JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 24, 139–151 (1995)

Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors

JOSEP TORRELLAS,[1] ANDREW TUCKER, AND ANOOP GUPTA

Computer Systems Laboratory, Stanford University, Stanford, California 94305

As a process executes on a processor, it builds up state in that processor's cache. In multiprogrammed workloads, the opportunity to reuse this state may be lost when a process gets rescheduled, either because intervening processes destroy its cache state or because the process may migrate to another processor. In this paper, we explore affinity scheduling, a technique that helps reduce cache misses by preferentially scheduling a process on a processor where it has run recently. Our study focuses on a bus-based multiprocessor executing a variety of workloads, including mixes of scientific, software development, and database applications. In addition to quantifying the performance benefits of exploiting affinity, our study is distinctive in that it provides low-level data from a hardware performance monitor that details why the workloads perform as they do. Overall, for the workloads studied, we show that affinity scheduling reduces the number of cache misses by 7–36%, resulting in execution time improvements of up to 10%. Although the overall improvements are small, modifying the operating system scheduler to exploit affinity appears worthwhile—affinity has no negative impact on the workloads and we show that it is extremely simple to add to existing schedulers. © 1995 Academic Press, Inc.

1. INTRODUCTION

The performance of shared-memory multiprocessors can be seriously limited by the long latency of memory accesses. In current high-performance machines, the penalty of a cache miss is often several tens of cycles. Consequently, it is important to develop techniques to reduce the number of cache misses suffered by the workloads running on these machines.

An interesting and common class of workloads for these machines is multiprogrammed workloads. Because these workloads generally contain more processes than there are processors in the machine, there are two factors that increase the number of misses. First, several processes are forced to time-share the same cache, resulting in one process displacing the cache state previously built up by a second one. Consequently, when the second process runs again, it generates a stream of misses as it rebuilds its cache state. Second, since an idle processor simply selects the highest priority runnable process, a given process often moves from one CPU to another. This frequent migration results in the process having to continuously reload its state into new caches, producing streams of cache misses.

To reduce the number of misses in these workloads, processes should reuse their cached state more. One way to encourage this is to schedule each process based on its affinity to individual caches, that is, based on the amount of state that the process has accumulated in an individual cache. This technique is called *cache-affinity scheduling*.

Cache affinity scheduling has been the subject of several previous studies. However, few of them have conducted a detailed analysis of the benefits of affinity scheduling on a real multiprocessor with real workloads. As a consequence, they do not consider many of the subtle issues and complex interactions of workload, architecture, and scheduler discussed here.

Squillante and Lazowska [12] measured response time under different affinity-based scheduling policies. Their results suggest that affinity scheduling provides substantial benefits. Their results, however, are based on analytical calculations with simple machine and application models, rather than a real implementation. Another analytical study was performed by Squillante and Nelson [13], who studied the effects of migrating processes among processors. They concluded that unconditionally fixing processes onto processors, while allowing processes to reuse more cache state, causes too much load imbalance. This load imbalance then results in fairness problems and idle time.

Mogul and Borg [9] used address traces of a variety of real workloads to study the potential for cache reuse. They found cache reload overheads of up to 8% of the execution time, depending on the workload. In contrast to our studies, however, their study was confined to process switching on a single processor and did not include operating system activity. Their conclusions were based on the use of a fairly short 16-ms time quantum. Simulations driven by application-only traces of real multiprocessor applications by Gupta *et al.* [4] with a short time quantum (10 ms) point to small benefits of affinity. However, I/O and other load variations were not considered. A study by Devarakonda and Mukherjee [2] evaluated the performance of a real implementation of cache affin-

[1] Current address: Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, 465 Computer and Systems Research Laboratory, 1308 West Main Street, Urbana, IL 61801.

ity on a multiprocessor, but used synthetic workloads. Their results suggest that implementation issues and workload choice can have a large impact on the measured performance of affinity scheduling, but the synthetic nature of the workloads means that the complex facets of real applications were missed.

One study that did measure real applications on a real system, by Thakkar and Sweiger [14], looked only at the performance of a database system under an extreme form of affinity. The authors studied a database application running in a 12 to 24 processor machine. They found a significant amount of cache state lost due to cache migration, and simply attaching processes to processors improved performance significantly. While our results disagree on the effectiveness of attaching processes to processors, the difference may be due to the different database implementations or to the larger number of processors they used, which cause bus contention. Bus contention increases miss latency, increasing the potential for gains from affinity.

Finally, Vaswani and Zahorjan [15] also used real applications and a real implementation of cache affinity. They studied a system under *space sharing*, a sophisticated scheduling scheme that partitions processors among applications. They found that, due to the relative infrequency of process preemption, the benefits of affinity scheduling were minimal. However, this conclusion is the result of using solely scientific workloads and scheduling them under space sharing. Even without affinity, this scheduling strategy rarely preempts processes (about once every 400 ms) when processes do not block like in scientific workloads. Therefore, the small gains for affinity scheduling with scientific workloads under space sharing are understandable. However, a study of workloads like databases, where processes block so frequently that they would rarely exhaust a time quantum much longer than 5 ms, may result in a different conclusion on the impact of affinity. Further, nonscientific workloads will often require significant modifications to run under space sharing.

Although Vaswani and Zahorjan study the effect of affinity on a space-sharing system, the vast majority of multiprocessors use a time-sharing system. It is still not known how effectively multithreaded UNIX kernels using time-sharing scheduling on small-scale multiprocessors can exploit affinity scheduling. This paper addresses this question, using data from a hardware performance monitor in a real machine. We show that affinity scheduling is easy to implement and does improve the performance of most of the workloads—we achieve gains of up to 10% in our machine. To understand the performance gains and generalize the study, we model the way workload characteristics affect a workload's ability to benefit from cache affinity. We base our evaluation on data from a hardware performance monitor in a 4-CPU high-performance multiprocessor running real scientific, software development, and commercial applications grouped in workloads. We show that affinity scheduling is preferable to two other means that have been used to increase the reuse of cache state, namely, attaching processes to processors and extending the time quantum. Finally, we also consider more complex ways of implementing affinity scheduling and conclude that they produce further performance improvements for workloads with numerous processes.

