheduler. The scheduler attempts to make more informed reallocation decisions by calculating the marginal gain and marginal loss of each application at reallocation points.

The performance of the Immediate Start Static policy compared to the Dynamic policy is presented in Table 6.6. Again, the $\eta = \infty$ workload is included for comparison. Since when $\eta = \infty$, there are no multiprogramming. The overhead of DYN in this case should not contain any cost of context switching, and subsequently any cost due to extra cache reloads and loss of data locality. However, because of the more complicated algorithm and the constant exchange of information between the scheduler and the application thread dispatcher, the minimum average response time without multiprogramming for the DYN policy remains larger than the ISS policy.

The 3.30% difference between these numbers indicates that the overhead of constant communication and the more complicated algorithm is not a major factor affecting the performance of DYN.

The ISS policy produced mean response time that are 5-17% smaller than the DYN policy consistently throughout the four workloads. The slightly worse performance of DYN compared to ISS is because of the extra overhead in DYN. These overheads include the overhead of context switching among applications, the overhead of constant communication between the scheduler and applications (small as indicated above), the overhead of more frequent cache reloads, and the overhead of more costly memory accesses due to loss of data locality.

Any application with sublinear speedup is more likely to fully utilize all its processors when a small number of processors is allocated to it than when a large number of processors is allocated to it. In our experiments with $\eta = 3$ workload, the system load is high enough that most of the time there are more than eight applications active in the system. This results in an allocation of no more than two processors to each active application. With less than two allocated processors, most applications can fully utilize all their processors. As a result, the benefit of DYN's frequent allocation readjustments to attempt to utilize the processors fully is less significant. This contributes to the relatively large difference in response time between ISS and DYN in the $\eta = 3$ workload.

6.6 Performance with Small Relative Overhead

If the overhead of DYN is made smaller relative to the service demands of the applications, the performance of DYN should improve. We attempt to test this observation by creating a new workload called *DoubleSD*. The *DoubleSD*-workload consists of the same ordering and interarrival times among applications as the $\eta = 10$ workload. However, we double the service demand of each application without changing its parallelism structure. By doing this, the *DoubleSD*-workload offers a similar load intensity to the system as the $\eta = 5$ workload.

ISS	DYN	Difference
38.14 ± 2.99	33.14 ± 0.55	-13.1%

Table 6.7: Mean response time of *DoubleSD*-workload under ISS and DYN

Table 6.7 shows the average response time of the *DoubleSD*-workload under the ISS and DYN policies. The DYN policy outperforms the ISS policy by 13.1%. Because of doubling the service demands of our applications, the overhead of the DYN policy becomes relatively smaller. The cost of attempting to fully utilize the processors by frequent reallocations in this case is justified. This explains the improved performance of DYN compared to ISS using the *DoubleSD*-workload.

6.7 Performance on Simulated Larger Systems

Using Hector, we were able to control the speed of the network to a limited degree. The system provides a set of switches which allow the user to add additional delays to off-station packets (ring delays), from 1 to 16 cycles, in powers of 2. Every packet that is destined for an off-station processor-memory pair is held up at the source station interface for the number of cycles selected. Packets destined for an on-station processor-memory pair are unaffected. The intent of these delays is to make the stations appear further away from each other, as if in a larger system. With the use of this delay hardware, we attempted to gain some insight into the expected performance of the policies on larger systems.

Table 6.8 shows the mean response time of ISS and DYN on Hector with different ring delays in effect. The first four rows with delay equal to zero are extracted from Table 6.6 for comparison. The Immediate Start Static policy produces smaller mean response time than the Dynamic policy in all cases. Under DYN, because of the frequent switching of processors from application to application, data locality is disrupted. This loss of data locality results in extra memory access costs since some memory accesses would require data fetching from remote PMs (i.e., other PMs on the same station, or PMs on other stations). The ISS policy, however, try to avoid data locality disruption by allocating a relatively static set of processors to each ap-

