The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors

Raj Vaswani and John Zahorjan

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

In a shared memory multiprocessor with caches, executing tasks develop "affinity" to processors by filling their caches with data and instructions during execution. A scheduling policy that ignores this affinity may waste processing power by causing excessive cache refilling.

Our work focuses on quantifying the effect of processor reallocation on the performance of various parallel applications multiprogrammed on a shared memory multiprocessor, and on evaluating how the magnitude of this cost affects the choice of scheduling policy.

We first identify the components of application response time, including processor reallocation costs. Next, we measure the impact of reallocation on the cache behavior of several parallel applications executing on a Sequent Symmetry multiprocessor. We also measure the performance of these applications under a number of alternative allocation policies. These experiments lead us to conclude that on current machines processor affinity has only a very weak influence on the choice of scheduling discipline, and that the benefits of frequent processor reallocation (in response to the changing parallelism of jobs) outweigh the penalties imposed by such reallocation. Finally, we use this experimental data to parameterize a simple analytic model, allowing us to evaluate the effect of processor affinity on future machines, those containing faster processors and larger caches.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).

Authors' email addresses: *raj/zahorjan@cs.washington.edu.*

1. Introduction

We examine the effect of processor reallocation on parallel programs multiprogrammed on UMA shared memory parallel computers. Such machines typically have a modest number of processors connected to main memory by a shared bus. This architecture is in use for machines ranging in power from single-user workstations to supercomputers, and provides a building block for several proposed architectures for large-scale shared memory multiprocessors [Lenoski et al. 90].

Recent work [Tucker & Gupta 89, Gupta et al. 91, McCann et al. 91] has shown that the best performance is obtained by partitioning the available processors among concurrently executing jobs (*space sharing*) rather than by rotating the processors among them in a quantum-driven fashion (*time sharing*). At their most basic level, space sharing policies divide the available processors among jobs. However, even within this domain, a fundamental degree of freedom is the frequency with which allocation decisions are made. At one extreme, processors can be statically equipartitioned among jobs, with reallocations done only when jobs enter or leave the system. At the other extreme, processors can be reallocated unequally in the short term in response to the instantaneous processor demands of jobs, with care to ensure an equitable allocation when averaged over a longer time interval.

We consider whether processor utilization should be emphasized over other considerations that arise on multiprocessor machines with caches. We assume that on such machines, parallel jobs will be composed of several *tasks* (kernel-schedulable threads of execution). These may either implement the application directly, or may be used to run user-level threads that implement the application [Bershad et al. 88, Birrell 89]. In either case, a scheduling policy that ignores the tasks' cache behavior may result in poor utilization. Instead, it may be beneficial to schedule tasks where they have "affinity" (useful data remaining in the cache) [Squillante & Lazowska 89]. Depending on the magnitude of the affinity effect, it may even be desirable to avoid reallocation entirely, returning to more static policies.

$$RT_{X,j} = \frac{work_{X,j} + waste_{X,j} + \#reallocations_{X,j} \times \left[reallocation\text{-}time + cache\text{-}penalty_{X,j} \right]}{average\text{-}allocation_{X,j}} \quad (1)$$

$$cache\text{-}penalty_{X,j} = \%affinity_{X,j} \times P_j^A + \%no\text{-}affinity_{X,j} \times P_j^{NA} \quad (2)$$

Figure 1 - The Response Time Model

In the remainder of this paper we evaluate the tension between the benefit of processor reallocation (increased utilization) and the cost of this reallocation (poor cache behavior). In Section 2 we characterize the components of application response time. Section 3 describes our experimental environment (hardware, software, and parallel applications used). In Section 4 we evaluate empirically the effect of processor reallocation on the cache behavior of these applications. Section 5 defines the set of allocation policies that we use to explore the domain of space sharing policies. In Section 6 we summarize the results of experiments using these policies to schedule various workloads of parallel applications. In Section 7 we use data gathered from these experiments to parameterize our model, and thus to evaluate allocation policies for future machines. Section 8 discusses related work, and Section 9 summarizes our conclusions.

2. The Effect of Processor Reallocation

We now present a simple analytic model for comparing the impact of various multiprocessor scheduling policies on job behavior. This model serves two purposes. First, by clearly identifying the components of job response time, it facilitates our understanding of the possible approaches to improving performance. Second, while we use experimentation to assess the impact of affinity on current machines, we use our model to extrapolate this effect for future, much faster machines.

Our primary metric for evaluating multiprocessor scheduling policies is average job response time. Speedup, a common metric of parallel software performance, is not particularly intuitive when evaluating dynamic policies since a job's allocation changes throughout its lifetime.

Let $RT_{X,j}$ be the response time for a particular job j running under policy X in a multiprogrammed environment. We characterize job response time by equation (1) in Figure 1. In this equation, $work_{X,j}$ is the total number of processor-seconds of useful work comprising job j under discipline X, $waste_{X,j}$ is the total number of processor-seconds spent by j holding processors on which it has no work to execute, $\#reallocations_{X,j}$ is the number of processor reallocations j experiences, *reallocation-time* is the path length cost of a context switch, $cache\text{-}penalty_{X,j}$ is the cache effect of the switch (discussed below), and $average\text{-}allocation_{X,j}$ is the average number of processors that the scheduling policy is able to provide to j during its lifetime.

The magnitude of the $cache\text{-}penalty_{X,j}$ term in (1) depends on whether or not a task has an "affinity" for the processor on which that task is activated. We say that a task has affinity for processors on which it has previously run, and does not for others. However, in a multiprogrammed environment, a task experiences some cache penalty even when returning to a processor for which it has affinity: some intervening task may have run on that processor, ejecting some or all of the returning task's cache context.

Equation (2) in Figure 1, which represents the cache penalty of processor movement, reflects this effect. In this equation, $\%affinity_{X,j}$ and $\%no\text{-}affinity_{X,j}$ represent the percentage of reallocations under discipline X that cause job j's tasks to resume on processors for which those tasks do or do not have an affinity, respectively; P_j^A and P_j^{NA} represent the average cache penalties experienced by j in each of these cases.

Note that bus contention and thread synchronization delays are encapsulated (in an approximate way) by the $work_{X,j}$ term of equation (1). Either or both of these effects may be aggravated by particular scheduling disciplines — one that frequently migrates tasks, for example, may result in higher miss rates, and thus increased bus utilization (contention). For our purposes, it is only important that both forms of contention reduce effective processor speed, lengthening the number of processor-seconds required to complete the application. Therefore, measuring $work_{X,j}$ (as we do in subsequent sections) captures implicitly the differences in contention effects induced by alternative processor allocation policies.

Further, in dividing by *average-allocation* we are assuming that the impact on response time of the reallocations is evenly distributed over that number of processors. This assumption is justified by the software structure of our applications, which employ user-level threads to achieve fine-grained parallelism at low cost. This encourages the use of many threads, which are supported by a smaller, fixed number of workers (implemented as kernel threads). In such a scheme, reallocations need not take place more frequently on any one processor than on another. This assumption has agreed well with our experimental observations.

The model represented by Figure 1 suggests the following "degrees of freedom" available to policies in their attempts to reduce job response time:

- *Balancing #reallocations$_{X,j}$ and waste$_{X,j}$.* Wasting processors can be avoided by reallocating them from jobs that are currently unable to use them to jobs that can. However, this implies an increase in *#reallocations$_{X,j}$*, and so an increase in the processor reallocation cost component of job response time. Relatively static policies (such as in [Tucker & Gupta 89]) in effect assume that the reallocation penalty grows faster than the waste: such policies sacrifice utilization in order to minimize the cache penalty due to reallocation. Conversely, dynamic policies are willing to impose some cache penalty on applications in order to improve utilization (reduce waste). In what follows, we examine policies that fall at different points in this spectrum: the two extremes (maximum reallocations / minimum waste and minimum reallocations / maximum waste) as well as several other policies that attempt to strike a more balanced compromise.
- *Decreasing cache-penalty$_{X,j}$.* The cache penalty suffered under a dynamic discipline can be reduced by making decisions that maximize *%affinity$_{X,j}$*. Such a dynamic policy might perform as many reallocations as one that ignores affinity effects, thus keeping waste down. However, the cost of each reallocation would be reduced.

