Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach

Anoop Gupta, **Andrew Tucker**, and **Luis Stevens**
Computer Systems Laboratory
Stanford University, CA 94305

Revised July 1991

Abstract

We present the design, implementation, and performance of a novel approach for effectively utilizing shared-memory multiprocessors in the presence of multiprogramming. Our approach offers high performance by combining the techniques of *process control* and *processor partitioning*. The process control technique is based on the principle that to maximize performance, a parallel application must dynamically match the number of runnable processes associated with it to the effective number of processors available to it. This avoids the problems arising from oblivious preemption of processes and it allows an application to work at a better operating point on its speedup versus processors curve. Processor partitioning is necessary for dealing with realistic multiprogramming environments, where both process controlled and non-controlled applications may be present. It also helps improve the cache performance of applications and removes the bottleneck associated with a single centralized scheduler.

Preliminary results from an implementation of the process control approach, with a user-level server, were presented in a previous paper [24]. In this paper, we extend the process control approach to work with processor partitioning and fully integrate the approach with the operating system kernel. This also allows us to address a limitation in our earlier implementation wherein a close correspondence between runnable processes and the available processors was not maintained in the presence of I/O. The paper presents the design decisions and the rationale for the current implementation, along with extensive results from executions on a high-performance Silicon Graphics 4D/340 multiprocessor.

1 Introduction

Applications written for parallel computers often assume that they will have sole use of the machine with all processors dedicated to them. On a multiprogrammed machine, where multiple users and applications may be active simultaneously, this is frequently not the case and each processor may be shared among multiple processes. In such environments, we have observed that the throughput of the system can degrade substantially when the total number of active processes in the system does not match the total number of processors.

Figure 1 shows the impact on performance when a parallel application's processes must contend for processors. The data is gathered from a Silicon Graphics 4D/340 multiprocessor with 4 processors. The graph shows the finish time for two parallel applications, MP3D and LocusRoute, when the two are started at the same time and as the number of processes is varied.[1] The figure shows that the performance of both applications worsens considerably when the number of processes in each

[1]MP3D [16] is a particle-based wind tunnel simulator, and LocusRoute [18] is a VLSI standard-cell router. Each application breaks its problem into a number of tasks, which are scheduled onto the processes executing that application. These applications and results are discussed in detail in Sections 3 and 4.

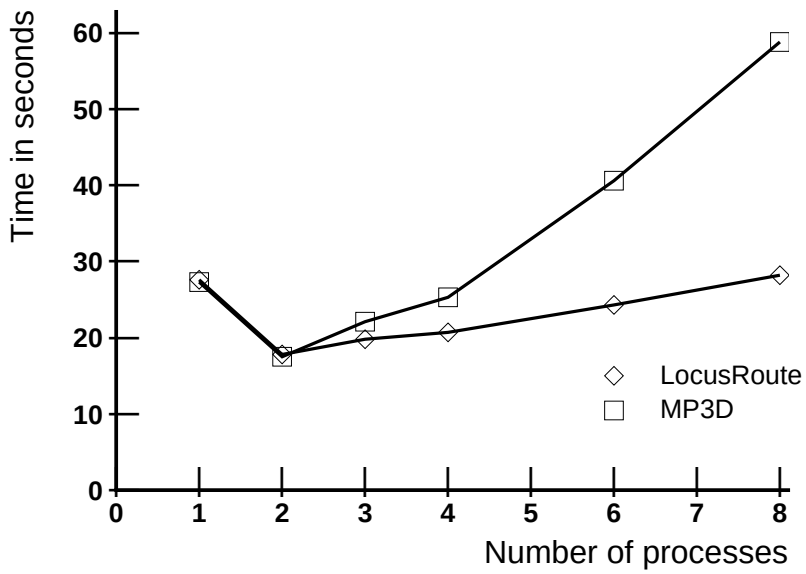

Figure 1: Finish time for MP3D and LocusRoute applications when started simultaneously and as the number of processes is varied.

application exceeds two, and thus the total number of processes in the system exceeds the number of processors. Furthermore, the larger the number of processes the worse the performance gets.

The performance degradation from multiprogramming, as seen above, can occur due to several reasons. First, there is the overhead of context switching between processes. Aside from the problem of corrupted caches (discussed below), a context-switch involves a number of system-specific operations, such as saving and restoring registers and switching address spaces, that do no real work. Second, many parallel applications use synchronization that requires busy-waiting on a variable. If the process that will set the variable is preempted, other processes may end-up wasting processor time waiting for that variable to be set. Finally, frequent context switching can indirectly affect processor cache behavior. When a context switch is performed, the preempted process may be rescheduled onto another processor, without the cache data that had been loaded into the cache of the original processor. Even if the process is rescheduled onto the same processor, intervening processes may have replaced needed cache data.

To address the above issues, several solutions have been proposed. To address the synchronization problem, researchers have proposed the use of gang scheduling strategies [17, 6, 7] that ensure that all processes belonging to an application execute at the same time. As another solution, Zahorjan et al. [26] and Edler et al. [6] have proposed using intelligent schedulers that avoid preempting processes while they are inside a critical section. Another technique that has been proposed is to use blocking synchronization primitives instead of busy-waiting primitives [17, 10]. To address the cache hit-rate problem, scheduling strategies that use cache affinity (the amount of relevant data a process has in some processor's cache) have been proposed [23, 10]. Another approach has been to use partitioned scheduling [2], where processes from an application are always scheduled onto the same subset of processors, ensuring that the data shared between the processes is likely to be found in the cache. Unfortunately, these techniques have had only limited success [10, 25].

In contrast to the above approaches, which focus almost exclusively on the operating system scheduler, we believe that a synergistic approach that involves both the application and the operating system can offer higher performance. In this paper, we focus on one such approach proposed in [24]. It requires *process control* from the applications and *processor partitioning* plus some interface support from the operating system.

The process control technique is based on the principle that to maximize performance, a parallel application must dynamically control its number of runnable processes to match the effective number of processors available to it. In a multiprogramming environment, this adjustment of processes must be dynamic because other applications are constantly entering and leaving the system, and consequently the number of processors available to an application is constantly changing. By dynamically maintaining a match, context switches are largely eliminated and good cache and synchronization behavior can be ensured. The process control approach is most easily applied to the wide variety of parallel applications that are written using the task-queue or threads style [1, 3, 4, 5, 8, 11, 12], where user-level tasks (threads) are scheduled onto a number of kernel-scheduled server processes. In such an environment, the number of server processes can be safely changed without affecting the running of user-level tasks. Another important implication is that process control can be made totally transparent to the applications programmer by embedding it completely in the runtime system of a programming language or threads package.

In the processor partitioning technique, a policy module in the operating system continuously monitors the system load and dynamically (based on need, fairness, and priority) divides up the processors among the applications needing service. Scheduling of processes within a partition is handled at a lower level independently of the policy module. Processor partitioning is motivated by our need to handle realistic multiprogramming environments where we expect a mixture of applications—there may be some parallel applications that control their processes, there may be others that don't, and there may be single-process applications like compilers, editors, and network daemons. The problem with using process control in such environments, without processor partitioning, is that applications that do not control their processes may get an unfair share of the processors. The processor partitioning technique further allows a closer binding to be established between an application and the processors executing it, thus helping to improve cache performance. The technique also helps in removing the bottleneck associated with a single centralized scheduler in highly parallel machines.

Preliminary results from an implementation of the process control approach were presented in our previous paper [24]. This implementation was done on an Encore Multimax running the UMAX operating system and supported process control through a user-level server. The Encore implementation, however, did not support processor partitioning and was not fully integrated with the kernel. In this paper, we extend the process control approach to work together with processor partitioning. We also address a limitation of our earlier implementation that made it difficult to maintain a close correspondence between the number of runnable processes and the number of processors assigned to an application. This paper presents the design decisions and the rationale for the current implementation, along with extensive results from executions on a high-performance Silicon Graphics 4D/340 multiprocessor.

Our experiments show that the process control approach performs significantly better than the regular priority-based scheduling algorithm used on the 4D/340 multiprocessor. The better performance holds true under a variety of application loads. In our earlier experiments with process control on the Encore [24], most of the performance benefits had come from processes not being preempted inside critical sections. In our current system, the role of preemption within critical sections is much smaller, since our applications now use blocking rather than busy-waiting synchronization primitives [10]. Most of the performance benefits, instead, come from the better cache behavior and better load balancing achieved when an application is working with fewer processes (due to process control) in a more stable processing environment (partly due to process control and partly due to processor partitioning).

The paper is structured as follows. In Section 2, we discuss design and implementation issues for the process control approach. Section 3 describes our experimental environment, and Section 4 examines the performance of the process control approach under various load conditions and as compared to other scheduling methods. Related work is presented in Section 5 and we conclude in Section 6.