Delay	Workload	ISS	DYN	Difference
0	20	15.23 ± 0.18	16.21 ± 0.29	6.4%
	10	17.28 ± 0.37	18.17 ± 0.38	5.2%
	5	23.39 ± 0.34	24.83 ± 1.25	6.2%
	3	24.10 ± 0.57	28.76 ± 0.85	16.2%
2	20	15.73 ± 0.15	17.57 ± 0.15	11.6%
	10	18.65 ± 0.22	19.35 ± 0.34	3.8%
	5	24.57 ± 0.51	26.27 ± 0.70	6.9%
	3	27.17 ± 0.53	31.88 ± 1.95	17.3%
4	20	17.51 ± 0.56	19.19 ± 0.30	9.9%
	10	19.95 ± 0.98	21.37 ± 1.20	7.1%
	5	25.37 ± 0.93	31.59 ± 1.06	24.5%
	3	30.16 ± 1.49	35.89 ± 2.59	18.9%
8	20	21.75 ± 0.08	23.32 ± 2.00	7.2%
	10	25.90 ± 1.03	27.01 ± 0.64	4.3%
	5	33.79 ± 3.08	42.97 ± 4.67	27.1%
	3	35.77 ± 1.48	43.83 ± 2.32	22.5%

Table 6.8: Mean response time of ISS and DYN with different ring delays

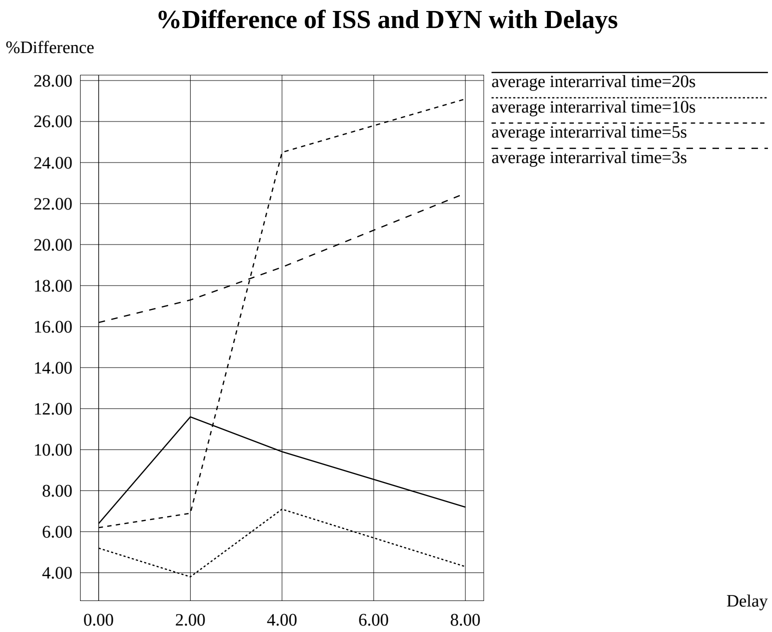

Figure 6.2: Percentage Difference of Performance for ISS and DYN with delays

plication. Hence, the extra ring delay does not affect the performance of ISS as much. Recall that as the delay becomes larger, across station memory accesses become more costly. As a result, the percentage difference in mean response time between the ISS and DYN under $\eta = 3$ and $\eta = 5$ becomes larger as the value of delay increases from zero to eight (see Figure 6.2). This demonstrates that at moderate to heavy load, the ISS policy becomes more suitable for NUMA systems as the system size increases. However, the increase cost of remote memory accesses do not affect the performance of DYN as much when using light workloads. This is because when the system load is light, the multiprogramming level is low. Because there are only a few applications in the system, there are fewer allocation readjustments among applications, and the processor allocations become relatively static. Thus, the performance difference between the ISS and DYN remains between 3% to 12% when $\eta = 10$ and $\eta = 20$ as the ring delay increases.

6.8 Single Application Workloads

The Dynamic policy is expected to perform well on applications with distinct phases whose need for processors changes significantly from phase to phase. Our Gravity application possesses this property. In these applications, the performance gain of better utilized processors outweighs the extra overhead of the Dynamic policy. However, when the variability in an application's parallelism profile is low, such as our Matrix Multiply application, the Dynamic policy's extra overhead will become significant. The need for making frequent scheduling decisions (to try to fully utilize the processors) in such applications is less than in those with highly variable parallelism profiles. We try to confirm this observation through experiments. With all the η workload in Table 6.1, we replaced all the GRAV_s and MVA_s applications by MM_s, and all the GRAV_l and MVA_l applications by MM_l, creating a workload consisting of only MM applications. We call this the MM-workload. Similarly, a GRAV-workload was created by replacing all MM and MVA applications with GRAV applications.