The model given by equations (1) and (2) suggests that the performance of a scheduling policy depends on a number of factors. To explain our results quantifying these factors, we first describe the environment in which those results were gathered.

3. The Experimental Environment

All of our work was done on a Sequent Symmetry Model B, a bus based, shared memory multiprocessor [Lovett & Thakkar 88]. Our machine consists of twenty 16 MHz Intel 80386 processors, each connected to a 64-Kbyte 2-way set associative cache with a line size of 16 bytes. The Symmetry Model B uses a copy-back, invalidation-based coherency protocol. We estimate that 0.75 μsec. is required to fetch a single cache block from main memory in the absence of bus contention, and therefore that (at least) 3.072 msec. would be required to fill entirely a single cache of 4K 16-byte blocks.

We control processor allocations in all of our experiments using Minos [McCann et al. 91]. Minos is a processor allocator designed to allow the easy implementation of alternative allocation policies. Minos runs as a user-level process, interacting with the Sequent operating system, DYNIX, to arrange the allocation of processors among jobs in the way dictated by the policy to be investigated.

In an attempt to model realistic workloads, we chose for our experiments a set of programs previously written for shared memory multiprocessors. These programs represent a variety of applications with differing parallelism structures. Figures 2 through 4 illustrate a number of characteristics of the applications. Included for each is the thread dependence graph: the nodes of the graph represent user-level threads and the edges represent the precedence relationships among them. Also shown are the percentage of time spent by the application at each level of physical parallelism, the total (elapsed) execution time, and the average processor demand, all measured by running the application in isolation on 16 processors.

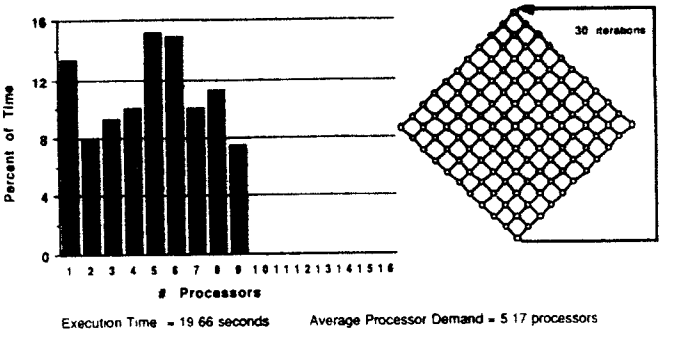


Figure 2 - The MVA Application

The first application, MVA, is a dynamic programming problem. Its precedence structure is representative of many "wave front" computations, and exhibits parallelism that first slowly grows and then slowly decreases.

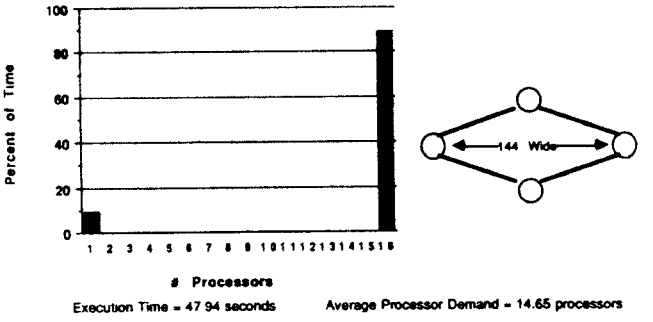


Figure 3 - The MATRIX Application

The second application, MATRIX, is an implementation of a parallel matrix multiply algorithm. The program uses a "blocked" algorithm designed to improve performance by exploiting cache locality [Fox et al. 88, Lam et al. 91]. Each thread of the computation is assigned a square block of elements of the output matrix for which it must compute values. It does this by dividing the two input matrices into blocks, and doing blockwise multiplication to compute partial results. The block sizes are chosen as large as possible under the constraint that the currently used blocks fit in the processor's cache. This results in very high cache hit rates, and so good application performance.

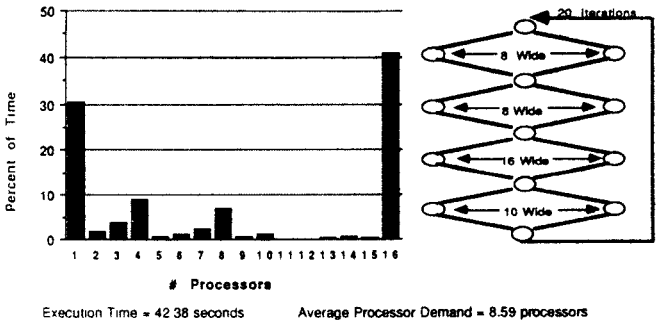


Figure 4 - The GRAVITY Application

The final application, GRAVITY, implements the Barnes and Hut clustering algorithm for simulating the gravitational interaction of a large number of stars over time [Barnes & Hut 86]. This application repeats five phases of execution for each time step of the simulation, the first being sequential and the remaining four parallel. Between each of the parallel phases is a barrier synchronization at which the parallelism decreases briefly to one. Thread execution times for GRAVITY differ during each phase, and within some phases, thread times depend on synchronization delays for critical sections of code.

4. Measuring Reallocation Costs

There are two costs to reallocating a processor: the kernel path length cost associated with the reallocation, and the cache effects of that reallocation. Some simple experiments to measure the former on our Sequent found it to be about 750 μsec. This value then serves as a standard against which to compare the cache related costs.

As stated in equation (2), the average cache penalty to job j is divided into components: P_j^A, the penalty incurred when one of j's tasks resumes on a processor for which the task has affinity, and P_j^{NA}, the penalty incurred when the task resumes where it has no affinity. We measured P_j^A and P_j^{NA} for our applications as follows. We ran each program on a single processor controlled by a special allocator. This allocator rescheduled the program every Q msec., and at every such point, took one of three actions:

- *The program was immediately replaced on the processor on which it had been running.* Measuring the response time of the program under these conditions provided us with a basis for comparison; we called this response time $RT_{stationary,j}$.
- *Enough memory was referenced sequentially to flush the cache, and then the program was replaced.* This case was designed to capture P_j^{NA}, the cost incurred for running on a processor for which the program had no affinity.[1] We called the program's response time under these conditions $RT_{migrating,j}$.
- *A task from another program was run on the processor for duration Q, and then the original program was replaced.* This case was designed to capture P_j^A, the cost incurred for running on a processor for which the program did have affinity. Another program was run on the processor for duration Q to reflect a multiprogramming environment in which the activity of other jobs would have ejected some portion of the returning task's context from the cache. We called the program's response time under these conditions $RT_{multiprog,j}$.

The total cache penalty due to lack of processor affinity was given by $RT_{migrating,j} - RT_{stationary,j}$, while the penalty incurred in spite of such affinity was given by $RT_{multiprog,j} - RT_{stationary,j}$.

Since P_j^{NA} and P_j^A are defined as the cache penalty per context switch (reallocation), they were given by:

$$P_j^{NA} = \frac{RT_{migrating,j} - RT_{stationary,j}}{\#\,switches\ that\ occurred}$$

$$P_j^A = \frac{RT_{multiprog,j} - RT_{stationary,j}}{\#\,switches\ that\ occurred}$$

Table 1 summarizes our measurements of cache penalties. Each row of Table 1 represents the workload measured. Since P_j^A depends heavily on the behavior of intervening tasks, the column labels indicate the workload run between successive dispatches of the measured workload. In the case of P_j^{NA}, of course, there is no such workload. We performed these measurements for values of Q (the frequency with which the programs were rescheduled) of 25 msec., 100 msec., and 400 msec. The first two durations were meant to represent the typical length of an I/O operation and a typical quantum length for time shared multiprocessors, respectively.[2] The 400 msec. value represented a rough estimate of the frequency with which a dynamic space sharing policy might perform reallocations.

These results indicate a significant penalty for running tasks on processors for which the tasks have no affinity. (Recall that the path length cost of the context switch is 750 μsec.) Further, there is a noticeable penalty for resuming tasks on processors for which the tasks do have affinity, but on which another task has run. The magnitude of these costs appears heavily influenced by Q, for the following reasons.