This paper is organized as follows. Section 2 describes our experimental environment and workloads. Section 3 characterizes the workloads from the standpoint of cache affinity. Section 4 describes the baseline affinity function. In Section 5, we discuss the results, first focusing on the basic results of affinity scheduling (Section 5.1), then on attached scheduling and increasing the time quantum (Sections 5.2 and 5.3), and finally on more complex affinity functions (Section 5.4). We conclude in Section 6.

2. EXPERIMENTAL ENVIRONMENT AND WORKLOADS

The results presented in this paper are based on a Silicon Graphics POWER Station 4D/340 [1], a 33 MHz bus-based multiprocessor with four CPUs. Each processor is a MIPS R3000 with a 64 Kbyte I-cache and a two-level D-cache: the first level is 64 Kbytes and the second level is 256 Kbytes. The latency of a cache miss serviced from the second-level cache is about 15 cycles; if serviced from main memory the latency is about 35 cycles. All caches are direct-mapped and have 16-byte blocks. To accurately study the execution of applications, we use a hardware performance monitor that measures time with a 60-ns granularity and has the ability to trace cache misses. The operating system used is a variant of UNIX System V called IRIX. It has a rescheduling overhead of approximately 50 μs and a nominal scheduling time quantum of 30 ms.

To evaluate affinity scheduling, we have chosen five parallel applications with varying potentials for increasing the reuse of cache state. The applications, shown in Table I, are used in real life and belong to the scientific, software development, and commercial domains. Except for *Pmake* and *Oracle*, all applications are run with four processes. Since *Pmake* and *Oracle* have a large I/O component, they run more efficiently when the number of processes is larger than the number of processors. For *Oracle*, we chose the number of processes for which the application is fastest.

The scientific applications are written in C and use the synchronization and sharing primitives provided by the Argonne National Laboratory macro package [7]. These three applications have different synchronization and sharing characteristics. For example, while processes in *Matrix* rarely synchronize, processes in *Mp3d* synchronize frequently, and *Cholesky*'s processes display an intermediate behavior. *Mp3d* and *Matrix* show a low invalidation traffic. In *Mp3d*, the large size of the working set is responsible for most of the misses and ensures that pro-

TABLE I
Parallel Applications Used in the Workloads Measured

Application	Domain	Description
Matrix	Scientific	Blocked multiplication of two 512 × 512 matrices using 64 × 64 blocks.
Cholesky	Scientific	Cholesky factorization of a sparse matrix with 10K rows and 200K nonzeros in its lower triangle [11].
Mp3d	Scientific	Simulation of a rarefied hypersonic flow of 50,000 particles over 15 iterations [8].
Pmake	Softw. develop.	8-process parallel compilation of 24 C files, averaging 460 lines of code each.
Oracle	Commercial	Oracle database [10] running a memory-resident 12-process TP1 benchmark [3].

cessors do not access data in other processors' caches. In *Matrix*, different processors frequently access the same data, but a large fraction of it is read-mostly. In *Cholesky*, however, there is some invalidation traffic caused by processors write-sharing data.

Turning to the other two applications, we see that, while *Pmake*'s processes synchronize infrequently, *Oracle's* need to synchronize regularly to keep the index structures consistent. Neither workload causes significant invalidation traffic. However, while different processors rarely share the same data in *Pmake*, processors may share read-mostly data in *Oracle*.

We group the applications into workloads of concurrently-executed applications with different flavors, namely, scientific workloads (*Matrix* + *Matrix*, *Matrix* + *Cholesky*, and *Matrix* + *Mp3d*), software development workloads (*Pmake* + *Pmake*), mixtures of the two (*Pmake* + *Matrix* and *Pmake* + *Cholesky*), and a commercial workload, an Oracle database (*Oracle*). In this paper, when we refer to cached data, we include both instruction and data words for the application and operating system. In the following section, we characterize these workloads from the standpoint of cache affinity.

3. POTENTIAL BENEFITS OF CACHE AFFINITY

There are numerous factors that determine to what degree a workload benefits from cache affinity. One of the purposes of this paper is to determine how characteristics inherent to a workload affect its potential to benefit from techniques that exploit affinity. In later sections, we will look at the ability of different techniques to fulfill this potential. In this section, we first look at the characteristics of a workload's constituent applications that affect its ability to benefit from cache affinity, and then at how the interactions between applications in a workload affect the benefit. Finally, we build a qualitative model of the benefits based on these characteristics.

3.1. Application Characteristics

There are three main characteristics that determine the potential for a process to exploit cache affinity: the amount of cached state the process reuses, the length of time it executes continuously without releasing the processor, and the reason for eventually releasing the processor. We now discuss these characteristics and their effect on gains from affinity.

3.1.1. Amount of Cached State Reuse. When a process runs, its interaction with the cache can be in one of two modes. When the process is scheduled on the processor, it is in a *reload* or *transient* phase, building up a working set of state in the cache. Once the bulk of the working set has been loaded into the cache, it reaches a *steady state* where cache misses are typically much less frequent. The goal of any technique that exploits affinity is to reduce the amount of data that needs to be fetched into the cache by avoiding transient phases as much as possible. Thus, the larger the difference in miss rate between the transient- and steady-state phases for an application, the larger the potential of cache affinity for the application.