2 Design and Implementation Issues

At a basic level, the concept of process control and processor partitioning is fairly simple. When an application is executed, processors are assigned to it. The application is informed of the number of processors that have been assigned to it, and other applications from which the processors were taken are similarly informed of the loss of processors. All applications concerned (if they support process control) suspend or resume processes as appropriate so that the number of runnable processes matches the number of processors assigned to them.

While the process control approach is simple and intuitive at a conceptual level, its effective implementation requires addressing a large number of subtle issues. In this section we discuss some of these issues and describe the solutions we have adopted. We first discuss how applications may dynamically change the number of processes they are using. We then consider the problems associated with the implementation of process control in these applications, and the interaction necessary between the application and the operating system. Finally, we discuss the policy decisions involved when partitioning processors among the applications, and techniques to ensure reasonable response time for interactive applications.

2.1 Programming Models Supported and the Safe Suspension Issue

Since the process control approach requires dynamic adjustment of the number of runnable processes in an application, a fundamental question that arises is what programming models allow this sort of dynamic adjustment. For example, a suitable programming model should be able to effectively utilize newly created processes, or previously suspended processes that are resumed at some point in time. Likewise, to support process control transparently to the applications programmer, the programming model should make it easy to determine when a process can be *safely suspended*, that is, suspended without potential starvation, loss of data, or significant loss of efficiency.

Although the problem of determining safe suspension points is intractable for arbitrary parallel applications, the problem is fortunately simple for the large class of applications that use a *task-queue* model. In this model, applications are broken up into a number of tasks, and server processes repetitively select tasks from a queue and execute them. In these applications, a server process can safely suspend itself after it has finished executing a task and before it has selected another task to execute. A server process can also safely suspend itself before it has finished executing a task, as long as it places that task back on the task queue and it makes sure that the task is not inside a spin-lock controlled critical section. Furthermore, resumed or newly created processes can do useful work immediately by simply picking tasks from the task queue and executing them.

Since task-queue based models are widely used to implement parallel applications on shared-memory architectures, the applicability of process control is quite large. For example, one can find several programming languages with runtime systems based on the task-queue model [3, 8, 11, 12], and consequently all programs written in these languages follow the model. Similarly, all applications written using threads packages [1, 4, 5] follow this model, as do many independently written applications [9, 18, 22]. Finally, the process control model can also be made to work for other programming paradigms that do not follow the task-queue model, though this requires extra support from the compiler or the programmer to identify safe suspension points.

For our prototype implementation, we have added process control to the runtime system of COOL [3], a task-queue based object-oriented programming language. To ensure safe suspension, we suspend a server process only when its task has finished or when its task blocks on a blocking synchronization primitive.[2] Processes can be created or resumed asynchronously without restriction.

[2]Note that the blocking synchronization primitives are at the level of the programming language and not at the kernel level. That is, when a task blocks, that task is put on a wait queue, and the server process picks another task from the task

2.2 The Mechanics of Process Control and the Distribution of Responsibility

Although applications using the task-queue or threads model allow the use of dynamic process control, the question still remains of how this control is to be implemented and how the responsibility of process control is to be distributed between the operating system and the application. In particular, the following issues that need to be addressed:

- What system-level events should the applications be informed about to support process control?
- How are these events to be communicated to the application, given that they may occur asynchronously with respect to the application?
- Given the goals of process control, how should applications safely and efficiently respond to relevant events?
- How can the above mechanisms be implemented without incurring significant overheads and without radically changing currently prevalent operating system environments?

In this subsection we discuss the various tradeoffs that are involved in resolving these issues, and present the specific solutions that we have adopted in our prototype implementation. Note that in the following discussion when we refer to an "application" implementing some facet of process control, we really mean the runtime system of the programming language or the threads package used by the application. In general, we expect all aspects of process control to be totally transparent to the applications programmer.

2.2.1 Identifying Relevant Events

In order to make use of process control, applications must be informed when events occur that may affect the number of processes they should be using. These events include kernel-controlled changes in the number of processors assigned to an application, as well as suspension and resumption of application processes on kernel semaphores. The latter includes blockages due to I/O transactions. As we will discuss in the next subsection, applications respond to these events by suspending and resuming processes as appropriate. First, though, since processing of events incurs costs, we need to determine which of the above events are truly relevant, i.e., require some action to be taken by the application to ensure good performance, and which are only marginally relevant and may be filtered away.

In general, the relevance of events depends on (i) the nature of the application (some applications may be better able to tolerate excess processes than others), (ii) the nature of the parallel machine (some machines may have higher penalties for poor cache hit rates than others), and (iii) the duration for which the mismatch between the number of processes and processors is expected to last. We consider events that cause a longer term mismatch to be more relevant than those that cause a short term mismatch. The tradeoff here is that if a process blocks only for a short duration, for example when a page fault is serviced from the buffer cache in main memory, it may be better to let a processor remain idle than to resume another process in its place. The resumed process will probably suffer from poor processor utilization since its working set will not be in the cache, and it may also destroy the data cached by the blocked process. On the other hand, if the process blocks for a longer duration, the idle processor will result in lower performance than if a new process was created.

Even when the relevance of events is known, there still remains the question of where to filter out the events. The events may be filtered by the kernel before they are sent to the application, or they

queue. It is not the case that the server process is blocked on some kernel queue and another process or kernel-level thread is run on that processor.

may be filtered by the application before any action is taken on those events. Significant flexibility is obtained by having the kernel communicate *all* events to the application, and having the application filter events. In this way, even if different applications (runtime systems of different programming languages) find different types of events relevant, they can all share the same kernel interface. The disadvantage of filtering at the application level is higher overhead. For example, if the application is informed about the asynchronous events via UNIX signals, the excess cost of sending and processing signals for irrelevant events may be non-trivial. The most efficient approach is to have the kernel filter system-level events, and only communicate those events that will be acted upon. This, however, forces the application to rely on the filtering provided by the kernel.

The approach we have taken in our prototype implementation is as follows. The kernel always informs an application when the number of processors assigned to it changes, since these are expected to be long duration events. (As will be discussed in Section 2.3, changes are expected to happen primarily when new parallel applications enter the system or old ones finish.) However, for blockages on kernel semaphores, the kernel decides whether or not to communicate an event based on the duration for which that particular type of semaphore is expected to block. Events corresponding to short duration semaphores, such as reads from the disk buffer cache in memory, are filtered out by the kernel. Events corresponding to longer duration semaphores, such as actual disk accesses, are communicated to the application. Sometimes the duration for which a process blocks on a semaphore may be highly variable, for example, on semaphores associated with a UNIX pipe. In such cases, it would be best to send an event to the application only after some additional state regarding the semaphore has been checked. In our current implementation, however, we do not exploit this optimization, and the kernel communicates such events anyway. The application then does further filtering, particularly in the case where two events that are close together cancel each other's effects.

2.2.2 Communicating Events to Applications

Another important issue is how an application should be informed about relevant events; that is, how it is informed when a processor is taken away from it, or an extra processor is made available, or a process blocks on I/O.[3] The two factors that influence the choice of the mechanisms used are (i) response time, that is how quickly the application is made aware of relevant system-level events, and (ii) communication overhead, that is the computational cost of communicating the events. There are two main choices for mechanisms; the kernel could asynchronously inform applications when a relevant event occurs, or the applications could synchronously poll the kernel checking for relevant events.

The role of the kernel is obvious in providing quick response time to events, since the kernel knows exactly when processor assignments are changed, or when a process blocks on some kernel semaphore. It is not possible for application-based polling to be competitive in this regard [24]. The primary drawback of the kernel asynchronously signaling the application is the high overhead associated with signal handling. In contrast, polling can be made quite cheap by having a shared data area between the application and the operating system where relevant events are recorded. As a result, it appears that when response time is critical, kernel-based signaling would be the mechanism of choice, but when some slack can be tolerated, polling may be best to reduce overheads.

In determining the type of communication necessary for process control, we consider the conditions under which it is important that an application receive and respond to events quickly. Relevant events can be broken down into two types, those that indicate that the application should decrease its number of processes (suspend a process) and those that indicate that it should increase its number of processes (resume or create a process). In the former case, the cost of a delay is frequently small. Although

[3]Note that these events (aside from blocking on I/O) are asynchronous to the execution of the application. For example, the kernel may take a processor away from an application any time it considers appropriate, without regard to the current execution state of the application.

it will result in excess processes for a brief time, the processes will simply be scheduled onto the processors in a round-robin manner, and will continue to do useful work. This is especially true if blocking synchronization primitives (with a small spin in front) rather than spin-based synchronization primitives are used [10]. If the number of runnable processes is *less* than the number of processors, however, one or more processors will sit idle. Although this may be reasonable for very short periods to avoid excess context switching, any long idle time will result in performance loss.

Also, regardless of how quickly an event is communicated to the application, the application may have to delay in responding to it. This is particularly true of "suspend" events. The application must wait until a process reaches a safe suspension point before responding to the event by suspending a process. "Resume" events, however, can be acted upon immediately, by creating a new process or resuming a previously suspended one.