[1] In fact, our results may somewhat overstate the non-affinity penalty that would be observed in practice, since an individual application might spread its cache context among several caches with the well known benefits this brings.

[2] Since multiprocessors have more processors than do uniprocessors, it is more likely that an idle one can be found when needed. This permits multiprocessor systems to reduce system overhead by using quanta that are typically longer than those on uniprocessors. DYNIX, for example, uses a 100 msec. quantum.

	P_j^{NA}	P_j^A			P_j^{NA}	P_j^A			P_j^{NA}	P_j^A		
		MAT	MVA	GRAV		MAT	MVA	GRAV		MAT	MVA	GRAV
MAT	882	120	177	165	1076	171	419	374	1679	737	1166	815
MVA	914	107	166	194	1267	164	330	221	2330	627	1061	1103
GRAV	364	154	301	210	1576	415	740	353	2349	1793	2080	1719
	Q = 25 msec.				Q = 100 msec.				Q = 400 msec.			

Table 1 - P_j^A and P_j^{NA} (in μsec.) for All Applications

Larger values of Q mean that a task runs for a longer period, accessing more data between each context switch than it did at smaller values of Q. If the task is resumed on a processor for which it has no affinity, the increased P_j^{NA} reflects the necessity of reloading this larger working set. If the task is resumed on a processor for which it does have affinity, the intervening task on that processor has also run for a longer period, accessing correspondingly more data. This increased use of the cache causes more of the returning task's data to be ejected, and therefore P_j^A also increases with Q.

In summary, the results of our measurements confirm that the cache effects of a processor reallocation can be the dominant portion of the reallocation cost. The improved utilization gained by frequent reallocation may be offset by the poor cache behavior induced by this reallocation. These results provide motivation to examine this tradeoff more carefully, and to evaluate policies that consider cache effects when making scheduling decisions.

5. Description of Scheduling Policies

Our goal now is to consider how important cache effects are to scheduling by evaluating a number of policies that differ in the amount of consideration they give to these effects. We examine only space sharing policies, since this basic characteristic has already been shown to be necessary for good performance [Tucker & Gupta 89, Gupta et al. 91, McCann et al. 91]. The policies we examine differ only with respect to the "degrees of freedom" listed in Section 2.

5.1. Equipartition

Equipartition is a space sharing policy that, to the extent possible, maintains a constant equal allocation of processors to all jobs. To do this requires reallocations only on job arrival and completion. In terms of our response time model, Equipartition is an extreme in the policy space that minimizes *#reallocations* at the expense of maximizing *waste*. In this sense, Equipartition provides perfect affinity scheduling, since tasks essentially never move.

Our Equipartition policy is based on the "process control" policy from [Tucker & Gupta 89]. Each time a reallocation must take place, the number of processors to allocate to each job is computed as follows. The "allocation number" of all jobs is initially set to zero, and then incremented by one in turn. Any job whose allocation number has reached its maximum parallelism (the maximum number of processors the job can use at any point in its computation) drops out. This process continues until either there are no remaining jobs or all processors have been allocated. The set of allocation numbers computed in this way gives the number of processors that should be allocated to the jobs.

5.2. Dynamic

Dynamic is a space sharing policy at the other extreme of the policy spectrum: it minimizes *waste* at the cost of a very large *#reallocations*. As described below, Dynamic has very poor affinity characteristics because it reallocates frequently and without regard to affinity.

Our Dynamic policy is taken from [McCann et al. 91]. Its allocation decisions depend on the current processor requirements of jobs. Dynamic attempts to allocate to each job exactly the number of processors it can use at that instant, with the additional constraint that averaged over longer time intervals the allocation is fair. Since the instantaneous processor demands of the jobs are known only to themselves, each job continually reflects to the allocator (via shared memory) the number of additional processors (possibly 0) the job could use. In addition, when a job has a processor that it cannot currently use profitably, it notifies the allocator that the processor is available for reallocation. Such processors are said to be *willing to yield.*

Dynamic attempts to satisfy requests for additional processors by using the least valuable processors currently available:

D.1 First, any unallocated processors are assigned.

D.2 Next, "willing to yield" processors are assigned.

D.3 Finally, an equitable allocation is enforced by preempting processors from the job(s) with the largest current allocation.

The Dynamic policy also includes an adaptive priority mechanism used to encourage jobs to give up processors not currently needed.[3] Each job is assigned a priority level

[3] The details of this priority scheme are not pertinent to the goals of this paper and so only an abbreviated summary is included here. [McCann et al. 91] contains a complete description.

that depends on its processor usage to that time. Job priorities are set using a scheme that raises them as a "reward" for using few processors and lowers them as a result of using many. In this way, a job acquires credit during periods when it uses few processors. The job may later spend these credits to obtain temporarily more than its fair share of processors. Such a priority mechanism has been shown to be essential due to considerations other than performance that arise in any realistic implementation of a processor allocation policy: fairness, interactive response time, and resilience to countermeasures designed to undermine the policy [McCann et al. 91].

5.3. Dynamic with Affinity (Dyn-Aff)

Since the Dynamic policy maximizes *#reallocations*, its performance is potentially sensitive to the cost of each one. An improvement might thus be obtained by making the same reallocation decisions but reducing the cost of each. The key to this is to maximize *%affinity*: that is, to introduce affinity scheduling.

Introducing affinity to Dynamic requires that the allocator have access to processor and task *histories* [Squillante & Lazowska 89]. For a processor, its history is an ordered list of the last T tasks to have run on it. For a task, its history is an ordered list of the last P processors on which it has run. In the work that follows, we remember only the last task or processor ($T = P = 1$).

We incorporated processor affinity into Dynamic's allocation decisions as follows:

A.1 Whenever a processor becomes available for reallocation, the last task to have run on it is identified using the processor's history. If that task (*last-task*) is not currently active on some other processor but is runnable with useful work to perform, and if the priority of the job to which *last-task* belongs is as high as that of any job currently requesting processors, then *last-task* is activated on the available processor. Otherwise, the processor is allocated to the requesting job of highest priority.

A.2 Whenever a job requests additional processors, it indicates to the allocator the processor that it would like to acquire (*desired-processor*). *Desired-processor* is determined using the per-task processor history, and is defined as the processor on which the task most critical to the job's progress last ran. If *desired-processor* is available for reallocation, then it is assigned to the requesting job. Otherwise, another available processor (if any) is assigned.

This augmented policy (Dyn-Aff) determines a processor's availability by applying allocation rules **D.1**, **D.2**, and **D.3** exactly as in the case of the basic Dynamic policy.

Under rule **A.2**, we allocate *desired-processor* only if that processor is not currently doing useful work, since otherwise we must preempt the task running there. Such preemption is counterproductive, since an active task presumably has greater affinity for the processor than the task we are attempting to schedule. This consideration limits the possible influence of affinity on the Dynamic discipline.

A further limitation is imposed by Dyn-Aff's unwillingness to sacrifice its priority scheme to affinity considerations: both rule **A.1** and **A.2** make the priority based decision in preference to the affinity one. Although a set of non-performance considerations argues for adherence to the priority scheme, this occasionally dictates that the policy ignore a potential affinity advantage. To evaluate the extent to which these non-performance considerations degrade potential performance, we define a new version of the Dyn-Aff policy, called Dyn-Aff-NoPri, that ignores priorities. (We emphasize that Dyn-Aff-NoPri is not suggested as a policy for implementation in real systems, but is rather an artificial policy used to determine the maximum benefit affinity scheduling might provide.) Dyn-Aff-NoPri behaves much like Dyn-Aff, except that:

- Allocation rule **D.3** is ignored. Dyn-Aff-NoPri does not enforce fairness via preemption of processors from the job(s) with the largest current allocation.
- Allocation rule **A.1** is modified so that when a processor is available it is always allocated to *last-task* if *last-task* is not currently active on another processor, but is runnable and has work to do, regardless of priority considerations.

Dyn-Aff-NoPri applies the other allocation rules exactly as in the case of the other policies.