What determines the difference between the misses in these two modes is the amount of cached data that the process reuses. To determine the magnitude of such reuse, we would like to measure the amount of data accessed within an interval where the process runs without releasing the CPU that was also accessed in the previous such interval by the process. Such intervals we call *dispatch intervals*. A dispatch interval starts or finishes when the OS inspects the run queue, independently of whether or not the same process is scheduled. We estimate the data reuse by comparing the number of misses within one dispatch interval with and without flushing the caches before the interval begins. We determine the difference between these two sets of misses for the dispatch intervals in the program and compute the average value. Our measurements show that two scientific applications, *Matrix* and *Cholesky*, reuse their cached state. For *Matrix*, the difference between the number of misses with and without flushing is about 3000, and for *Cholesky* it is about 1000. These cache reload misses imply that every flush causes 48 and 16 Kbytes, respectively, to be loaded into the cache. This suggests that both applications have good potential for exploiting cache affinity. This result is expected, since both applications are carefully blocked to minimize steady-state misses.

Oracle processes also reuse cache state, suffering 2,000 extra misses every cache flush. However, the processes in *Pmake* and *Mp3d* barely reuse their cache state. Cache flushing causes 500 and 0 extra misses respectively. In the former case, this is to be expected for an interactive application, but the latter result is somewhat surprising. This effect is caused by *Mp3d* sweeping through a large data set. The result is that the cache is continuously refilled and the miss rates in transient and steady states are the same. The small amount of cached data reuse in these two applications suggests that they will not benefit much from cache affinity.

3.1.2. Duration of the Dispatch Interval. Even if an application has a large amount of reused data, the gains from affinity may still be limited by the length of the dispatch intervals of its processes. The dispatch interval length affects the potential benefits from affinity because it controls the proportion of time an application spends in the steady state. If the dispatch interval is long, the time in the transient state will be small compared to the time spent in the steady state, and thus the loss of performance due to the extra misses will be small.

Table II shows the distribution of the dispatch interval durations for the applications. These distributions were used to compute the amount of cache state reuse in Section 3.1.1. We see that *Matrix* has long dispatch intervals, usually exhausting the 30 ms of the time quantum. Since it takes about 3 ms to reload the 3000 reload misses measured in the previous section, the benefits of affinity for *Matrix* will be relatively small. A similar conclusion applies for *Cholesky*. *Mp3d* has shorter dispatch intervals but it does not reuse cached state. Finally, *Pmake* and *Oracle* have relatively short dispatch intervals. Since processes in *Oracle* have a sizable cache-reload transient for these short dispatch intervals, they could benefit from cache affinity.

3.1.3. Reason for Terminating the Dispatch Interval. The third process characteristic that we consider is the reason for terminating a dispatch interval. We distinguish between those that block the process and those that do not. In the former case, since the process is blocked, the process is more likely to take long to run again and it is therefore less likely to become runnable in time to reuse its state left in the cache. Therefore, the potential for cache affinity is smaller.

A dispatch interval may be terminated by the expiration of the time quantum or by a number of events initiated by the process. These include blocking on a semaphore, unsuccessfully trying to acquire a lock, issuing a system call, suffering a TLB fault, and other less frequent events. Of the possible causes of termination, only semaphore blocks actually block the process, that is, remove the process from the run queue. Semaphore blocks occur mainly because of I/O activity. Other less frequent causes of semaphore blocking are operating system lock contention and system calls to block a process.

TABLE III
Breakdown of Events That Terminate Dispatch Intervals

Cause	*Matrix* (%)	*Cholesky* (%)	*Mp3d* (%)	*Pmake* (%)	*Oracle* (%)
End of quantum	94.6	46.2	26.2	21.5	1.1
Semaphore block	2.2	2.3	10.1	47.5	39.2
Synchronization	0.0	47.9	60.7	0.0	21.9
System call	2.2	2.1	2.0	14.8	37.6
TLB fault	0.0	0.2	0.2	10.3	0.0
Other	1.0	1.3	0.8	5.9	0.2

The causes for dispatch interval termination are presented in Table III. As expected, scientific applications rarely block. For *Matrix*, 95% of the dispatch intervals end due to time quantum expiration. *Cholesky* has a high percentage of time quantum expirations, but it also has an equally high percentage of dispatch intervals interrupted due to failure to acquire a lock. *Mp3d* is dominated by this synchronization effect. The only two applications where semaphore blocking accounts for a large fraction of the dispatch interval terminations are *Pmake* and *Oracle*.

3.2. Workload Characteristics

When we run multiple processes concurrently as a workload, interactions among the processes can affect the potential benefit of exploiting cache affinity. There are two major interactions: the way dispatch intervals of different processes interleave, and the amount of cache state displacement that this interleaving causes. We consider each in turn.

TABLE II
Distribution of Dispatch Interval Duration for the Applications

	Distribution of the dispatch interval duration (% of total intervals)						
Application	1–5 ms	5–9 ms	9–13 ms	13–17 ms	17–21 ms	21–25 ms	25–30 ms
Matrix	3	1	0	1	1	1	93
Cholesky	19	9	5	5	5	8	49
Mp3d	17	9	8	5	14	11	36
Pmake	40	11	8	6	7	8	20
Oracle	67	32	1	0	0	0	0

Note. Because of their irrelevance, dispatch intervals smaller than 1 ms have been eliminated. No dispatch interval is longer than 30 ms.