Based on the above considerations, in our prototype implementation we chose to have the kernel signal an application when an event occurs that indicates the application should increase the number of processes it is using. The application can then immediately create a new process or resume a previously suspended one, as appropriate. When the application should decrease the number of processes it is using, however, we do not send a signal. Instead, each process polls the kernel whenever it reaches a safe suspension point. Since polling in our implementation is much cheaper than sending and receiving signals, this reduces the cost of communication by almost one-half without affecting the response time of the application to such events.

At a more detailed level, the interaction between the kernel and the application takes place as follows. To communicate process control related information, the kernel and the application use a shared counter (one per application) that resides in the kernel address space. This integer counter is read-only for the application and reflects changes in the number of processes the application should have active (as per the process control philosophy). Whenever the kernel observes a relevant event, it suitably updates this counter. If the event was one that increased the value of the counter, indicating that the number of processes should be increased, the kernel sends a signal to the application indicating that action needs to be taken. On receiving the signal, or on reaching a safe suspension point, the application reads the kernel counter, and takes whatever action is necessary. The application bases its action on the change in the value of the kernel counter since it was last read by an application process. The application process simply compares the current value of the counter to the old value that had been recorded earlier, suspending processes if the current value is smaller than the old value and resuming or creating processes if the current value is larger.[4] It is not necessary for the application to modify the kernel counter, which avoids protection problems for the operating system.

The signal is sent by setting a bit in the process control block of one of the application's running processes. Since a process normally responds to a signal only when it is returning from a system call or context switch, we also send a special interrupt to the processor on which the chosen process is running, thus ensuring quick attention to that signal. There are several reasons why we use a shared counter instead of directly encoding event information in signals. First, the counter provides efficient polling of the necessary kernel information, allowing the previously described optimization of avoiding signals when the counter value is decreased. Second, the shared counter lets any process reading the counter, not only the one to which the signal was sent, respond to process control events. This flexibility helps improve response time. For example, consider the case when between the time the kernel decides to send a signal to a process and the time that it actually sends it, the destined process blocks. While the destined process can not respond to the signal until it unblocks, under our approach another process can take the necessary action. Third, the shared counter helps combine multiple process control events, reducing overhead. For example, if multiple processes need to be resumed, this can be determined simply by reading the kernel counter once and computing the difference between the old and the new values. Finally, we avoided using signals to encode information because under UNIX information

[4]For the moment, we assume that applications can make use of any created or resumed processes. In the next subsection we will consider the problem of applications with limited parallelism.

may be lost when multiple signals are sent to the same process. As a result, in our implementation, the role of the signals is solely advisory. They are there to improve responsiveness, but they do not affect the correctness of the process control algorithm.[5]

Let us now examine the effects on performance of the delays in our system. One cost is that a newly allocated processor may remain idle because the application has not yet resumed or created a process. We expect the waste of processor cycles due to this to be small. This is because (as discussed earlier) our implementation ensures that the signal handler is invoked quickly by sending an interrupt immediately after sending the signal in such cases. A second cost is that when a processor is taken away, until one of the application processes reaches a safe suspension point, we will have more processes than processors. If this duration is small, say because the tasks are reasonably small, then the excess processes may not hurt performance as all processors will probably continue to do useful work. If the duration is long, however, the performance may suffer. One problem is that if a process is preempted while it is inside a spinlock-controlled critical section, the other processes may waste time spinning idly. In our implementation, we minimize such idle time by always using blocking synchronization primitives with a small amount of spin time before blocking. By making the spin time before blocking equal to the context switching time, we can ensure good performance for both short and long critical sections [10]. However, there still remains the disadvantage of worse cache behavior when there are excess processes. The negative impact of this factor is reduced in our implementation due to processor partitioning — since all processes within a partition are from the same application, they often have a significant amount of shared data.

We now examine some benefits of having a small delay between process control events and the corresponding actions. As was stated earlier, the duration for which a process will block in the kernel is frequently unpredictable. Because of the unpredictability, process control signals are sent to applications for many short-duration blockages. In such cases, instead of immediately resuming a process in response to an initial blockage and then suspending one when the blocked process soon wakes up, it would be better if no action were taken. This would minimize the overheads of resumption and suspension and those of cache corruption. The implicit delays that are present in our system help performance in these cases. As an example, recall than when a process blocks in the kernel, the shared counter is incremented and a signal is sent to one of the other processes of that application to take action. If the blocked process resumes quickly, then by the time that the signal handler is run, the kernel may have already decremented the counter again. As a result, the signal handler will find the value of the kernel counter unchanged, and as desired, no action will be taken. Similarly, consider the case where an application has been asked to suspend a process. Now the application can not respond to this command until one of the processes reaches a safe suspension point. If during this time, the kernel counter is incremented (say because another process blocks on I/O or because another processor has become available), then the suspension command will be nullified.

2.3 The Policy of Processor Partitioning

To make the process control technique usable in realistic system environments, it is important to have some way of dividing, or *partitioning*, the processors in the system among the active applications. Processor partitioning allows process-controlled applications to be separated from non-controlled ones, avoiding problems with fair distribution of processing resources. Otherwise, the non-controlled applications may get an unfairly large fraction of the processing resources. Another benefit is that since processes running on any given processor are likely to be from the same parallel application (with

[5] A complication arises when the only running process of an application blocks on a kernel semaphore. As described so far, the kernel would find no other process belonging to that application that it can inform to create or resume a new process. For the duration of the blockage, all processes of the application will be blocked and any processors assigned to the application will be idle. To avoid this, the kernel will temporarily wake up the blocked process and send a signal to it indicating (via the shared counter) that the number of processes should be increased. After creating a new process or resuming a previously suspended one (if appropriate), the signal-handling process returns to its former blocked state.

common code and shared data), it helps to increase the cache hit rate, thus increasing processor utilization. Finally, processor partitioning helps avoid the bottleneck associated with a centralized scheduler with a single run-queue.

As just stated, the processor partitioning approach divides the processors in a multiprocessor among the applications needing service. This is to be contrasted with most scheduling strategies that time multiplex the processors among the applications. The basic construct in the processor partitioning approach is that of the *processor set*. Each processor set consists of a local run-queue and other related data structures. A high-level *policy module* is responsible for assigning both resources (processors) and tasks (application processes) to it. Each processor executes processes that have been assigned to its processor set in a regular time-sliced manner [2], though this can be changed on a per-partition basis. It is possible to have a processor set with no processors assigned to it, in which case the processes assigned to it will simply be waiting in the run queue.

The policy module plays a critical role in making processor partitioning effective. For example, it must decide when to create or delete processor sets, how to distribute the processors among the processor sets, and how to assign applications to processor sets. Furthermore, it must make these decisions in view of higher level goals. These in our case are to provide: (i) fast response time for high priority interactive applications, and (ii) high throughput for compute-intensive parallel applications. In the following paragraphs we discuss issues that arise in the design of processor sets and the policy module. Since the design space is very large, we use the details of our prototype implementation to make the discussions concrete. Much experimentation, however, remains to be done in this area.

We first explore the organization of processor sets, that is, how to associate applications with processor sets. One straightforward solution is to create a separate processor set for each application. This approach has two disadvantages. First, the number of processor sets may become very large, thus greatly increasing the complexity of processor allocation decisions. (Although the number of parallel applications running at any one time in a typical system may be small, the total number of applications is often much larger, especially if one includes serial applications such as compilers and editors and system processes such as network daemons.) Second, many of these applications may not be able to effectively use even a single processor for the duration it is allocated to them. An example would be a compiler process that performs a lot of I/O. As a result, making effective use of processors under this approach requires that processors be frequently moved between processor sets, which is both inefficient and makes the processor allocation algorithm difficult.

An alternative strategy is to create one processor set per *class* of applications. For example, we can have one processor set for all process-controlled applications, another for all non-process-controlled parallel applications, another for compute-intensive serial applications, another for OS daemons, and so on. As desired, such a strategy avoids the problem of non-controlled applications grabbing an unfair share of the processing resources. It also has the advantage that since processors are allocated in larger clumps, there is greater sharing of resources. For example, if one compiler process is not using a processor due to an I/O blockage, another compiler process in the same processor set could use it during that time. However, this strategy has the problems that the set of application classes is quite ad-hoc, and that processor allocation to these aggregate processor sets, with multiple applications each, is an extremely difficult task.

The approach that we have taken in our prototype implementation is in between the above two. Instead of allocating a processor set for each application in the system, we only allocate processor sets for compute-intensive and/or parallel applications. All other applications (e.g., network daemons, editors, etc) execute within a perpetual *default* processor set. When an application first begins execution, it always starts in the default processor set. Applications that establish themselves as being compute-intensive or parallel are then migrated to separate processor sets. A processor set is deleted when all processes assigned to it have completed. The data structures, however, are saved and reused. If the number of processor sets begins to exceed the number of processors in the system, we assign multiple applications to the same processor set. Process control signals caused by changes in the

number of processors allocated to a processor set are then sent to all applications in the processor set, so all applications will have the same number of processes. An alternative would have been to continue to assign each parallel application a separate processor set (thus, some applications may have no processors assigned to them), letting the processor allocation algorithm ensure that all applications get a fair share of processors over some longer interval of time. This is also a reasonable option, but would have degraded response time.