5.4. Dynamic with Affinity and Yield-Delay (Dyn-Aff-Delay)

As described above, Equipartition and Dynamic represent extremes in the policy space with respect to *waste* and *#reallocations*. The final policy we consider is a less aggressive version of Dynamic that falls between these two extremes.

Under Dynamic and Dyn-Aff, a job indicates that it holds a "willing to yield" processor as soon as that processor becomes idle. Under the modified policy (Dyn-Aff-Delay), we allow applications to retain such processors for short periods of time in the hope that additional work for them to do will be generated within the job. If this happens, the job can begin this work without incurring a processor reallocation penalty.[4] Dyn-Aff-Delay thus trades slightly increased *waste* (while jobs are waiting for more work to arrive) for a reduced *#reallocations*. Dyn-Aff-Delay is in other respects identical to Dyn-Aff.

[4] These same considerations arise in implementing locks for mutual exclusion, and in this context a number of researchers have proposed a "spin-then-block" policy, the basic idea being to spin for a short time and then, if the lock is still not free, to give up the processor [Lo & Gligor 87, Karlin et al. 91].

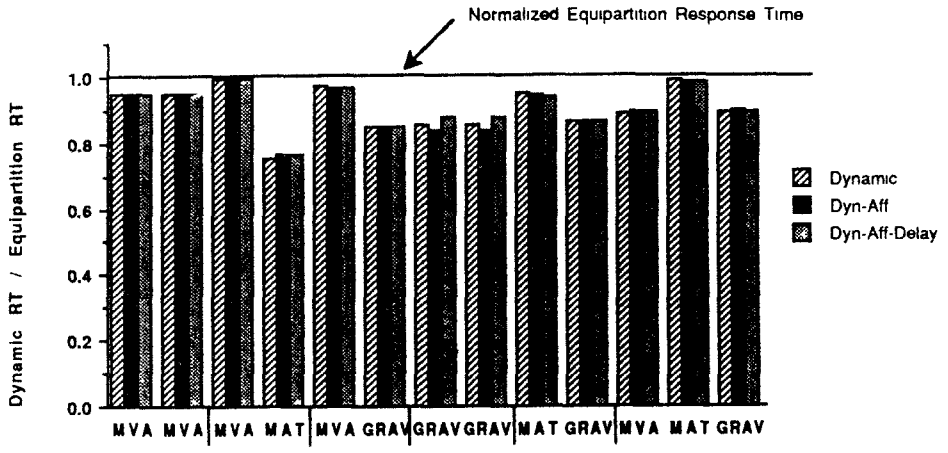


Figure 5 - Response Times under Various Dynamic Disciplines Relative to Equipartition

6. Policy Comparison on Current Technology Machines

In this section we have two goals. The first is to evaluate experimentally the impact of processor affinity on performance for current multiprocessors. Our second goal is to obtain parameter values from these experiments that can be used with our response time model to obtain a similar evaluation for future machines.

The results in this section are based on experiments with workloads containing some number (possibly zero) of jobs of each of the three application types discussed in Section 3. We chose the six sets of programs summarized in Table 2, each of which (based on the characteristics of its component programs) imposes qualitatively different demands on the scheduling policy.

	#1	#2	#3	#4	#5	#6
MVA	2	1	1	0	0	1
MATRIX	0	1	0	0	1	1
GRAVITY	0	0	1	2	1	1

Table 2 - #Copies of Each Program in Each Mix

Workload #1 presents a light load to the system in terms of the total average number of processors required by all jobs. Workload #2 represents a situation where one job (MVA) has dynamically changing parallelism but the other (MATRIX) has massive and constant parallelism. Workloads #3 and #4 present moderate loads, requiring more frequent reallocations than either of the previous two. Finally, #5 and #6 present reasonably heavy loads involving quickly changing parallelisms.

Using these workloads, we compare the dynamic policies to Equipartition since the latter places maximum emphasis on affinity, while the others emphasize affinity to varying but lesser degrees. As stated earlier, our primary metric of performance is average job response time. Figure 5 shows the average response time for each job in each workload mix for the three versions of the Dynamic policy relative to the response times obtained under Equipartition. The average values shown represent enough replications of each experiment so that the 95% confidence interval is within 1% of the point estimate of the mean.

Based on these results, we come to the following conclusions for current technology machines:

- *Aggressive reallocation of processors is preferable to more static allocation.* As Figure 5 shows, the response times for all jobs under the dynamic disciplines are smaller than the Equipartition response times. This result is in agreement with those in [McCann et al. 91], which contains a more complete discussion of this comparison.

- *Affinity scheduling provides little benefit under current conditions.* Figure 5 shows that the response times obtained by the three variants of Dynamic are basically identical. To gauge the effectiveness of the affinity versions of Dynamic in meeting their goals, we tracked the number of times that a task was restarted on a processor for which that task had affinity. Table 3 shows the percentage of such occurrences for workload #5 under the basic Dynamic policy and its two affinity-based derivatives. The dramatically higher *%affinity* under the two affinity policies suggests that they frequently dispatch tasks to processors for which those tasks have affinity. However, because current cache penalties (P_j^A and P_j^{NA} from Section 4) are small relative to the time between reallocations (row 3 of Table 3), response times (row 4) do not significantly improve. Section 8 discusses another factor that also contributes to this result.

	Dynamic		Dyn-Aff		Dyn-Aff-Delay	
	MAT	GRAV	MAT	GRAV	MAT	GRAV
%affinity	21%	31%	83%	54%	86%	59%
#reallocations	2469	1745	2409	1780	1611	1139
Realloc. interval (msec.)	293	222	300	218	445	340
Response time (sec.)	87.5	51.4	87.0	51.5	86.3	51.4

Table 3 - Influence of Affinity on Scheduling

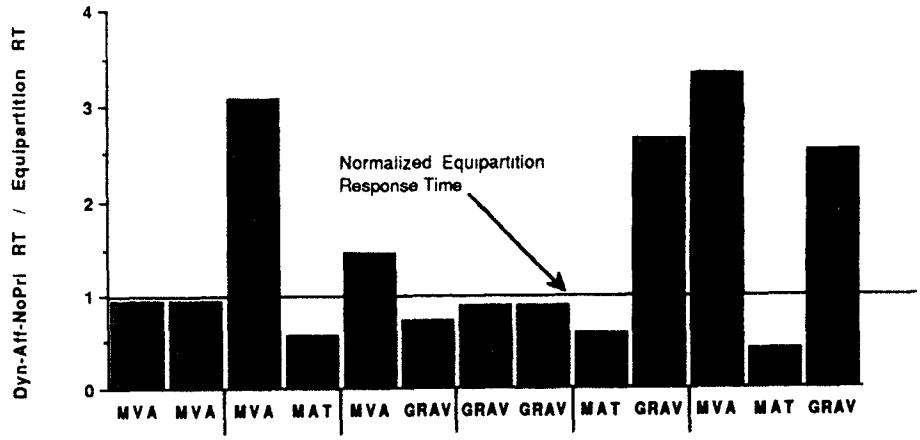


Figure 6 - Response Times under Dyn-Aff-NoPri Relative to Equipartition

We thus conclude that in our current environment, processor affinity need not unduly affect reallocation decisions. This result, while obvious in retrospect, was difficult to predict because measurement was useful in determining the actual number of reallocations experienced by jobs interacting in mixed workloads.

- *"Yield-delay" provides little benefit under current conditions.* While Dyn-Aff-Delay meets its goal of reducing *#reallocations* (row 2 of Table 3), this has little effect on response time. Once again, the reason is that on the current machine the reallocation penalty is small relative to the time between reallocations. Reallocation cost is thus only a small fraction of job response time, and reducing this component even further has negligible effect.

- *Fairness should not be sacrificed in an attempt to improve processor affinity.* Figure 6 shows the average response time for each job in each workload mix for the Dyn-Aff-NoPri policy relative to the response times obtained under Equipartition. In contrast to the results for the other dynamic policies (Figure 5), job response times relative to Equipartition are extremely variable. This occurs because Dyn-Aff-NoPri sacrifices the priority scheme, and thus fairness, for an increase in *%affinity*; this unpleasant behavior is why we consider Dyn-Aff-NoPri to be an artificial policy (as stated previously).