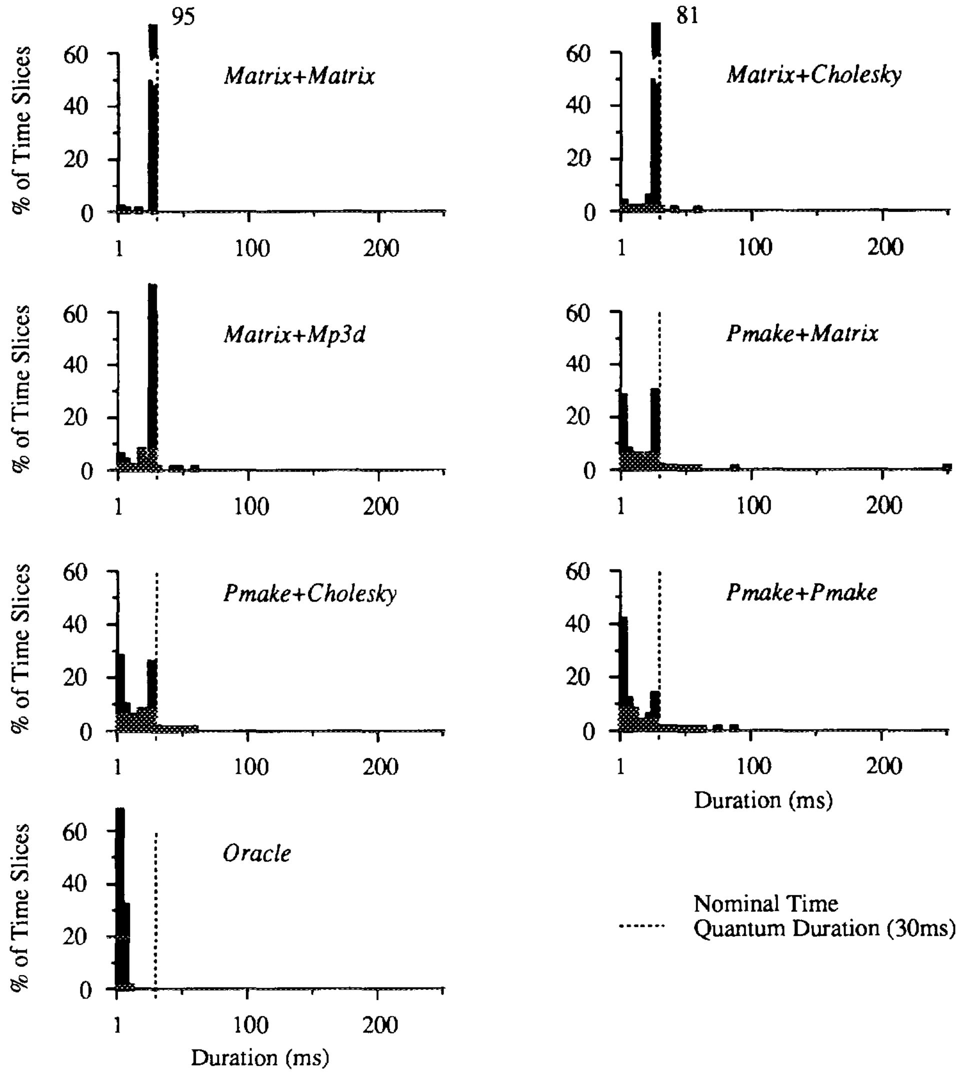

FIG. 1. Distribution of the duration of the effective time slices under UNIX scheduling for the workloads studied. The charts show what fraction of the effective time slices last a given amount of time.

3.2.1. Effective Time Slice. The amount of time a process runs on a processor without intervening processes running is called an *effective time slice*. An effective time slice may contain several dispatch intervals if the same process is rescheduled several dispatch intervals in a row on the same CPU. If this occurs frequently, the potential for benefit from cache affinity will decrease, since applications will already be exploiting cache state by running on the same CPU for a long time. This situation is not frequent in traditional schedulers when the number of processes is twice or more the number of processors. Instead, the processes of the different applications tend to interleave well their use of the CPUs. To illustrate this, Fig. 1 shows the distribution of the duration of the effective time slices for our workloads on the standard UNIX scheduler. The figure shows what fraction of the effective time slices last a given amount of time. From the figure, we see that, although there are a few times when enough processes block that the remaining running processes can run for multiple dispatch intervals, this is rare. In most cases, the histograms of the effective time slices are similar to the superimposition of the histograms of the dispatch intervals for the constituent applications presented in Table II. This suggests that there is still potential for cache affinity to be exploited in these workloads.

3.2.2. Effect of Intervening Applications. When several processes share the same cache, they displace each other's state from the cache. This effect decreases data reuse in the workload and therefore increases the potential of cache affinity. The larger the fraction of the cache displaced by an intervening process, the higher the potential of cache affinity. We approximately estimate the amount of cache displacement caused by an application by the total number of I + D misses it suffers in an average effective time slice that started with flushed caches.

TABLE IV
Summary of Process and Workload Characteristics That Determine the Potential of Exploiting Cache Affinity

Workload	Misses in cache reload (thousands)	Median eff. time slice (ms)	Frequent process block?	Misses as intervening application (thousands)	Potential affinity benefit
Matrix + *Matrix*	3 + 3	30 + 30	N + N	6 + 6	mod. + mod.
Matrix + *Cholesky*	3 + 1	30 + 25	N + N	6 + 5	mod. + low
Matrix + *Mp3d*	3 + 0	30 + 20	N + N	6 + 18	mod. + none
Pmake + *Matrix*	0.5 + 3	5 + 30	Y + N	1.5 + 6	low + low
Pmake + *Cholesky*	0.5 + 1	5 + 25	Y + N	1.5 + 5	low + low
Pmake + *Pmake*	0.5 + 0.5	5 + 5	Y + Y	1.5 + 1.5	low + low
Oracle	2	3	Y	2.5	mod.

Note. "mod." stands for "moderate."

Our results show that *Mp3d* suffers as many as 18,000 misses, effectively wiping out the second-level data cache. *Cholesky* and *Matrix* each cause about 5000 and 6000 misses respectively, wiping a large part of the first-level data cache. Finally, the other applications have a smaller effect because their effective time slices are smaller. For example, *Pmake* misses 1500 times and *Oracle* misses 2500 times.