Table 6.9 compares the performance of ISS and DYN using single application

Workload	η	ISS	DYN	Difference
MM	20	7.00 ± 0.20	7.96 ± 0.40	13.7%
	10	9.60 ± 0.10	10.31 ± 0.23	7.4%
	5	14.41 ± 0.48	17.67 ± 0.73	22.6%
	3	29.39 ± 0.72	36.09 ± 2.10	22.7%
GRAV	20	19.61 ± 0.17	17.45 ± 0.13	-11.0%
	10	18.90 ± 0.19	16.52 ± 0.23	-12.6%
	5	19.08 ± 0.33	17.39 ± 0.55	-8.8%
	3	20.77 ± 0.79	20.17 ± 1.17	-2.9%

Table 6.9: Mean response time under ISS and DYN using single application workloads

workloads. The ISS policy is more appropriate for the MM-workload, which consists of applications with minimal variation in parallelism, at all values of η. However, the DYN policy performs very well on the GRAV-workload, which consists of applications with distinct phases of parallelism. This is consistent with our expectation that applications with significantly different phases of parallelism are better being executed under the Dynamic policy. Note that as the load intensity on the system becomes larger, the performance gain of ISS over DYN on the MM-workload increases (with the exception of $\eta = 10$ case), while the performance improvement of DYN over ISS on the GRAV-workload becomes less significant.

When a workload is comprised of some applications of MM-like structure and some applications of GRAV-like structure, we can use the following guidelines to choose between using ISS or DYN as the scheduling policy.

- When the system load is moderate to high, the ISS policy should be chosen. This is supported by the fact that the performance of ISS approaches DYN on a GRAV-workload as the system load increases, while the performance of DYN becomes worse compared to ISS on a MM-workload as the system load increases.

- If the average service demand of all the MM-like applications in the workload is greater than the average service demand of all the GRAV-like applications, the ISS policy would again be a better choice.

- As the service demands of applications become larger, the DYN policy becomes more appropriate. This is as a result of our experiments with the *DoubleSD*-workload.

- The DYN policy is less likely to be the better choice as the NUMAness of the system increases. This is because the overhead of data locality disruption of DYN is more apparent in NUMA systems in which the cost difference between local memory accesses and remote memory accesses is large. This is supported by our result from the simulated large systems experiments in the last section.

Chapter 7

Conclusion

In this thesis, we presented our experimental study of the problem of processor scheduling in multiprogrammed NUMA shared-memory environments. We concentrated on two aspects of the problem: how application characteristics can be applied in processor scheduling and their effectiveness; and how well a dynamic policy performs in NUMA systems compared to more static policies that utilize application characteristics.

The application characteristics that we examined are mathematical models that approximate the execution time function of applications. Two models that were previously suggested by Sevcik and Dowdy respectively were studied. Their accuracies were compared using parallel applications. We showed that Sevcik's model provides an accurate approximation of execution time function for applications of different parallelism structure. In particular, Sevcik's model approximates applications that begin to slow down after some number of processors more accurately than Dowdy's model.

Then, we presented two static scheduling policies, the Static policy and the Immediate Start Static policy, that take Sevcik's approximated execution time function of applications as application characterizations. Both policies use a marginal gain function and a marginal loss function, which are based on the approximated execution time function of applications, to calculate how efficiently an application can utilize its processors. The two policies differ in the way they handle new arrivals.

The Static policy simply queues the new arrival if there are no processors available, while the Immediate Start Static policy attempts to start the new arrival immediately by requesting an active application to release a processor. The Dynamic policy which allocates processors according to current parallelism of applications was also presented.

7.1 Results of Experimentation

The performance of these three policies was studied using workloads created from parallel applications. We arrive at the following conclusions:

- The property of immediately starting new arrivals is crucial to the scheduler. The ISS policy performs significantly better than the STA policy under all our workloads. This performance gain increases as the system load increases. This result also shows that it is important for the scheduler to reallocate processors *both* at arrivals and at completions.

- The ordering of jobs in the arrival queue, either Smallest-Service-Demand-First or First-Come-First-Served, makes little difference to the performance of our static policies. This is supported by the fact that STA-FCFS and ISS-FCFS provide almost as good results as STA-SSDF and ISS-SSDF.

- Using application characteristics helps scheduling. The Immediate Start Static policy, which utilizes application characteristics, consistently outperformed the Dynamic policy at different system loads for a mixed workload.