Nonetheless, our purpose in introducing Dyn-Aff-NoPri was to evaluate the extent to which non-performance considerations (fairness, etc.) degrade potential performance by forcing the policy to ignore possible affinity advantages. We therefore calculated the average response time experienced by each job under both the basic affinity policy (Dyn-Aff) and Dyn-Aff-NoPri. We made the calculation only for the homogeneous workloads (those containing multiple instances of the same type of job), since this statistic is meaningless for workloads containing different types of jobs.

Table 4 summarizes the average job response time for the homogeneous workloads (#1 and #4) under both Dyn-Aff and Dyn-Aff-NoPri. As the table demonstrates, sacrificing the priority scheme to affinity considerations results in a negligible improvement in the case of the MVA workload (#1), and a degradation in the case of the GRAVITY workload (#4). Since this weak and inconsistent behavior is obtained at the cost of significant unfairness (Figure 6), we conclude that (enforced) equitable allocation is essential in a reasonable policy, and eliminate Dyn-Aff-NoPri from further consideration.

	Dyn-Aff	Dyn-Aff-NoPri
Workload #1 (2 MVA jobs)	20.22	20.13
Workload #4 (2 GRAV jobs)	50.07	53.07

Table 4 - Average Job Response Time (Homogeneous Workloads Only)

Our observations to this point provide a good set of guidelines for constructing a scheduling policy for shared memory multiprocessors. In summary, an efficient policy will combine space sharing with careful but aggressive reallocation of processors. Although affinity considerations do not currently appear significant, there is little cost to including them in such a policy. Dyn-Aff-Delay thus appears to be the best such policy, since presumably it is most resilient to aberrant application behavior.

7. Policy Comparison on Future Technology Machines

Our results so far demonstrate that on current machines, affinity scheduling is not particularly useful: the dominant effect on performance is the increased processor utilization obtained from dynamic reallocation, which far outweighs the impact of these reallocations on application cache behavior. However, it is possible that on future machines, the cache effects of reallocations will negate the potential utilization improvement by forcing jobs to spend inordinate amounts of processing power reloading cache context instead of computing. In order to evaluate the point at which this might occur, we must extend our model for job response time (Figure 1) to reflect the characteristics of future machines.

$$RT_{X,j} = \frac{\frac{work_{X,j} + waste_{X,j}}{processor-speed} + \#reallocations_{X,j} \times \left[\frac{reallocation-time}{processor-speed} + \frac{cache-penalty_{X,j}^{future}}{\sqrt{processor-speed}} \right]}{average-allocation_{X,j}}$$

$$cache-penalty_{X,j}^{future} = \frac{\%affinity_{X,j} \times P_j^A}{cache-size} + (\%no-affinity_{X,j} \times P_j^{NA}) \times \sqrt{cache-size}$$

Figure 7 - The Response Time Model, Extended for Future Machines

7.1. Extending the Model

In this section, we describe the manner in which we incorporate the (projected) characteristics of future machines into our model. Section 7.2 discusses the necessary assumptions in more detail. The extended model itself is shown in Figure 7.

7.1.1. Faster processors

Increased speed is the primary characteristic of future machines that affects our model. We make the optimistic assumption that applications will be able to take full advantage of the increased processing power offered by future processors. That is, we assume that the purely computational terms in our model will decrease linearly with increasing processor speed. Thus, we simply divide the *work*, *waste* and *reallocation-time* terms of equation (1) by a *processor-speed* factor, which gives the speed of a future processor relative to a processor of our Sequent Symmetry. This assumption is optimistic since recent results indicate that due to architectural constraints, software may be unable to exploit the full power of faster processors [Ousterhout 90, Anderson et al. 91].

As processor speed increases, cache misses become increasingly important since more cycles are lost if miss resolution is necessary. Thus, one cannot build arbitrarily faster machines without addressing the bottleneck that the memory subsystem becomes on such machines. The following sections discuss proposed solutions to this bottleneck, which give rise to other characteristics of future machines that must be incorporated into our model.

7.1.2. Larger caches

This approach attacks the memory bottleneck by reducing the number of capacity misses (those due to limited cache size), and thus the number of overall misses. Previous work [Thiebaut & Stone 87, Mogul & Borg 91] has indicated that one effect of larger caches will be to allow more data to be preserved across context switches (processor reallocations). This suggests that a task returning to a processor for which the task has affinity will incur a smaller cache penalty if the cache is large: it is more likely that the returning task's cache image (i.e., useful data in the cache) has been left undisturbed by any other task(s) to have run on that processor. We incorporate this into our model by assuming that this effect is linear in cache size, and simply divide the cache penalty incurred when restarting on a processor to which a task has affinity (P_j^A) by a *cache-size* factor. This factor represents the size of a future cache relative to that of our Symmetry.

Increasing cache size may also be expected to affect tasks (re)starting on processors for which the tasks have no affinity. At one extreme, the cache penalty of restarting on such a processor might remain constant with increasing cache size; at the other extreme the penalty might increase linearly with cache size. [Wang et al. 89] observe that program hit rates grow extremely slowly as cache size increases, suggesting that the additional amount of cached data useful to a program is small. However, larger caches allow applications the luxury of loading more data into the cache, and future applications may therefore use the cache more extensively than do current applications. We therefore choose a function between constant and linear, and assume that the penalty grows as $\sqrt{cache-size}$, where *cache-size* is the relative size as above.

7.1.3. Faster cache miss resolution

Multilevel caches present one strategy for reducing the cost of cache misses by resolving misses in the first-level cache with data from the second-level cache rather than from (slower) main memory. However, this strategy depends on increased hit rates in both the first- and second-level caches in order to avoid accessing a still-slow main memory. It seems unlikely that these hit rates can be increased enough to allow the effective access time of main memory to remain constant as processor speeds increase, even if prefetch techniques are employed [Jouppi 90]. Thus, in order to fully exploit faster processors, the speed of the main memory subsystem (including any contention for the system bus) must increase with processor speed, although alternate strategies (such as multilevel caching) allow that the magnitude of this change need not precisely equal that of the processor speed. We assume that the miss resolution speed must increase as $\sqrt{processor-speed}$, a ratio in general agreement with [Jouppi 90]. The cache penalty in our model is therefore scaled by this factor.

7.2. Discussion

Much of what follows is predicated on the fact that our response time model is expressed using *seconds* as the time unit, rather than *processor cycles*. For example, $cache\text{-}penalty_{X,j}$ is in terms of *seconds lost* rather than (the more commonly used) *cycles lost*. For this reason, $cache\text{-}penalty_{X,j}$ is not expressed as a function of processor speed, except as described below.

The model attempts to reflect the fact that cache misses become increasingly important as processors become faster. However, we consider a "faster" processor to be one that provides an increased "effective processing rate" rather than one with merely a faster clock rate. A processor with a smaller cycle time (faster clock rate) is not useful if too many cycles are spent waiting for cache misses to be resolved. As described above, our model assumes that such a processor can be utilized effectively only if 1) memory speed is increased so that cache miss resolution is faster, and/or 2) program hit rates are increased so that fewer memory accesses are made. To gauge the amount by which hit rates must be increased, we analyzed a simple model consisting of two levels of cache memory and a single central memory. We found that because multiprocessor hit rates may already be expected to be quite high, there was little room for improvement: hit rates could not be increased enough to obviate the need for faster miss resolution. For this reason, the model assumes that (effective) memory speed must increase as $\sqrt{processor\text{-}speed}$, and the miss penalty is divided by this factor.

The model also assumes that larger caches will allow more data to survive across reallocations. This assumption may be optimistic, for the following reason. As already discussed, the relative cost of a cache miss increases with processor speed. It therefore becomes increasingly important that programs use algorithms that explicitly consider the cache in order to improve performance; "blocked" algorithms represent one example of this [Fox et al. 88, Lam et al. 91]. This greater emphasis may lead to an increase in the number of programs tailored to the cache size, which would reduce the importance of affinity: even a single intervening task would overwrite large portions of a returning task's cache context, reducing the benefit of returning to such a processor. We chose the simpler (optimistic) assumption because this effect is difficult to model without further assumptions about the characteristics of future programs.