3.3. *Summary*

With the above data, we can get an accurate picture of what workloads can benefit from techniques that exploit cache affinity. The most promising workloads are those whose processes (1) are costly to reload because of the large amount of reused data, (2) execute for short effective time slices, (3) block infrequently or not at all, and (4) are interleaved with other processes that replace a large part of the cache when executed.

Table IV summarizes the values of the characteristics discussed in this section for our workloads. In the last column, we qualitatively evaluate the potential of the workload. Let us first consider *Matrix* + *Matrix*. We see that its processes are costly to reload (column 2), do not usually block (column 4) and, as suggested by the number of misses in column 5, a significant amount of a process' cached state is displaced when an intervening process runs. However, the gains will be limited because processes run for an effective time slice as long as 30 ms (column 3). In fact, we can compute a ceiling on the possible gains with the ratio between the cost of reloading one process and the effective time slice. The former is the cost of 3000 misses, which is approximately 3 ms. The latter is 30 ms. The potential, therefore, is about 10%. *Matrix* + *Cholesky* is similar, with even lower potential for *Cholesky* due to its lower cache reload cost.

Turning to *Matrix* + *Mp3d*, we see that the potential for *Matrix* is slightly higher than in the previous workloads because *Mp3d* displaces more cache state. However, *Mp3d* has no potential for benefit because its processes have no cache reload cost. Similarly, *Pmake* has little chance to benefit because its processes have a low cache reload cost and block frequently. The benefits of other applications paired with *Pmake* (*Matrix* and *Cholesky*) will also be limited since *Pmake* does not replace much cache state (column 5). Finally, *Oracle* processes are fairly costly to reload and run for a short effective time slice. However, they frequently block on I/O and they are interleaved with processes that displace little cache state.

In summary, we believe that due to the large number of factors that can limit the potential benefits of an application for exploiting cache affinity, few workloads are able to benefit greatly. In our workload set, the only cases with even moderate potential are *Matrix* + *Matrix*, *Matrix* + *Cholesky*, and *Oracle*.

4. IMPLEMENTATION OF AFFINITY SCHEDULING

To achieve the potential for benefit from affinity described in Section 3, we need an implementation of affinity scheduling that is effective, has low overhead, and does not raise the possibility of starvation or loss of response time. Our method is to modify the existing process priority scheme of a UNIX scheduler. We first describe the existing system and then discuss the modifications required to add affinity scheduling and their effect.

In the standard UNIX scheduling system, runnable processes are placed in a single global run queue, ordered by a single number denoting their scheduling priority. A processor selecting a process to run picks the highest priority process from the queue. The priority of a process is based in large part on its past CPU usage. As a process accumulates CPU time, its priority is reduced. This gives processes that frequently block more chances to run. The accumulated CPU time is periodically decayed to a fraction of its former value, however, so that long-running processes are not completely starved in favor of newer processes. The different processes of a parallel program are scheduled independently.

The exact algorithm is as follows. The priority, *prio*, of a process is defined to be

$$prio = base + \frac{cpu}{2}.$$

A low value of *prio* designates a high priority to run. *base* can be assumed to be equal for all processes we will be considering. *cpu* is a measure of the accumulated CPU time; it is initially 0 for a new process and is incremented whenever a timing interrupt (raised every 10 ms on our machine) is received while the process is running. Finally, the value of *cpu* is decayed periodically to put a limit on the decrease of priority and avoid starving long-running processes.

To add affinity to the existing system, we temporarily raise the priorities of processes that are "attractive" from the standpoint of affinity scheduling when searching the run queue. We make two changes, one to minimize process migration, and the other to increase the effective time slice of processes. The first change consists of subtracting a constant factor, a_p (for *processor*), from the *prio* values of processes whose most recent execution was on the processor that is searching the queue. This discourages migration between processors. The second change consists of subtracting another constant, a_t (for *time*), from the *prio* value of the process that has just finished executing on the processor that is searching the queue. This encourages processes to run for consecutive dispatch intervals and therefore minimizes the displacement of cached state by intervening processes. Both adjustments are just for the purpose of scheduling at that moment, and the priorities relapse to their normal values after the processor has selected a process to run.

Given the priority algorithm used in our machine, we can compute the impact of a given value of the constants. For example, with the a_t constant, a process reaches an effective time slice of roughly $40 * a_t$ ms. In addition, with the a_p constant, a process will only migrate when it receives $20 * a_p$ ms less processor time than other processes within a decay interval.

We emphasize that the implementation requires only minor modifications to the existing scheduler. It is also very efficient, since it only involves priority comparisons between at most three processes. In fact, given minor changes to the run queue structure, we can examine these processes without searching the run queue. We also note that, with our algorithm, there is no risk of unfairness or starvation since the normal priority system is still in place. Processes that are executed will have their priorities decreased by aging as they accumulate CPU time. Eventually, the priorities of these processes will be low enough such that the priority boost given by affinity will not prevent the other processes from running. In addition, new and I/O-bound applications, which have little accumulated CPU time, will normally have high enough priority to be able to overcome priority boosts given to other processes by affinity, and thus run as soon as they become runnable.

A more complex approach to affinity scheduling would involve adjusting the priority of a process as a function of the time since the process last ran on the processor that is searching the queue. While we would like to initially explore the simpler and more efficient approach described above, we will also investigate this more complex approach in Section 5.4.

5. PERFORMANCE RESULTS

In this section, we evaluate the performance gains accomplished by exploiting cache affinity. First, we present the base results, obtained by using affinity scheduling as described in Section 4. We then evaluate two alternative ways of exploiting cache affinity, namely attaching processes to processors and increasing the time quantum of the machine. Finally, we explore a more complex affinity scheduling scheme.