- When the system load is high, it is better to use a more static policy that reallocates only at arrivals and completions. This conclusion can be drawn from our comparison of ISS and DYN using a mixed workload where ISS outperformed DYN more significantly at a higher load than at a smaller one. It is also supported by the results of the single application workload experiments. On a GRAV-workload, as the load intensity increases, the performance of ISS approaches DYN; while on a MM-workload, as the system load increases, the percentage improvement of ISS over DYN becomes greater.

- As the service demands of applications become larger, the DYN policy becomes more appropriate. When the overhead of DYN is made smaller relative to the service demands of applications, the cost of frequent reallocating processors in DYN can be justified.

- A dynamic scheduler might be inappropriate in NUMA systems. The simpler and more static approach of the ISS policy attempts to maintain data locality and cache affinity by avoiding frequent reallocations of processors, at the same time reducing overhead. As a result, ISS performs better than the more dynamic policy, DYN.

- When remote access times are made larger (as they would be in a larger system), the ISS policy continues to dominate the DYN policy for a mixed workload. This reinforces our belief that a dynamic scheduler is less suitable for NUMA systems.

- For workloads composed of applications with highly variable parallelism in different phases, the DYN policy might be the best.

7.2 Future Work Suggestions

As stated in chapter 2 of this thesis, the goal of this research is to try to gain more understanding in processor scheduling in multiprogrammed NUMA multiprocessors. We do not attempt to develop a scheduling policy that is the best one for NUMA systems, but rather address issues that should be considered when developing a scheduler for such systems.

The Immediate Start Static policy that was presented can be improved upon. Here are some possible modifications to ISS that would be interesting to study:

- The model of marginal gain and marginal loss used in ISS is too simple. A more accurate model would take data locality into account. For example, when a processor is available, and if application j has a slightly higher marginal gain than application k, it may be more efficient to allocate application k the processor if it is "closer" to application k's current processor set than to application j's current processor set.

- The Immediate Start Static policy completely ignores the parallelism structure of applications when reallocating. This results in the poor behavior of ISS when running applications of variable parallelism structure, such as Gravity. Improvements of the ISS can be made if the parallelism structures of applications are considered when making scheduling decisions.

- Our Immediate Start Static policy allocates only one processor to a new arrival. We can potentially improve the performance if the policy allocates a new arrival a number of processors up to the point where it is no longer the one with the largest marginal gain.

Another interesting extension of this work is to compare the performance of our ISS policy with other previously suggested policies. The Equipartition policy and policies that make use of other application characteristics are good candidates for comparisons.

Appendix A

Pseudo Code

This appendix contains the pseudo code of the processor scheduler and the task dispatcher that is linked to each application. Each bullet represents the occurence of an event, followed by the the sequence of actions.

A.1 Processor Scheduler

The pseudo code for the processor scheduler is given below:

- when an application J is requesting its first processor

```
if there is a processor P available
    allocate processor P to application J
    include application J into the set of active applications
else
    put application J at the end of the WaitingArrivalQueue
    % search for the least deserving application
    if policy is DYN
        search for the application K with the largest
          CurrentCpuCount in the set of active applications
        ask application K to release a processor at its
          next thread termination
```

```
else if policy is ISS
    search for the application K with the smallest
      MarginalLoss in the set of active applications
    ask application K to release a processor at its
      next thread termination
else if policy is STA
    do not make release request
```

- when an application J is requesting another processor

```
if there is a processor P available and
   the number of processors allocated to application J
   is less than its Pmax
    add processor P to the processor set of application J
```

- when an application J wants N AdditionalProcessors

```
%This event only occurs with the DYN policy.
  % first allocate using idle processors
  assign N to the WantCpuCount of application J
  while N is greater than zero
      if processor P is available
          allocate processor P to application J
          decrement WantCpuCount of application J
          decrement N
  % secondly, allocate using OkToYield processors
  while N is greater than zero
      if processor P with owner application K is OkToYield
          ask application K to release processor P
          decrement N
```

```
% finally, reallocate to maintain equipartition
while N is greater than zero
    search for the application K with the largest
      CurrentCpuCount in the set of active applications
    ask application K to release a processor at its
      next thread termination
    decrement N
```

- when an application J is WillingToYield its processor P

```
mark processor P of application J as OkToYield
```

- when an application J is Yielding its processor P

```
mark processor P of application J as not OkToYield
make processor P available
```

- when an application J terminates

```
for each processor P allocated to application J
    make processor P available
remove application J from the set of active applications
```