In general, our model parameters are chosen to be both few enough and simple enough to be measured easily. We felt that manageability was imperative for the model to be useful, but this simplicity has a cost: we are able to describe only trends, and not precise behavior based on specific machine characteristics. Further refinement is necessary before more detailed extrapolations such as these are possible.

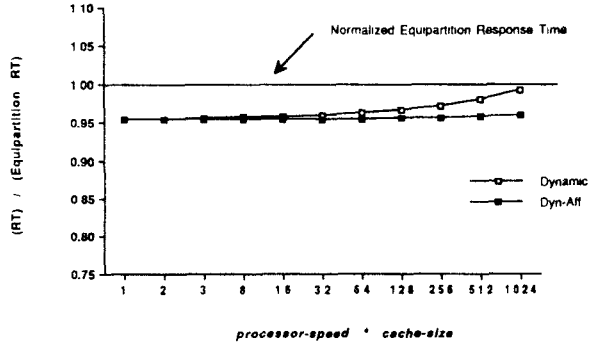
7.3. Results

Using the extended model to predict response times on future machines requires values for all of its terms. We obtained P_j^A and P_j^{NA} from the measurements made for each of our applications (Section 4). We extracted the other parameters from the results of scheduling various workloads with each of our allocation policies (Section 6). An example of this data is that shown in Table 3 (Section 6). We then evaluated the performance of each policy at various values of *cache-size* and *processor-speed*. We were interested in the point at which the cache penalty of reallocation overwhelmed the utilization benefit, and therefore again chose to evaluate the various dynamic policies with respect to Equipartition.

Figures 8 through 13 depict the performance of each dynamic policy relative to Equipartition for each application in each workload.[5] The X-axis of each graph is the product of $processor\text{-}speed$ and $cache\text{-}size$, since we observed that the response time results to more than three significant digits depend only on the product of these terms, and expressing the results this way simplifies the presentation.

Based on these results, we make the following observations:

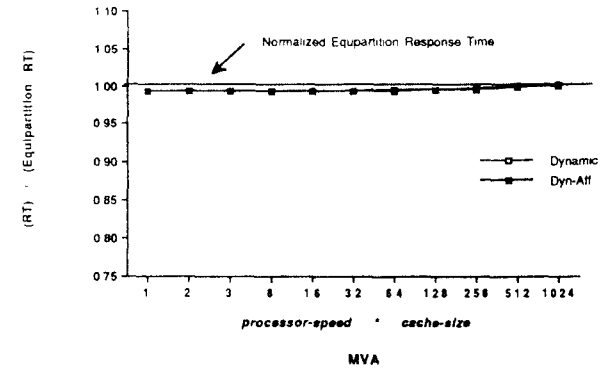
- *The benefits of increased processor utilization overwhelm the performance lost to cache effects.* As can be seen from the figures, the performance of the best dynamic policy (Dyn-Aff-Delay) is superior or equivalent to that of Equipartition. In graphs that show the performance becoming equivalent, the "crossover point" is quite far in the future. This suggests that a careful policy can reap the benefits of properly utilizing faster processors without negating this effect by inducing poor cache behavior.



wkload1 - 2 mva

Figure 8 - Relative Resp. Times for Workload 1

[5] In some cases, we have omitted the curve for Dyn-Aff-Delay. For workloads with few reallocations, "yield delay" has a negligible impact on performance, and Dyn-Aff-Delay behaves much like Dyn-Aff. We omit the Dyn-Aff-Delay curve in such cases to avoid cluttering the presentation.



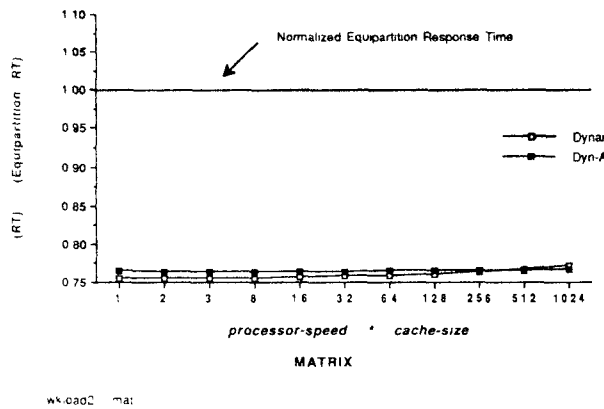
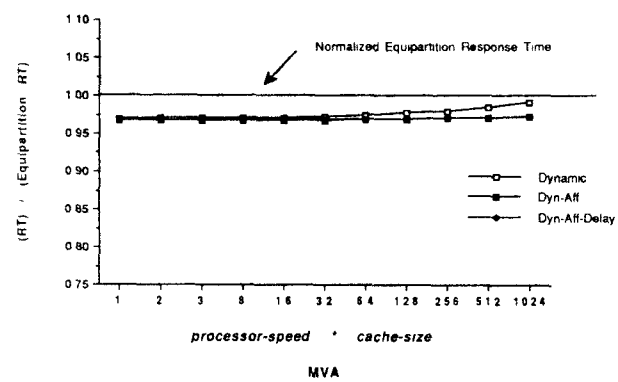


Figure 9 - Relative Resp. Times for Workload 2



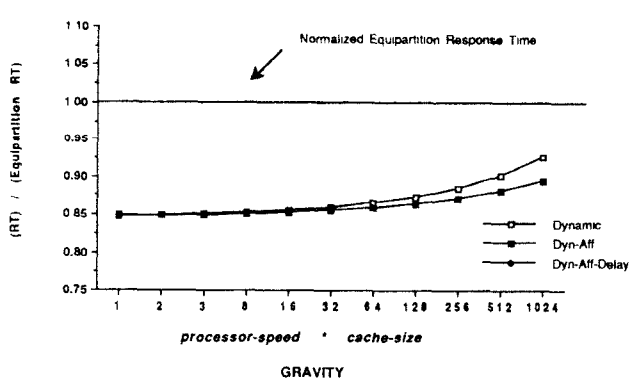


Figure 10 - Relative Resp. Times for Workload 3

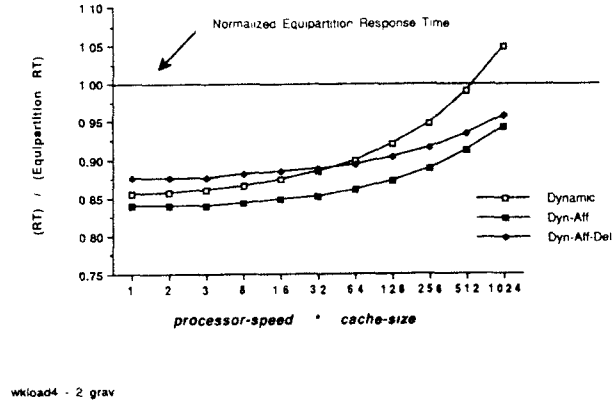
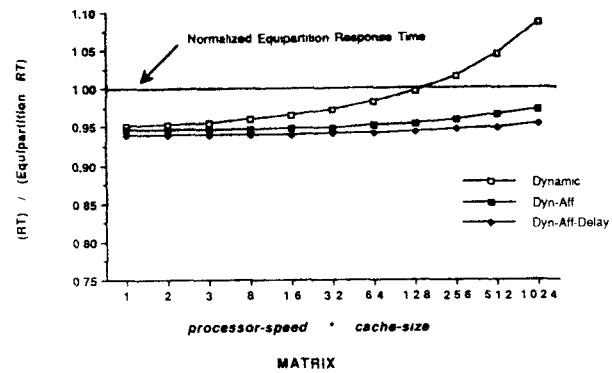


Figure 11 - Relative Resp. Times for Workload 4



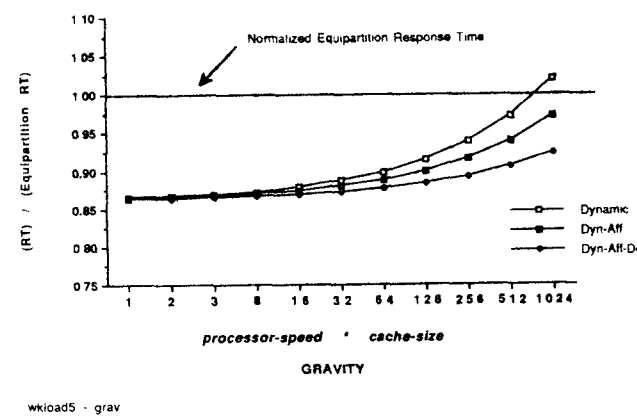


Figure 12 - Relative Resp. Times for Workload 5

• *Affinity scheduling becomes more important as machine speed increases.* This point is illustrated by Figure 8, which gives the performance of the workload containing 2 MVA jobs. The curves for Dynamic and Dyn-Aff in this figure begin to diverge, suggesting that Dynamic's "oblivious" reallocation strategy degrades application cache behavior to a point that negates any utilization gains. Since Dyn-Aff makes reallocation decisions more carefully, it continues to provide good performance. We therefore conclude that it is advisable to include affinity information in current scheduling policies because this extra consideration does not currently degrade performance, and will prove to be important in the future.