5.1. Base Results

This section discusses the performance of the affinity function described in Section 4. We study two different degrees of affinity scheduling. First, we consider a scheduler with light affinity (*LightAff*) by setting both a_t and a_p to 6. As indicated in Section 4, the value of these constants implies a potential effective time slice of about 240 ms or 8 time quanta, and results in a process being allowed to migrate only if it has received about 120 ms less CPU time than the other processes. Second, we study a scheduler with heavy affinity (*HeavyAff*) by setting both values to 16, for an effective time slice of about 640 ms and a migration threshold of about 320 ms.

Table V compares the performance of the two affinity scheduling schemes to the performance of the default UNIX operating system used in our machine, referred to hence as the standard scheduler. The table also shows the performance of attached scheduling, which we will describe in Section 5.2. In the table, we compare both the number of misses and the execution time of the workloads. To determine the execution time of a workload, we run all its applications to completion and compute the average time the applications took to complete. In addition, the numbers that we present are the average of 5 experiments. In the following, we first discuss the main conclusions on the effect of *LightAff* and then discuss the effects of *HeavyAff*.

Our observations on the effect of affinity are as follows:

TABLE V
Workload Execution Time with Different Scheduling Disciplines

				Affinity scheduling						
	Standard scheduling			*LightAff*		*HeavyAff*		Attached scheduling		
Workload	Exec. time (s)	No. of misses (millions)	Idle time (%)	Exec. time (%)	No. of misses (%)	Exec. time (%)	No. of misses (%)	Exec. time (%)	No. of misses (%)	Idle time (%)
Matrix + Matrix	16.8	11.2	2	−6.1	−36.3	−6.8	−39.4	−1.1	−9.1	2
Matrix + Cholesky	17.3	11.1	2	−4.2	−13.4	−6.6	−18.3	+1.0	−15.7	5
Matrix + Mp3d	14.6	20.7	2	+0.5	−8.7	−3.6	−15.6	+11.0	+17.4	2
Pmake + Matrix	52.6	49.7	4	−10.3	−11.8	−2.5	−15.6	+44.1	−13.4	36
Pmake + Cholesky	43.1	46.2	10	−7.5	−6.7	−5.2	−9.7	+45.9	−6.8	39
Pmake + Pmake	60.0	70.5	21	−1.6	−6.5	−1.2	−8.0	+75.7	+1.8	54
Oracle	64.8	141.4	12	−8.0	−7.1	−7.0	−8.1	+15.6	−11.4	29

Note. The execution time and number of misses for affinity and attached scheduling are shown as a percentage change from the corresponding standard scheduling values.

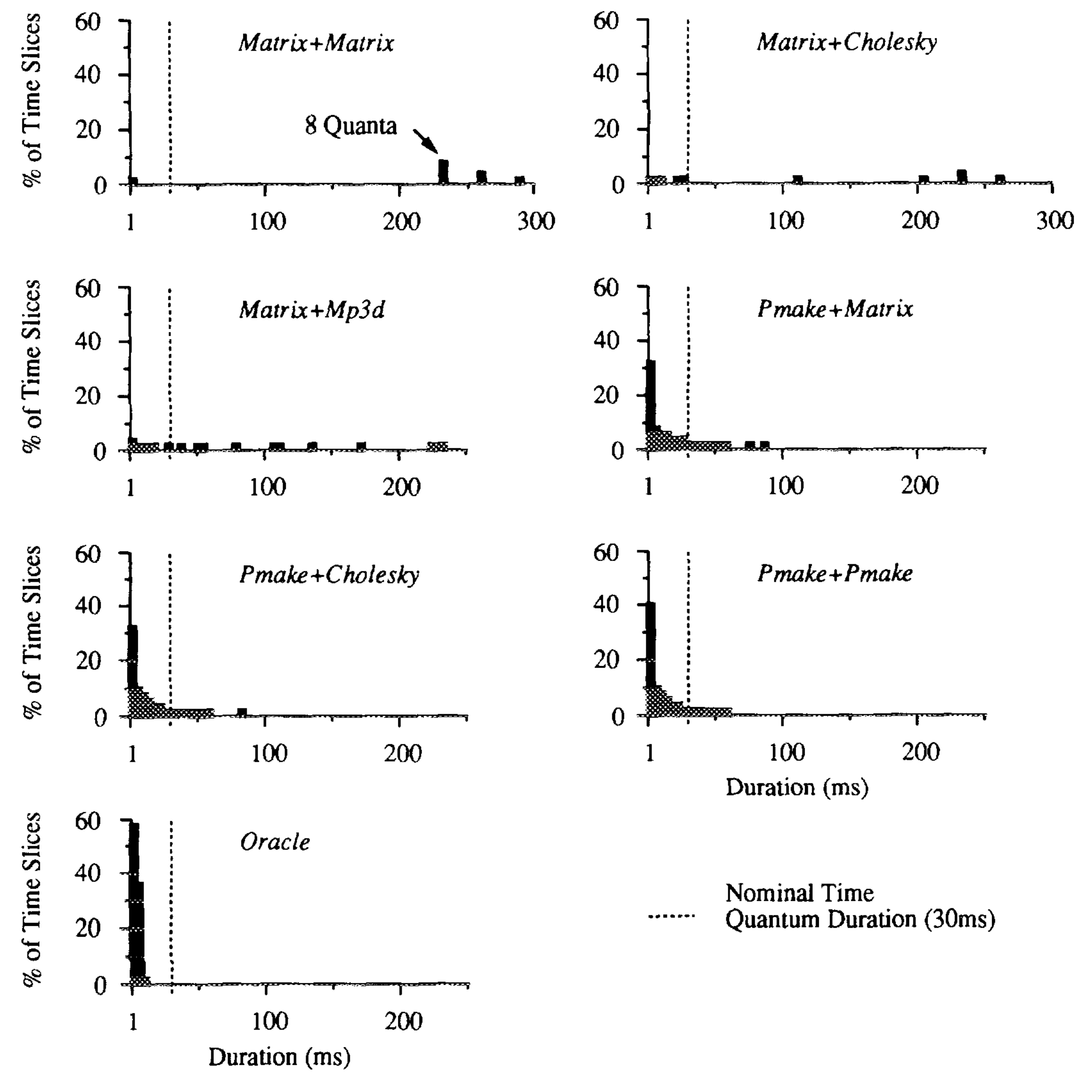

FIG. 2. Distribution of the duration of the effective time slices under affinity scheduling. The plots are normalized to the number of effective time slices under standard scheduling. Since there are fewer slices under affinity scheduling, the normalized number of slices does not add up to 100.