The next major policy decision is the assignment of processors to processor sets. Our main goal is to fairly allocate processors among applications, but the existence of serial applications in the default processor set makes the problem more complicated. Our assumption in designing the policy module was that high loads in the default processor set are often transient, and that handling these loads with high priority is important to providing good response time. As a result, in our implementation, the policy module first allocates processors to the default processor set. If there is even a single runnable process in the default processor set, at least one processor is allocated to it. If the number of runnable processes is greater, a larger number of processors is allocated, up to a maximum as determined by the number of other "active" processor sets. The remaining processors are then distributed evenly among the other processor sets, to the extent that each can use them. If the processors cannot be divided evenly (for example, if 4 processors are to be divided among 3 processor sets), processor sets that are assigned fewer processors in one allocation interval are assigned more processors in the next interval. Thus, over the long run, fair allocation is preserved. Note that this discussion assumes that process-controlled parallel applications can efficiently use as many processors as they are allocated. Since this assumption is not true in general, we are in the process of implementing a system call that will let applications communicate their resource requirements to the kernel, enabling the kernel to take away processors when an application cannot use them.

Another policy question is the frequency with which processor reallocations must be done. More frequent processor reallocations will be better at adapting to varying application loads and at preserving fairness, but will add overhead as the costs of processor migration are incurred more often. In response to this, our policy module performs processor reallocations under two circumstances. First, whenever a new processor set is created or an old one is deleted (that is, when a new parallel application enters the system or an old one finishes) a processor reallocation is done. This means that reallocation will always immediately adapt to application load changes. Second, to allow for adjustments even when applications are not entering or leaving the system, processor reallocations are done periodically. (This interval is currently 300ms.) In all reallocations, every effort is made to ensure that processors are not gratuitously moved between processor sets, as this destroys the data accumulated in processor caches by the applications.

As stated earlier, one of our goals is to provide fast response time to interactive applications. In adhering to this goal, when the load in the default processor set is very low and sporadic, the policy as described above gives rise to the following tradeoff. If we keep one processor permanently assigned to the default processor set, then we guarantee good response time, but this processor will mostly be idle. It could probably have been better used by some other processor set. (This is especially important for machines with a small number of processors.) However, if we allow the policy module to take away all processors from the default processor set (for example, this would be done by the policy module if at the time of reallocation the default processor set had no runnable processes), then consider the situation of an interactive application that arrives soon after the last reallocation. It may have to wait until the next reallocation interval (up to 300ms away) before getting any service, resulting in very poor response time.

The solution we have adopted works as follows. If at the time the reallocation is done the default processor set has no runnable processes, all processors are taken away from it. However, every so often (100ms in the current implementation), the kernel checks if any runnable processes have arrived in the default processor set. If so, it does a "partial" reallocation, assigning one processor to the default processor set. This ensures that the response time is reasonable. (Note that the application from which

the processor is taken away is informed of this, so that it can reduce the number of processes it is using.) When processors that are assigned to the default processor set become idle, they wait for a short interval of time (about 10–20ms) for more work to arrive.[6] If they find no work, they return back to one of the other processor sets and become usefully employed.

There is another practical complication that we need to address. In many machines, it is sometimes necessary that a process run on some specific processor. For example, on a Silicon Graphics 4D/340 the network driver does not protect global data, and thus any processes wishing to do network I/O are forced to run on processor 1. We address such situations as follows. When a process wants to run on a specific processor, the kernel places it on a special global queue with a note saying that it must be run on the requested processor. (This is the *must-run* queue in the Silicon Graphics IRIX operating system.) All processors check this global queue before their normal processor-set run queue, and execute relevant processes from this queue first. When such special operations are completed, the process is returned to its normal queue. Certain high-priority system processes may also be placed on the global queue for optimal response time. Since processes on the global queue run only for a short time on the processors that execute them, they should have little effect on the performance of applications assigned to processor sets.

2.4 A Process Control Example

We conclude this section by using a simple example to demonstrate process control and processor partitioning in action. Consider a four-processor parallel machine, with two CPU-bound parallel applications, A and B. Each of the applications may be adjusted via process control to use any number of processes, but initially starts with one process.

When application A begins, it is the only application running in the system. It begins running in the default processor set, which has been assigned all 4 processors. The application then notifies the system that it is a parallel application. The processor partitioning policy module then assigns the application to its own processor set, and allocates 4 processors to that processor set. When this allocation has completed, the kernel sets the counter shared between the kernel and application A to 4 and sends a signal to A's process. When the signal is received, the process checks the shared counter, and creates 3 new processes. The process returns to performing work, and we now have 4 processes running on 4 processors, and a stable system.

Now, consider what happens when application B begins. Like A, B begins executing in the default processor set. The policy moves one processor to the default processor set to execute the application's initial process. This results in taking a processor from A. The kernel decrements A's shared counter by 1 (to 3). The first process of A's to reach a safe suspension point will check this counter and suspend itself. Application B notifies the system that it is a parallel application, and is assigned its own processor set. The policy module allocates B the processor previously allocated to the default processor set and one of the processors previously allocated to A. After this reallocation occurs, both A's shared counter and B's shared counter are set to 2 and a signal is sent to B's process. When B receives the signal, it checks the shared counter and creates a new process to take advantage of the extra processor. No signal is sent to A, but the first process of A's to reach a safe suspension point will suspend itself (recall that processes always check the shared counter at safe suspension points). After this occurs, each application will have 2 processes executing on 2 processors, and we will have a stable system.

The next case we will consider is the effect of an I/O operation. Assume a process of application A does a read. The process is suspended pending completion of the read. When the process is

[6]The reason for waiting for a short interval is the following. Consider a situation where an interactive process is doing a lot of disk I/O. If every time it blocked on I/O we took its processor away, then it would not get the processor back for another 100ms or so, even though the I/O may have completed much earlier. Waiting for a short period helps this situation considerably.

suspended, the kernel increments A's shared counter (to 3) and sends a signal to one of the other processes of A. When that process receives the signal, it checks the counter and resumes one of its previously suspended processes. While the I/O operation is being performed, the extra process serves to avoid letting the processor that had been running the I/O-initiating process sit idle. When the I/O operation completes, the process that was performing is resumed. Before exiting the kernel, though, the kernel decrements A's shared counter (to 2). The process that initiated I/O returns to executing code. For a brief time, 3 processes will be running on 2 processors. The first process of A's to reach a safe suspension point will suspend itself. When that process is suspended, the applications will again be at an equilibrium.

Finally, consider what occurs when B completes before A. B's processor set is destroyed once application B exits and the two processors assigned to it are released and reassigned to application A. A's shared counter is set to 4 and a signal is sent to one of its processes. That process receives the signal and resumes 2 previously suspended processes. At the end of this, A will have 4 processes executing on 4 processors.

3 The Experimental Environment and Benchmark Applications

The process control approach has currently been implemented on a Silicon Graphics PowerStation 4D/340 multiprocessor. The system consists of four 33MHz MIPS R3000/R3010 processors on a shared bus and provides peak computing power of about 100 MIPS and 35 MFLOPS. Each processor has a 64 Kbyte instruction cache, a 64 Kbyte first-level data cache and a 256 Kbyte second-level data cache. The cache line size is 16 bytes, while the fetch size is 64 bytes. A first-level hit costs 1 clock cycle, a second-level hit costs about 14 clock cycles, and a second-level miss costs about 50 clock cycles. As a result of the large miss penalties, it is very important to have high cache-hit rates to get high processor utilizations.

The operating system running on the SGI 4D/340 is IRIX, a multithreaded version of UNIX System V with added functionality for supporting parallel applications. To support process control, we have made a number of changes to IRIX. As described in the previous section, these changes consist of support for processor partitioning, the implementation of a policy module to govern allocation of processors, and an interface between parallel applications and the kernel that provides communication in both directions regarding the running environment of the applications.

The performance of the basic operating system primitives used in processor partitioning and process control is shown in Table 1. A kernel trap refers to a simple system call, such as gettimeofday(), after which the processor continues running the same process. A context switch is a full processor context switch, placing one process on the run queue and scheduling another. Reallocating processors involves the full reallocation of processors to processor sets in the policy module. The time shown is for the case when a single processor is moved from one processor set to another. This time is expected to grow linearly with the number of processor sets in the system. Process assignment moves a process from one processor set to another. A processor set creation initializes a new processor set assuming that an empty processor-set data structure exists. Although the processor partitioning implementation has not been tuned, the time taken by these operations is small compared to the indirect effects of scheduling policy on performance. For example, our measurements show that if a process is preempted and then rescheduled back after its cache data has been replaced, the time to fill the second-level cache is about 10000 microseconds.