- when a processor P becomes available

```
%This happens when an application is asked to release a processor
%and subsequently yields a processor in DYN and ISS, or when
%an application terminates and releases all its processors.
  if WaitingArrivalQueue is non-empty
```

```
    if queue is FCFS
        remove application J from the head of the queue
    else if queue is SCDF
        search for the application J with smallest
          service demand and remove it
    allocate processor P to application J
    if policy is DYN
        decrement WantCpuCount of application J
else
    if policy is DYN
        search for the application K with the largest
          difference between its CurrentCpuCount and its
          WantCpuCount in the set of active applications
        allocate processor P to application K
    else if policy is ISS or policy is STA
        search for the application K with the largest
          MarginalGain in the set of active applications
        allocate processor P to application K
```

A.2 Thread Dispatcher

The pseudo code of the thread dispatcher that is linked with each application J is given below:

- when a thread T is created

```
put thread T at the end of the GlobalThreadQueue %FIFO queue
```

- when a thread T terminates on processor P

```
if policy is DYN
    if processor P is OkToYield and
       ( application J is asked to release a (any) processor
         or application J is asked to release processor P )
            Yield processor P
    else
        if GlobalThreadQueue is non-empty with N threads
            remove thread U from the head of the queue
            execute thread U on processor P
            notify the processor scheduler that application J
              wants N AdditionalProcessors
        else   %application J cannot utilize this processor P
            notify the processor scheduler that application J
              is WillingToYield processor P
else if policy is ISS
    if processor is OkToYield and
       application J is asked to release a (any) processor
            Yield processor P
    else
        if GlobalThreadQueue is non-empty
```

```
            remove thread U from the head of the queue
            execute thread U on processor P
else if policy is STA
    if GlobalThreadQueue is non-empty
        remove thread U from the head of the queue
        execute thread U on processor P
```

Bibliography

[And91] Thomas E. Anderson. *Operating System Support for High Performance Multiprocessing.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991.

[BH86] Josh Barnes and Piet Hut. A Hierarchical O(n log n) Force-Calculation Algorithm. *Nature*, 324(4):446–449, December 1986.

[BLL88] Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[Dow88] Lawrence W. Dowdy. On the Partitioning of Multiprocessor Systems. Technical report, Department of Computer Science, Vanderbilt University, March 1988.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. *Proceedings of ACM SIGMETRICS*, pages 120–132, May 1991.

[LV90] Scott T. Leutenegger and Mary K. Vernon. The performance of Multiprogrammed Multiprocessor Scheduling policies. *Proceedings of ACM SIGMETRICS*, pages 226–236, May 1990.

[MEB88] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in Multiprogrammed Parallel Systems. *Proceedings of ACM SIGMETRICS*, pages 104–113, May 1988.

[MEB91] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Characterization of Programs for Scheduling in Multiprogrammed parallel systems. *Performance Evaluation 13*, pages 109–130, October 1991.

[MVZ91] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors. Technical report, Department of Computer Science and Engineering, University of Washington, February 1991.

[Sev89] Kenneth C. Sevcik. Characterizations of Parallelism in Applications and Their Use in Scheduling. *Proceedings of ACM SIGMETRICS*, pages 171–180, May 1989.

[Sev92] Kenneth C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Multiprocessors. Technical report, Computer Science Research Institute, University of Toronto, 1992. To appear in *Performance Evaluation*, 1993.

[SL93] Mark S. Squillante and Edward D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[Squ90] Mark S. Squillante. Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation. PhD thesis, Department of Computer Science and Engineering, University of Washington, October 1990.

[Sri91] Nandini Srikantiah. Processor Scheduling in Hierarchical NUMA Multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, 1991.

[SVW+92] Michael Stumm, Zvonko Vranesic, Ron White, Ronald Unrau, and Keith Farkas. Experiences with the Hector Multiprocessor. Technical report, Computer Science Research Institute, University of Toronto, 1992.

[TG89] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proceedings of 12th ACM SOSP*, pages 159–166, December 1989.

[Unr93] Ronald C. Unrau. *Scalable Memory Management Through Hierachical Symmetric Multiprocessing*. PhD thesis, Department of Electrical and Computer Engineering, Univeristy of Toronto, January 1993.

[VZ91] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. *Proceedings of 13th ACM SOSP*, pages 26–40, October 1991.

[ZB91] Songnian Zhou and Timothy Brecht. Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors. *Proceedings of ACM SIGMETRICS*, pages 133–142, May 1991.

[ZM91] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. *Proceedings of ACM SIGMETRICS*, pages 214–225, May 1991.