• *Cache affinity scheduling reduces the execution time of scientific workloads that reuse cache state.*

As noted in Section 3, *Matrix + Matrix* and *Matrix + Cholesky* have a moderate potential for gaining from affinity. The distribution of effective time slices with affinity scheduling is shown in Fig. 2. In the figure, we see that the new effective slices are much longer than before, often as long as 8 time quanta. As seen in Table V, this increase in the effective time slice translates into a removal of 36 and 13% of the misses in the two workloads respectively. However, misses account for only about 20% of the workloads' execution times. The result, as shown in Table V, is that the decrease in execution time is moderate: 6 and 4% respectively.

• *Cache affinity scheduling also reduces the execution time of frequently-blocking workloads with substantial cache state reuse.*

As we discussed in Section 3, *Oracle* also has a moderate potential for gains from affinity. *Oracle* has a large amount of reused data and processes have short effective time slices because they often block on I/O, synchronize, or issue system calls. Since the processes mostly block before being preempted, affinity scheduling will not affect the effective slice noticeably. This is verified by the data in Fig. 2, which is very similar to the data without affinity in Fig. 1. However, by encouraging processes to return to the CPUs on which they last ran, we eliminate about 7% of the misses. Although this is a relatively small number, it has a noticeable performance impact because misses account for about 60% of the time in *Oracle* and create bus contention. In addition, affinity scheduling slightly reduces the idle time. The overall effect is a performance improvement of 8%.

• *Implementation issues affect the results.*

The small changes necessary to support affinity scheduling often interact with other functions in the system. We briefly describe two cases of this.

When a process unsuccessfully tries to acquire a lock, it yields the CPU. The expectation is that the process that holds the lock will be scheduled. However, under affinity scheduling, the yielding process will have its priority boosted by affinity and therefore will be picked to run again. To avoid this, we prevent the process from being selected for this reschedule. While this solves the problem, it still allows the CPU to interleave between two processes trying to acquire a lock. If the process holding the lock does not have affinity for that CPU, it may not be able to run until the CPU for which it has affinity becomes free or until the spinning processes have aged. This effect can cause affinity to slightly slow down workloads with synchronization-intensive applications like *Mp3d,* as seen in the 0.5% drop in performance for *Matrix + Mp3d.*

Our second example involves the priority system. Our implementation of affinity scheduling interacts with the algorithm that decays the accumulated CPU time in a way such that the decay increases the priorities of long-running processes less than those of young processes. The result is that applications with short-lived processes like *Pmake* tend to accumulate execution time faster than without affinity. As a result, the *Pmake* application in *Pmake + Matrix* finishes earlier than in the nonaffinity case, reducing the average job completion time of the workload. This effect explains the 10% reduction of execution time in Table V. The overall completion time of the workload is only decreased by 5%. A similar effect occurs for *Pmake + Cholesky.*

• *Increasing the level of affinity to large values does not improve performance significantly.*

Increasing the level of affinity from *LightAff* to *HeavyAff* invariably increases state reuse, therefore reducing the number of misses. Thus, as shown in Table V, the miss reduction over the nonaffinity scheduler goes from 7–36% for *LightAff* to 8–39% for *HeavyAff.* The effect on execution time, however, is minimal. Scientific workloads like *Matrix + Matrix* and *Matrix + Cholesky* benefit slightly. However, given the already long effective time slices with *LightAff,* there is little potential for benefit beyond what *LightAff* provides. In other workloads, *HeavyAff* does not help performance, and slightly increases idle time for I/O-intensive workloads like *Pmake + Pmake* and *Oracle.* We conclude that a low level of affinity is sufficient and desirable.

• *This implementation of affinity scheduling does not introduce unfairness.*

The problem of unfairness in affinity scheduling appears if, while encouraging one process to reuse cache state, we are consistently denying another process a fair share of CPU time. Our implementation is not unfair because a favored process will eventually accumulate CPU time and therefore have its priority reduced with respect to processes that are not running. Figure 3 shows the rate of accumulation of CPU time for the applications of two workloads under *HeavyAff.* From the figure, we note that the allocation of CPU time is fair since, for a given plot, the slopes of the curves are similar.

Overall, we note that our affinity algorithm fulfills the potential described in Section 3 quite well, reducing the number of misses in the workloads by 7–36% and producing low to moderate reductions of execution time in the 1–10% range. In addition, affinity did not cause problems with load imbalance or unfairness.

5.2. Attached Scheduling

A simple alternative to our affinity scheduling algorithm is to fix processes on CPUs for the processes' entire lifetimes. This strategy tries to increase the reuse of cache state by eliminating process migration. The length of the effective time slice of the processes, however, is not affected. The "attached scheduling" columns in Ta-