There exist a number of programming languages and threads packages that use the task-queue model and thus may easily be adapted to use process control. The language used in our prototype implementation is COOL [3] (Concurrent Object-Oriented Language). COOL provides basic tasking and synchronization mechanisms and an overall object-oriented framework for parallel programming. We chose COOL primarily because we are familiar with the internals of its runtime system and there

Table 1: Execution Times of Basic Operations (in microseconds)

Kernel Trap	Context Switch	Reallocate Processors	Process Assign	Processor Set Create
29	51	39	65	46

are several large parallel applications that have been implemented using it. We added process control to the COOL runtime system in such a way that application programs were unaffected and simply needed to be relinked with the new runtime system of the language. The modifications to the runtime system consisted of adding a few function calls to appropriate parts of the startup and dispatch code. These were straightforward and took little programming time.

For our experiments with process control, we primarily used workloads composed of three COOL applications and a parallel *make* program. The applications we used are LocusRoute, MP3D, and Sparse. All three applications are reasonably complex engineering programs representing the fields of VLSI design, aeronautical simulation, and numerical analysis (details can be found in the SPLASH report [21]).

LocusRoute [18] is a global router for VLSI standard cells. It uses an iterative algorithm to route wires with the goal of minimizing congestion. The major data objects consist of a queue of wires to be routed and a shared cost array. Each task repetitively selects a wire to be routed, explores alternate routes, and places the wire in the best route. For our experiments we ran LocusRoute with a 904-wire circuit, performing 20 routing iterations. MP3D [16] is a particle-based simulator used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The primary data objects in MP3D are particles (representing air molecules) and space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of particles over a sequence of time steps and gathering appropriate statistics. For our experiments we ran MP3D with 5000 particles and 100 time steps. Sparse [19] performs Cholesky factorization on sparse, symmetric, positive definite matrices, representing systems of equations. To make effective use of the memory hierarchy, Sparse uses supernodes which are groups of columns with similar non-zero structure. A typical parallel subtask in Sparse consists of one supernode updating another supernode in the matrix. For our experiments, we used a 4884×4884 matrix with 285494 nonzero elements.

4 Experimental Results

We begin this section with performance results for individual applications as the number of processes is varied. The purpose is to provide a better understanding of the speedup characteristics of individual applications and to provide information on the basic benefits of the process control approach. The following subsection presents results for the system performance when multiple applications are run concurrently under standard UNIX, gang, and process control schedulers. We show that the process control approach wins by a significant margin. In the next subsection, we present performance when both process controlled and non-controlled applications are present at the same time. We show that the use of the processor partitioning approach prevents the non-controlled applications from monopolizing the system resources, thus resulting in good performance for both kinds of applications. In the following subsection we present experiments that show the affect of our policy module on the response time of interactive applications. Finally, in the last subsection we show the effectiveness of the process control approach for applications performing a significant amount of I/O.

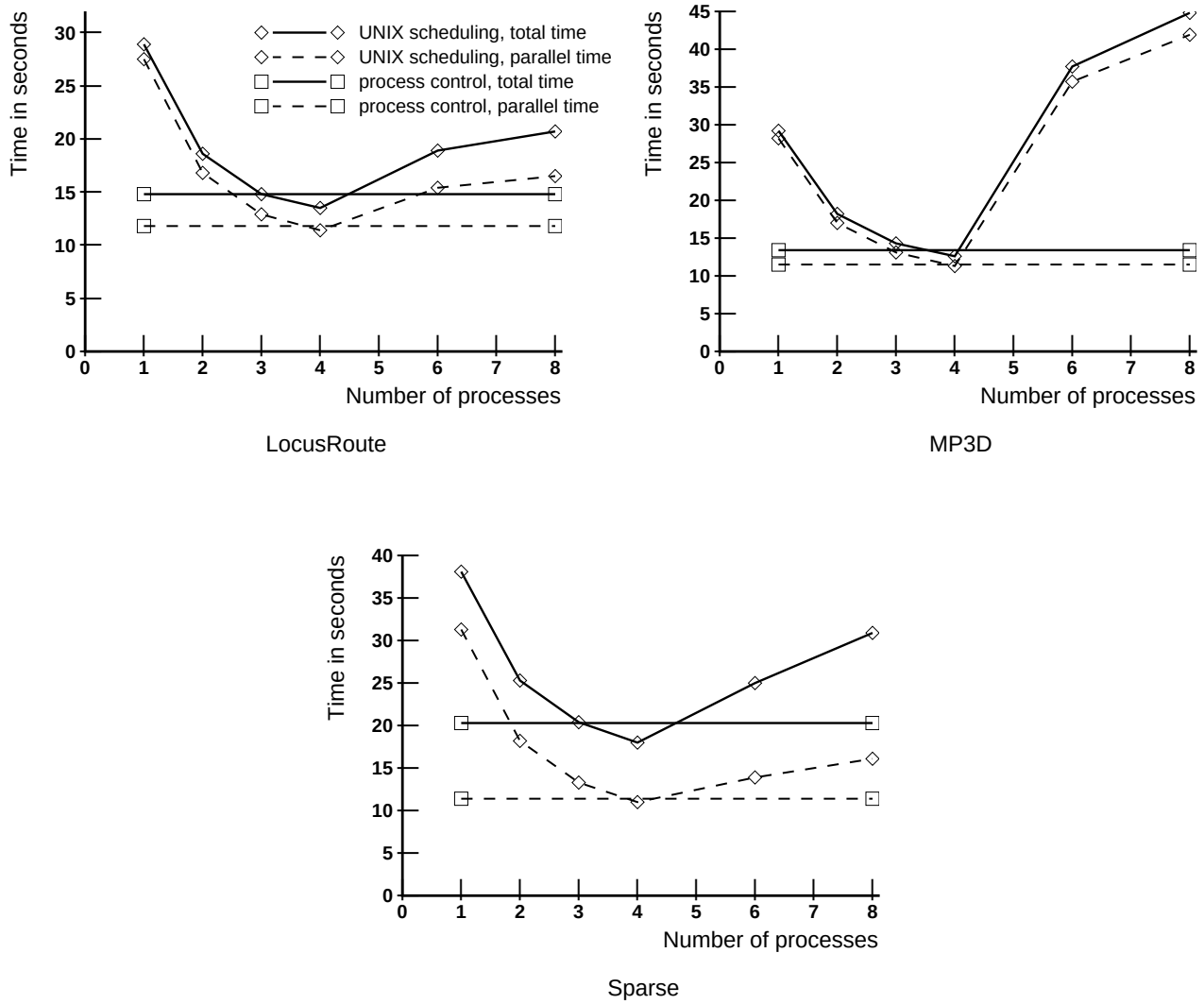

Figure 2: Execution time of applications running individually with the standard UNIX scheduler as the number of processes is varied and with process control.

4.1 Basic Application Performance

In order to get an idea of how the applications behave with different numbers of processes, and to show the effects of excess processes, we begin with a simple experiment where each application is executed with no other competing applications on the system. Each application is run, using the regular UNIX scheduler, with the number of processes varying between 1 and 8. We also examine the performance of the applications using process control when each application is started with 8 processes. The running times (in seconds) are shown in Figure 2. The dashed line in each graph refers to execution time of the parallel portion of the application, that is, the time when the application is actually doing parallel work. The application may have created multiple processes before this point, but only one is busy doing initialization during this period and the rest are waiting for work to be placed on the task queue. The solid line refers to the total elapsed time for the application, including both serial and parallel portions of the application.

With 1 to 4 processes, all three applications behave similarly. The peak performance occurs when the number of processes matches the number of processors. The speedup of the parallel portion of the code with 4 processes for LocusRoute, MP3D, and Sparse is 2.4, 2.5, and 2.8, respectively. The speedup is not perfect due to load balancing problems and due to a worse cache hit rate as compared to the sequential runs. As the number of processes is increased, the cache hit rate gets worse due to two reasons: (i) there is less spatial locality as adjacent locations within the same cache line may be used by different processes (also called *false sharing*), and (ii) the number of misses corresponding to true communication between the parallel processes increases. As one would expect, the performance of the applications using process control is similar to that with the standard UNIX scheduler with 4 processes, since process control simply causes the application to suspend processes until it is using 4 processes. The slightly higher running times with process control are due to the cost of handling process control signals and due to the suspension and resumption of processes — the applications received about 40 signals each to block or unblock processes, largely due to intermittent execution of system processes in the default processor set.

With greater than 4 processes, there is a significant drop in performance when the applications are run under the standard UNIX scheduler. LocusRoute takes 1.5 times as long to run with 8 processes as it does with 4 processes. MP3D is even worse, taking over 3.5 times as long to run with 8 processes as it does with 4 processes. This is largely due to poor cache behavior as intervening processes flush relevant data from the cache. The parallel portion of Sparse slows down by a factor of about 1.5 when 8 processes are used, and the performance of the application as a whole drops even more, by a factor of about 1.7. This is because Sparse spends several seconds in the beginning for reading, storing, and preprocessing its large input matrix. During this time, only one process is doing useful work. Without process control, this process does not get the full use of one processor, as the remaining seven processes created by the COOL runtime system compete for processor resources. With process control, however, the idle processes are immediately suspended, and hence the better speedup.