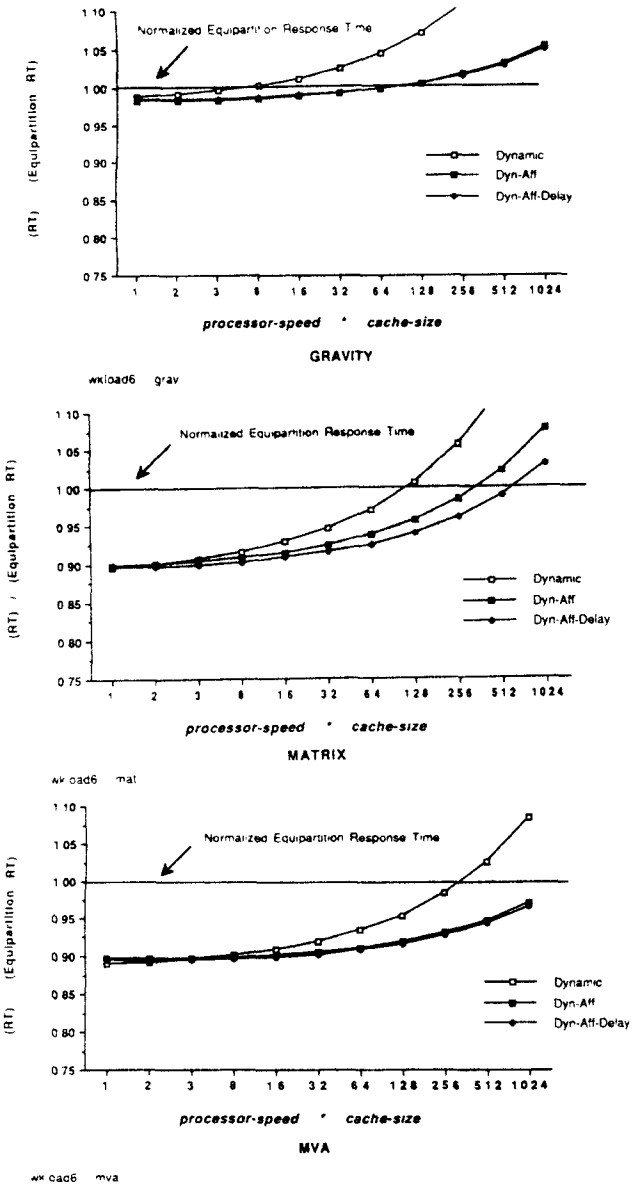


Figure 13 - Relative Resp. Times for Workload 6

- *"Yield-delay" also becomes more important as machine speed increases.* We illustrate this point using Figure 12, which shows the performance of the workload consisting of 1 MATRIX and 1 GRAVITY job. This figure shows that the difference between Dyn-Aff (the policy of choice based on the above) and Dyn-Aff-Delay becomes more pronounced with increasing machine speed. As previously noted, Dyn-Aff aggressively reallocates processors at every opportunity, while Dyn-Aff-Delay reduces needless preemptions by allowing jobs to retain unused processors for short periods of time. This not only eliminates the overhead of performing the reallocation, but also the cache effects of this reallocation that (as we have seen) begin to assume more importance.

We therefore conclude that, as in the case of affinity, it is useful to incorporate a yield-delay strategy into the allocation policy. These extra strategies cost nothing on current machines (where they also add nothing), but become more important as machine speed increases.

8. Related Work

Our conclusion regarding the relative unimportance of affinity scheduling is seemingly at odds with much previous work on this topic; we now address this issue. Section 8.1 provides some background necessary for placing our work in the proper context. In Section 8.2, we compare our results to specific work by other researchers. Section 8.3 summarizes the comparison.

8.1. Background

Evaluating affinity scheduling involves first quantifying the cache effects of processor reallocation, and then determining the influence of these effects on the choice of scheduling discipline. Our measurements of the former (Section 4) are in close agreement with those of other researchers. In spite of this similarity, our conclusions are quite different.

As stated previously, we examine only space sharing policies, while other researchers have (so far) typically studied time sharing policies augmented with affinity considerations. Time sharing policies *inherently* induce poor cache behavior for several reasons. First, they typically allocate jobs a larger amount of processors than do space sharing policies, but for shorter periods. [Tucker & Gupta 89, Gupta et al. 91, McCann et al. 91] have shown that this is inadvisable because contention effects (cache invalidations, synchronization delays) reduce the effectiveness of the additional processors. Further, time sharing policies achieve fair allocation by rotating processors among jobs. This maximizes the adverse consequences of multiprogramming, since jobs needlessly overwrite each others' cache contexts; space sharing policies reduce this effect by allowing jobs to retain processors as long as they are useful (barring occasional forcible preemption to enforce fairness).

Finally, time sharing policies reallocate based on an arbitrarily chosen quantum that usually has little to do with job behavior. A potentially larger number of context switches are therefore involuntary (caused by quantum expiry). For such switches, the amount of data necessary across reallocations may be large, since a job preempted involuntarily will need to complete the interrupted computation, using the same data. Under space sharing policies, conversely, most reallocations are initiated by the jobs themselves (as they require either more or fewer processors). A large percentage of reallocations thus result from jobs voluntarily relinquishing processors as they reach the end of some phase of computation. Even if such jobs eventually need the yielded processors returned, the data they use at that point is likely to be different from that which they were using when the processors were ori-

ginally relinquished. Since previously cached data is useless in such cases, maintaining affinity relationships is less critical.

Space sharing policies thus eliminate much of the poor cache behavior induced by time sharing policies, and the possible impact of affinity on the former policies is therefore smaller than on the latter. With this basic difference in mind, we now discuss the work of other researchers in more detail.

8.2. Comparison to Previous Work

Our experimental work was preceded by the modeling work of [Squillante & Lazowska 89]. Using an analytic model of cache footprint behavior, and an analytic model of a multiprogrammed system and its workload, they concluded that affinity scheduling can have a pronounced effect on performance.

The main reason for the differences between our respective sets of conclusions is that our empirical observations do not agree with the assumptions in [Squillante & Lazowska 89]. They assume that a task returning to a processor will find useful data remaining in the cache even after many intervening tasks have run there. However, our measurements indicate that even a single intervening task can eject large portions of the returning task's context (Section 4). This discrepancy occurs because Squillante and Lazowska study time sharing policies, for which the reallocation interval is small. More frequent reallocation not only makes cache effects more important to response time, but ensures that tasks do not run long enough to interfere with each other significantly. Our measurements of typical reallocation intervals in the space sharing domain (row 3, Table 3) suggest they are large enough so that a) cache effects are only a small fraction of overall response time, and b) tasks are allowed to make extensive use of the cache, resulting in significant inter-task interference.

The assumption regarding data survival is also optimistic due to the fundamental difference between time sharing policies and space sharing ones, explained above. Under space sharing policies, a task will often voluntarily relinquish a processor when it reaches the end of some phase of computation. Even if the task subsequently returns to a processor where some of its old data remains, this data might not in fact be useful: the task will most likely begin some new computation that requires new data to be loaded. Thus, the assumptions made by Squillante and Lazowska, while appropriate to their time sharing domain, do not appear to hold in our space sharing domain. Our conclusions regarding space sharing policies therefore differ from the ones they reach regarding time sharing ones.