4.2 Multiprogramming Performance

We now explore the performance of different scheduling policies when multiple applications are run at the same time. In addition to the regular UNIX and process control schedulers, we also consider gang scheduling as provided by the Silicon Graphics IRIX operating system. Figure 3 compares the performance of these scheduling strategies when the LocusRoute and MP3D applications are run concurrently, and Figure 4 compares the performance when all three applications are run concurrently. In each case, all applications were started at the same time with 4 processes each. The bottom (light gray) portion of each bar in the figures refers to the elapsed time for the parallel portion of the code (corresponding to the dashed line in Figure 2). The top (dark gray) portion refers to the elapsed time for the serial portion of the code. The complete height of the bar denotes the total elapsed time for the application run (corresponding to the solid line in Figure 2).

As the data shows, process control does significantly better than the other two scheduling strategies. There are two main reasons for this. First, with process control and processor partitioning, each application runs in its own environment with almost no interference from other applications. This lack of interference results in better cache behavior and leads to higher performance. Second, with process control, each application runs at a better "operating point" on its speedup curve. To illustrate this point, note that most parallel applications get sublinear speedups with increasing number of processors (i.e., the speedup with N processors is less than twice the speedup with $N/2$ processors) due to load balancing problems, worse cache behavior, greater contention for locks, etc. Consequently, when a large number of processors are used to run an application, the processor efficiency is less than when fewer processors are used. The result is that in the presence of multiprogramming, techniques like gang scheduling exhibit low processor efficiency, since they use all the processors in the machine for each application. In contrast, the process control approach dynamically partitions the machine into

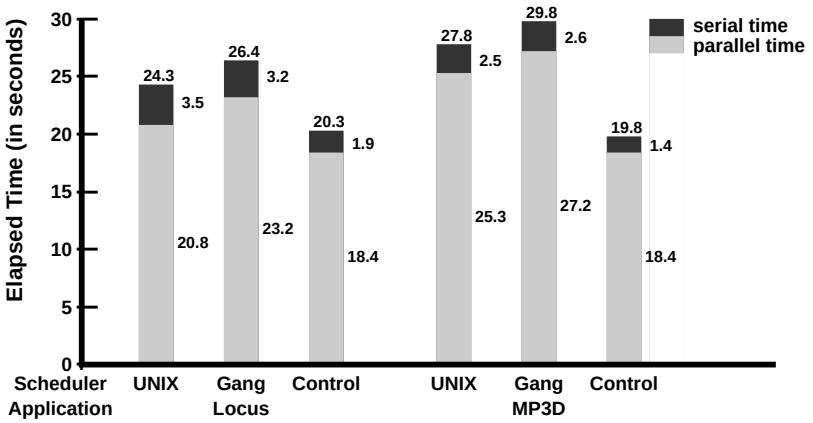

Figure 3: Performance comparison when the LocusRoute and MP3D applications are run concurrently under standard UNIX, gang, and process control schedulers.

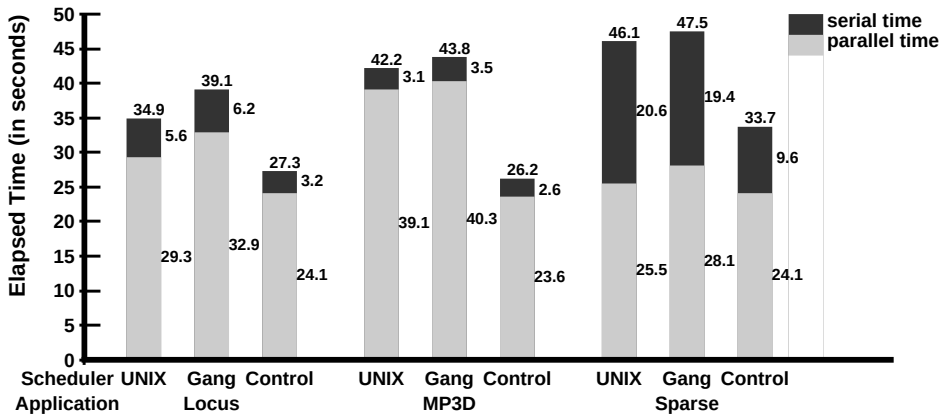

Figure 4: Performance comparison when the LocusRoute, MP3D, and Sparse applications are run concurrently under standard UNIX, gang, and process control schedulers.

several smaller machines, one per application, with each smaller machine providing higher processor utilization.[7]

Focusing on the performance of gang scheduling, the low performance is quite surprising — the applications do even worse than with standard UNIX scheduling. One reason for this, as we discussed above, is that the applications are operating at a less favorable point on their speedup curve. Although the applications have the same number of active processes under standard UNIX scheduling and under gang scheduling, with UNIX fewer processes are running simultaneously, on average. Another reason for the poor performance of gang scheduling, we believe, is the specific implementation of gang scheduling used in the Silicon Graphics IRIX system. Under IRIX, applications can either be gang-scheduled or non-gang-scheduled. Whenever a processor picks a process from a gang-scheduled application, it sends an interrupt to all other processors indicating that they should also start running

[7]As an interesting aside about the workings of the processor allocation module, observe that with process control and three applications (see Figure 4), the four processors in the SGI 4D/340 do not divide evenly among the applications. As discussed in Section 2, the policy module switches one of the processors between the applications periodically (every 300ms) to maintain fairness. Since the overhead of the processor reallocation is small and it is done infrequently, the process control approach performs quite well.

processes from this application. A processor responding to such an interrupt attempts to schedule such a process if several conditions are satisfied (for example, if the current process that is scheduled on it has been running for at least 20ms). Because of these conditions and because of the complex dynamics of a real system (where processes are blocking and unblocking on I/O and semaphores continuously), we found that all processes from the same application were rarely scheduled together. Detailed traces of the processes scheduled on each processor showed that with 3 applications running concurrently, all 4 processes from an application were running simultaneously only 12% of the time. Moreover, gang-scheduling interrupts were generated 20,000 times over the course of the applications' execution. Estimating that the combined overhead of sending and receiving an interrupt is about 80 microseconds, the total interrupt overhead alone comes to 1.6 seconds. Because gang scheduling, at least in this implementation, does not seem to provide any advantages, we do not consider it for the remaining experiments in this paper.

4.3 Mixing Controlled and Non-controlled Applications

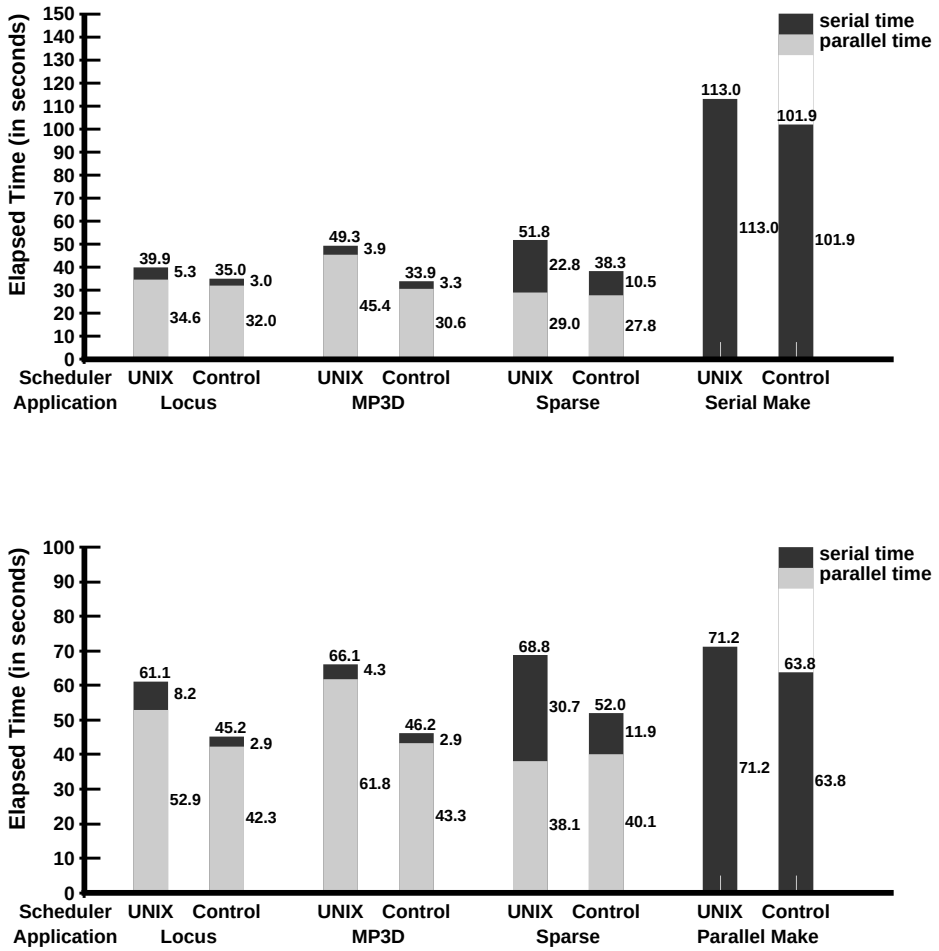

Figure 5: Performance of 3 applications running simultaneously with a serial make and a 4-process parallel make with standard UNIX and with process control scheduler.

The experiments so far have focused on system workloads consisting solely of process-controlled parallel applications. We now investigate the performance of the scheduling strategies when applications that do not respond to process control are also run concurrently. In Figure 5, we present results for two different workloads. In the first one, our three process controlled applications are run concurrently

with a serial *make* application, and in the second one, they are run concurrently with a 4-process parallel *make* application. In both cases, the make application is run in the default processor set. The parallel make starts multiple compiler processes, compiling different files, up to a preset maximum level of parallelism.

The data shows that the process control approach performs significantly better than the standard UNIX scheduler both for the process-controlled applications and for the make applications. According to the processor allocation policy described in Section 2.3, in the workload with the serial make, one processor is assigned to the default processor set to handle that application. In the workload with the parallel make, multiple processors are assigned to the default processor set if all 4 compiler processes are running. If the number of runnable processes drops due to I/O, some of the processors are switched back to the other applications. From the limited evidence provided by these experiments, it appears that the process control approach can handle loads consisting of multiple serial and parallel applications quite well.

4.4 Response Time Issues for Interactive Applications

Another interesting class of serial applications is interactive programs, where the applications sleep for long periods of time broken by short periods of CPU use. Applications of this type abound on timesharing systems, from network servers to editors. The important factor in the performance of these applications is response time, not throughput. In our system we have sacrificed some response time in favor of parallel application throughput by allowing all processors to be taken away from the default processor set (see Section 2.3). We now examine how much effect this tradeoff has on response time. To test this aspect of performance, we implemented an application that repeatedly sleeps for a random period of time, then wakes up and executes for a short time before sleeping again. By comparing the measured duration of the sleep with the actual duration that was requested, we can tell how long it took to run the process after it woke up.

We ran this application at the same time as our other three parallel applications. We compared the performance under the standard UNIX scheduler and under process control with two different policy modules. Under policy-1, at least one processor is always assigned to the default processor set to improve response time. Under policy-2, all processors may be taken away from the default processor set. However, the run queue of the default processor set is checked every 100ms and a processor is moved back if needed (as described in Section 2.3). The parallel execution times and approximate response times are shown in Figure 6.

With the standard scheduler, the average response time is fast (5ms) but the performance of the parallel applications, as seen before, is poor (average execution time for the applications is 42.8s). With process control and policy-1, the response time is just as good (5ms) but the performance is much better (average execution time is 35.0s). With process control and policy-2, the response time slows down (67ms), but the performance is even better than that with policy-1 (average execution time is 30.9s). The response time is worse for policy-2 because all processors are allocated to non-default processor sets most of the time. When the application in the default processor set becomes runnable, there is a time lag of between 0ms and 100ms before a processor is allocated to the default processor set. In addition there may be extra delay while higher-priority processes (also in the default processor set) are executed. We believe that for many applications the additional delay of 60ms in response time may be acceptable, particularly since it occurs only when no other processes in the default processor set are runnable. If faster response time is necessary, it may be achieved by increasing the frequency at which the default run queue is checked, at a slight extra cost in overall performance. At this point, however, more experience and experimentation are needed to fully understand this issue.

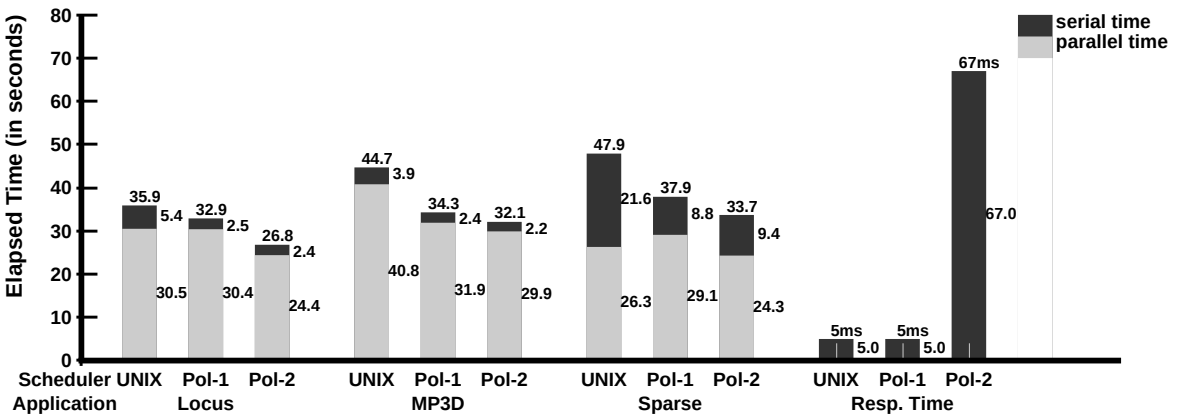

Figure 6: Performance of 3 applications running simultaneously with an interactive serial application, along with system response time.

4.5 The Performance of Process Control in the Presence of I/O

As stated in Section 2.2, when a process blocks on I/O, a signal is sent to the application so that it may create or resume a process to run on the processor just made idle. So far, however, our experiments have not shown the benefits of this feature since our applications perform very little I/O. To generate more I/O activity in the system without changing the applications, we reduced the memory in our machine from 32 Mbytes to 12 Mbytes. We then ran LocusRoute, MP3D, and Sparse, with the standard UNIX scheduler and under process control. The process control experiment was further divided into two subparts. In one case, we turned OFF the signals that are generated when a process blocks on I/O. In the other case, these signals are turned ON. (Signals that are sent when a new processor is assigned to a processor set or when one is taken away are turned ON in both cases.) The execution times are shown in Figure 7. Although the vagaries of the swapping algorithm in the operating system cause the relative performance of different applications to change in each run, the average of the execution times of the three applications (or the *mean turnaround time* of the applications) is consistently the best for process control with I/O generated signals turned ON. The mean turnaround time is 66.0s with I/O signals turned ON, 74.9s with I/O signals turned OFF, and 87.6s with the standard UNIX scheduler. While reducing the main memory in the machine is a somewhat contrived situation, our experiment does demonstrate improved processor utilization when proper action is taken on I/O and page faults.

5 Related Work

A number of researchers have studied the problem of scheduling processes on multiprocessor architectures. In this section, we briefly explore some of the more directly relevant work.

5.1 Scheduler Activations

Anderson et al. at the University of Washington [1] have developed a system that is similar in concept and motivation to the process control and processor partitioning approach proposed in [24] and further developed in this paper. In their work, *scheduler activations* are directly mapped onto processors assigned to an application in a one-to-one manner. (Scheduler activations are essentially the same as processes within an address space or kernel-level threads.) The scheduler activations are responsible

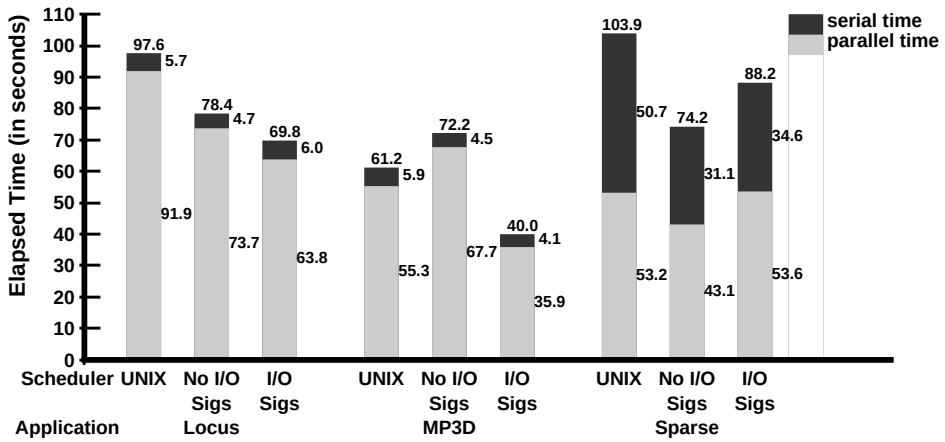

Figure 7: Performance of applications with standard UNIX scheduler and with process control. For process control scheduler, data are presented with I/O generated signals turned ON and turned OFF.

for executing user-level tasks as per the task-queue model. When a task running on a scheduler activation blocks for I/O, or when the kernel assigns another processor to the application, the kernel directly and automatically creates a new scheduler activation. This scheduler activation is then run on the newly freed/allocated processor and it starts picking tasks from the task queue and executing them. Similarly, when the kernel wishes to take away a processor from an application, it directly intervenes and destroys one of the scheduler activations in a *safe* manner. Primitives are also provided for the application to inform the kernel when it cannot use the resources that have been allocated to it, or when it needs more resources. As we see, similar to the process control approach, there is a close correspondence maintained between the server processes and the number of processors allocated to an application. To the extent that the number of active processes matches the number of processors, both approaches eliminate the need for context switches and thus ensure low overheads and good cache performance.