In contrast to the time sharing policies of [Squillante & Lazowska 89], the "process control" policy of [Tucker & Gupta 89] is a space sharing policy. However, although [Gupta et al. 91] subsequently evaluated various multiprocessor scheduling strategies using trace-driven simulation, affinity scheduling in the context of "process control" was not studied.[6] Such an evaluation was unnecessary, since (as mentioned previously) "process control" already provides perfect affinity scheduling by avoiding reallocations except on job arrival and termination. This relatively static policy is an appropriate choice given the workloads studied in [Tucker & Gupta 89, Gupta et al. 91] where sufficient parallelism was available to keep all processors busy at all times. Under these conditions, dynamic reallocation in response to changing parallelism is unnecessary, and the issue of considering affinity during such reallocations never arises.

On the other hand, the programs we study exhibit parallelism that varies in both amount and frequency of change. Our Dynamic policy therefore reallocates processors in response to the changing needs of jobs, reducing wasted processing power. Since this necessitates more frequent reallocations than under "process control", affinity considerations assume more importance. Our work thus extends that of Gupta et al. by evaluating affinity in the space sharing domain.

[Mogul & Borg 91] also used simulation to conclude that the cache effects of context switches significantly degraded uniprocessor program performance. Our conclusions differ from theirs due primarily to the difference between our respective environments. The uniprocessor environment studied by Mogul and Borg necessitates a time sharing discipline. As just explained, affinity is more important under these conditions: the increased number of involuntary context switches implies that restarted tasks require more data to be preserved across context switches, and priority assignment schemes that use affinity information might therefore provide better performance than do standard schemes. This analysis agrees with the results in [Mogul & Borg 91], where workloads with mostly voluntary context switches were found to be much less sensitive to the cache effects of the switches than were workloads with mostly involuntary (scheduler-driven) switches. Since our multiprocessor environment allows us to implement space sharing policies, we find affinity to have a much weaker influence, for the reasons described in Section 8.1.

[6] [Gupta et al. 91] does include an evaluation of a time sharing based affinity policy (time sliced priority scheduling with relinquishing locks). The quanta they used were large enough so that reallocations were relatively infrequent, and inter-job interference significant. This led to conclusions much like ours: affinity was found to have a positive but small effect.

8.3. Summary

Our measurements of the cache effects of reallocation match those of other researchers quite precisely. However, our fine-grained measurements (Section 4) were made in what was essentially a time sharing environment, with quantum-driven reallocation taking place at arbitrary points in a task's computation. For the reasons explained above, the space sharing policies we study reduce the magnitude of the cache penalties that might be observed in practice, as well as the impact of these penalties on job response time. Our subsequent experiments with live workloads confirm that cache effects make only a small contribution to overall response time under the disciplines we use. Thus, the fundamental difference between the behavior of time sharing and space sharing policies causes our conclusions to differ from those of other researchers (who have mainly studied the former), despite similar quantitative results.

9. Conclusions

We have addressed the influence of processor affinity on processor scheduling in multiprogrammed, shared memory multiprocessors. Such affinity is acquired by executing tasks as they bring currently used data and code into the processor-local cache. The goal of affinity scheduling is to improve performance by, whenever possible, dispatching tasks to processors for which the tasks have affinity.

Using an experimental testbed running on a Sequent Symmetry, we measured a number of workloads to obtain their basic affinity characteristics and to compare the performance of several scheduling disciplines that differ in their consideration for affinity. We then used the results of these experiments to drive a simple analytic model with which we studied affinity effects on future, faster machines.

Based on these experiments and the results of our model, we conclude that:

- *Even on current multiprocessors, the cache effects of a processor reallocation can exceed the simple path length costs.* This result is in full agreement with those of other researchers.

- *Despite this, affinity scheduling has negligible effect on performance for current multiprocessors.* The major reason for this is that current cache penalties are small in comparison to the time between processor reallocations, under even the most aggressive dynamic scheduling disciplines. The fact that we examine only space sharing policies also contributes to this effect, since such policies cause jobs to be involuntarily preempted less frequently than do time sharing policies. For this reason, the amount of data required by jobs across reallocations — and thus the influence of affinity — is potentially smaller under such policies.

- *Affinity scheduling and "yield delay" have a modest effect on future, much faster machines.* As processors become faster, proportionately more processing power is wasted if cache miss resolution is necessary. In terms of our model, the *cache-penalty* due to reallocation begins to outpace *waste* as processor speed increases, and reducing reallocations (even at the expense of some waste) assumes more importance.

- *Even on much faster machines, a scheduling policy based on dynamic reallocation of processors among jobs outperforms a more static, equipartition policy.*

Since the affinity based variants of the Dynamic policy do not hurt performance on current configurations, and are more robust with respect to increases in machine speeds and cache sizes, it appears that they are the best choice for implementation in machines of the style addressed here.

We emphasize that these conclusions do not state that cache effects are unimportant in this class of machine, but rather that they have only limited influence on the kernel processor scheduling discipline. It is clear that cache effects can have a significant effect on how applications should be programmed, and that cache considerations will become more central to application programming of these machines as they become faster. Part of our continuing work is an investigation of these cache effects on the design of software layers above the kernel, e.g., the user-level thread package.

Acknowledgements

Derek Eager, Bill Joy, Mark Squillante, and the referees — particularly John Hennessy and Ed Lazowska — provided valuable comments on the contents and presentation of this paper.

References

[Anderson et al. 91] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The Interaction of Architecture and Operating System Design. *Proc. 4th Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Sys.* (April 1991).

[Barnes & Hut 86] J. Barnes and P. Hut. A Hierarchical *O(NlogN)* Force-Calculation Algorithm. *Nature 24* (1986), pp. 446-449.

[Bershad et al. 88] B.N. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice & Experience 18*, 8 (August 1988), pp. 713-732.

[Birrell 89] A.D. Birrell. An Introduction to Programming with Threads. DEC System Research Center (Jan. 1989).

[Fox et al. 88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker. Solving Problems on Concurrent Processors, Volume I. Prentice-Hall (1988).

[Gupta et al. 91] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods of the Performance of Parallel Applications. *Proc. 1991 ACM SIGMETRICS Conf. on Meas. and Mod. of Comp. Sys.* (May 1991).

[Jouppi 90] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proc. 17th Intl. Symp. on Computer Architecture* (May 1990).

[Karlin et al. 91] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for Shared-Memory Multiprocessors. To appear *Proc. 13th ACM Symp. on Oper. Sys. Princ.* (Oct. 1991).

[Lam et al. 91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *Proc. 4th Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Sys.* (April 1991).

[Lenoski et al. 90] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. *Proc. 35th IEEE Comp. Soc. Intl. Conf. -- COMPCON 90* (Feb. 1990).

[Lo & Gligor 87] S.-P. Lo and V.D. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. *Proc. 7th Intl. Conf. on Dist. Comp. Sys.* (Sept. 1987).

[Lovett & Thakkar 88] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. *Proc. 1988 Intl. Conf. on Par. Proc.* (Aug. 1988).

[McCann et al. 91] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Scheduling Policy for Multiprogrammed, Shared Memory Multiprocessors. Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington (revised Feb. 1991).

[Mogul & Borg 91] J. C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. *Proc. 4th Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Sys.* (April 1991).

[Ousterhout 90] J.K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? *Proc. Summer 1990 USENIX Conference* (June 1990).

[Squillante & Lazowska 89] M.S. Squillante and E.D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. Technical Report 89-06-01, Department of Computer Science, University of Washington. To appear *IEEE Trans. on Parallel and Dist. Systems.*

[Thiebaut & Stone 87] D. Thiebaut and H. S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5 (Nov. 1987), pp. 305-329.

[Tucker & Gupta 89] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proc. 12th ACM Symp. on Oper. Sys. Princ.* (Dec. 1989).

[Wang et al. 89] W.-H. Wang, J.-L. Baer, and H. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. *Proc. 16th Intl. Symp. on Computer Architecture* (June 1989).