While the approaches are conceptually similar, there are some differences in the underlying mechanisms and implementations. In the scheduler activations model, the user-level threads package and the kernel are more tightly coupled than in our approach. For example, the kernel directly creates and destroys scheduler activations in response to changes in numbers of processors, while in our approach the kernel only provides hints to the threads package which then resumes or suspends server processes. The method used to ensure safe suspension is also different in the two approaches. The scheduler activation approach uses a recovery-based method. To destroy an activation (when a processor is to be taken away), the kernel first preempts two activations saving their tasks. The kernel then creates a new activation which makes sure that the two saved tasks were not blocked within a critical section. It does so by further executing one of the saved tasks until it is outside of critical sections, then putting that task on the threads package's run queue, and then executing the second saved task. In contrast, we use a simple preventive approach in our prototype implementation—we wait until an application process itself decides that it is at a safe point before suspending it. Although this introduces some delay in suspension of excess processes, our method does not require that all critical sections in the application code be clearly identified. In our experiments so far, we have not found the delays in suspension to be a problem. Rather, we have found that the slightly sluggish response of our approach helps prevent many short duration suspensions and resumptions of processes, which if done immediately would have degraded system performance. This would be even more true in large-scale multiprocessors [13] where the cost of cache misses can be very high, with each miss costing tens or hundreds of processor cycles. In such machines, it is often better to have the processor be idle for a

few cycles, rather than bringing in a new task that may destroy the cache data accumulated by the previous task.

5.2 Processor Partitioning

The processor partitioning kernel interface we use owes much of its structure to the implementation of processor sets developed for Mach [2]. In Mach, users may flexibly create and destroy processor sets, assign processes to different processor sets, and allocate processors to processor sets (security restrictions notwithstanding). This allows a variety of scheduling policies to be implemented by user-level servers, though initially only a simple batch-based scheduling policy was implemented. In our system, we use the same basic mechanisms, but have implemented a sophisticated kernel-based policy module to control processor allocation. This lets us provide an integrated scheduling policy that provides processor partitioning and does not depend on user-level interaction and control.

In Section 2 we mentioned that applications need to be able to communicate their resource requirements to the kernel, so that processors that cannot be used by an application can be allocated elsewhere. The importance of using such information in scheduling decisions has previously been emphasized by Majumdar et al. [14] and Zahorjan and McCann [27]. In a recent report, McCann et al. [15] empirically evaluate the use of such a "dynamic" processor allocation approach and the results show noticeable benefits when running applications with highly variable parallelism. The benefits are relative to the performance of an "equipartition" policy that is based on the concept of process control [24], but unfortunately lacks some important elements of our process control approach. The equipartition policy described requires that a processor being moved between applications be "released" by the application to which it was allocated before the reallocation can take place. This results in problems with response time, as new applications must wait for processors to be released before executing, and in problems with fairness when running uncooperative applications (which do not release processors when requested). Our process control policy avoids these problems by allowing the kernel to reallocate processors as needed, informing applications of reallocations but not waiting for the applications to respond.

5.3 Preventing Oblivious Process Preemption

The problems that occur when a process is preempted inside a critical section have been studied by several researchers. The NYU Ultracomputer operating system [6] lets processes give hints to the scheduler to avoid being preempted inside a critical section. Zahorjan et al. [26] suggest a similar approach, where a shared flag is used to inform the kernel that the process is inside a critical section. A slightly different approach has been taken in the Psyche system [20]. In the Psyche system, the kernel (rather than the application process) sets a shared flag when it is about to preempt a user process. The application process can then decide not to enter the critical section. While these approaches help avoid preemption inside simple critical sections, they do not address the problem of poor cache behavior resulting from context switching, and as a result they are of limited value.

5.4 Gang Scheduling

A number of researchers have proposed gang scheduling as a solution to the problem of running multiple parallel applications on a shared-memory multiprocessor. The basic idea is to run all processes belonging to an application at the same time, so that no process has to busy wait for a preempted process. The idea first originated as *coscheduling* with the Medusa system [17]. Other implementations have been proposed for the NYU Ultracomputer system [6] and Silicon Graphics' IRIX system also supports it. While the concept of gang-scheduling is attractive, providing an implementation that

works well on a real system is difficult. At least in the case of the IRIX operating system, we have found that as a result of blockages due to I/O and page faults, all processes of an application are rarely scheduled simultaneously (details were presented in Section 4.2). Gang scheduling also has the problem that it results in poor cache performance when all processes of one application alternate with processes of another application every time slice [10]. Finally, gang scheduling is a centralized approach to scheduling and it may thus become a bottleneck in highly parallel systems [7].

6 Conclusions

In this paper, we have proposed an approach to multiprocessor scheduling that uses cooperation between the kernel and applications to execute applications efficiently. Our approach, called *process control*, involves providing support for applications to dynamically match the number of processes they use to the number of processors that are available to them. Processors are partitioned among applications in such a way that multiple applications may be run simultaneously while preserving fairness. The partitioning also serves to promote stable running environments and improves cache efficiency.

An important facet of the structure of our system is that it allows flexibility in the applications' use of process control. The decision of when a process should be suspended, resumed, or created, is left to user-level software, rather than being embedded in the kernel. This also allows our modifications to fit into the standard UNIX model of process scheduling; the system still supports run queues, time slices, process priorities, and other standard UNIX mechanisms. In fact, if there is only one active processor set in the system, processes will be scheduled exactly as they would be in a standard UNIX system.

The process control approach is currently implemented on a Silicon Graphics 4D/340 multiprocessor. We have made the necessary modifications to the IRIX operating system and we have modified the runtime system of the COOL programming language to support process control. We have evaluated the performance of several workloads, consisting of both serial and parallel applications, when using the process control approach, and also when using more traditional scheduling techniques.

The process control approach performed much better than the standard UNIX scheduler when the number of processes exceeded the number of processors under both single-application and multi-application conditions. The better performance can be attributed to two main factors. First, process control offered significantly reduced context switch rates, eliminating direct context switch overhead and indirectly providing better cache hit rates. The hit rates improved since there were no intervening processes to displace useful data from the cache. Second, a more subtle factor, the performance also improved because the process control approach allowed each application to run at a better "operating point" on its speedup versus processors curve. By running each application with fewer processes, process control helped improve cache hit rate (reducing false-sharing and communication misses), load balancing, and synchronization overhead for the applications.

The process control approach was also shown to work well when non-process-controlled and interactive applications were run concurrently with process-controlled applications; processors were allocated in such a way as to preserve the performance of process-controlled applications while retaining fast turnaround and response time for other types of applications. We also studied the performance of the process control approach in the presence of a significant amount of I/O. The use of kernel notifications when processes block on I/O kept processors from idling and maintained high efficiency.

In conclusion, we believe that process control is a simple but effective approach to improving the performance of parallel applications on multiprogrammed systems, and that it is one that may be cleanly integrated into a traditional operating system. We think that process control will provide

additional benefits when applied to large-scale parallel machines [13], where the costs of scheduling without careful attention to processor allocation are much greater.

Acknowledgments

We would like to thank Rohit Chandra for making modifications to the runtime system of COOL to support process control and Kourosh Gharachorloo for helpful comments on the paper. The research was supported by DARPA contract N00014-87-K-0828. Anoop Gupta is also supported by a NSF Presidential Young Investigator Award with matching funds from TRW, Tandem, and Sumitomo.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95–109, October 1991.

[2] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[3] Rohit Chandra, Anoop Gupta, and John Hennessy. *COOL: A Language for Parallel Programming*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[4] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, 1988.

[5] Thomas W. Doeppner, Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Brown University, 1987.

[6] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pages 1–17, 1988.

[7] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, May 1990.

[8] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–43, 1984.

[9] Anoop Gupta, Charles Forgy, Dirk Kalp, Allen Newell, and Milind Tambe. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[10] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of SIGMETRICS '91*, pages 120–132, May 1991.

[11] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[12] Monica S. Lam and Martin Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[13] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[14] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of SIGMETRICS '88*, pages 104–113, 1988.

[15] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors. Technical Report 90-03-02, Department of Computer Science, University of Washington, February 1991.

[16] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[17] John K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[18] Jonathan Rose. LocusRoute: A parallel global router for standard cells. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, June 1988.

[19] Edward Rothberg and Anoop Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[20] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 70–78, 1990.

[21] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[22] Larry Soule and Anoop Gupta. Characterization of parallelism and deadlocks in distributed digital logic simulation. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, June 1989.

[23] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[24] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[25] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 26–40, October 1991.

[26] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.

[27] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of SIGMETRICS '90*, pages 214–225, 1990.