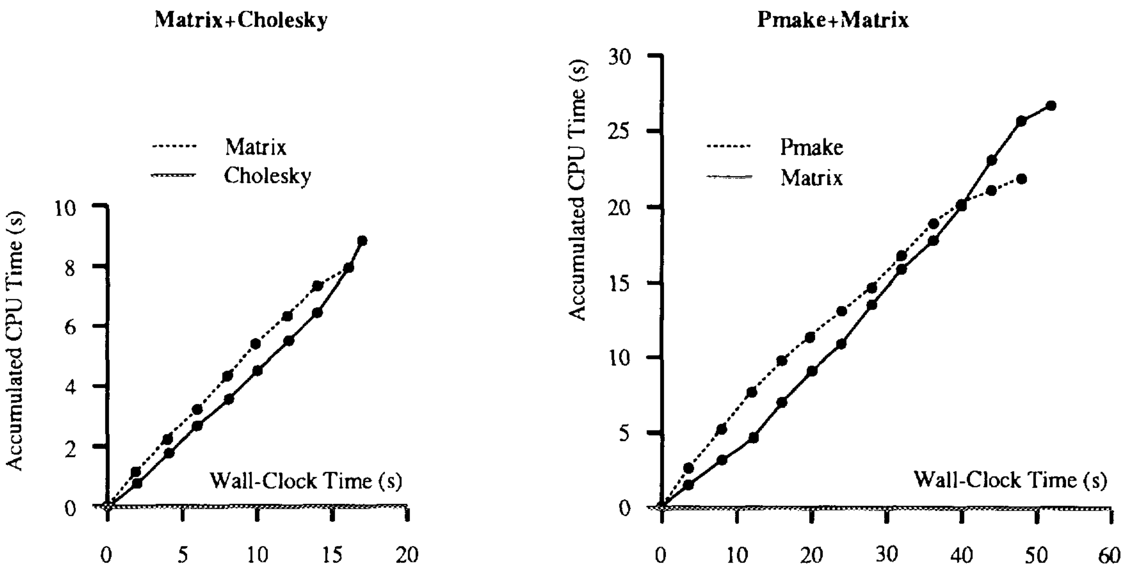

FIG. 3. Rate of accumulation of CPU time for the applications of two workloads under *HeavyAff*.

ble V show the performance of this strategy. While attached scheduling lowers the number of misses over standard scheduling, the result is that it slows down workloads. This is due to increased load imbalance in the machine. When a CPU has no processes to run, it is forced to remain idle even if another CPU has more than one process to run. Table V shows the higher average idle time in the workloads. In addition, the load imbalance may cause processes to wait longer at synchronization points, forcing extra re-schedules like in *Matrix* + *Mp3d* that result in an increased execution time and number of misses. In summary, we find that attaching processes to CPUs is not an acceptable alternative to affinity scheduling.

5.3. *Increasing the Time Quantum*

The simplest way to increase the reuse of cached state is to use a standard scheduler but increase the nominal time quantum of the machine. This strategy is the converse of attached scheduling—it tries to extend the effective time slice of processes without limiting process migration.

Figure 4 shows the performance of the standard scheduler with time quanta of 10 and 100 ms, along with the performance of *LightAff* affinity scheduling with 30-ms time quanta. The bars are normalized to the performance of the machine's original scheduler. We see most obviously that the performance with 10-ms time quanta is

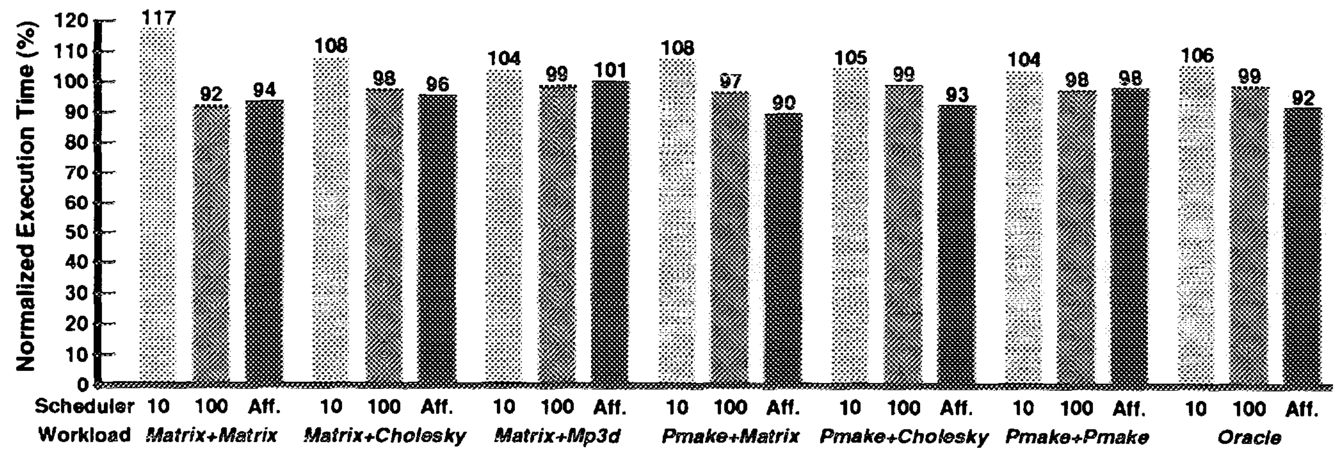

FIG. 4. Workload execution time under the standard scheduler with 10- and 100-ms time quanta and under the affinity scheduler. The bars are normalized with respect to the standard scheduler with 30-ms time quanta.

uniformly poor, taking 4–17% longer than with 30-ms time quanta. This is expected, since 10-ms time quanta produce very short effective time slices, reducing the amount of time a process spends in the steady state.

With 100-ms time quanta, we see first that all of the workloads perform better than with 30-ms time quanta. The differences vary from 1 to 8%. When comparing 100-ms time quanta with affinity scheduling, we can classify the interesting workloads into two classes. Some workloads, like *Oracle,* perform better with affinity. Others, like *Matrix* + *Matrix* and *Matrix* + *Cholesky*, perform as well with 100-ms time quanta as with affinity. In the case of *Oracle,* affinity scheduling outperforms the original scheduler by reducing process migration. It is clear, therefore, that longer time quanta will not help. This applies to all workloads where processes rarely exhaust the time quanta. For *Matrix* + *Matrix* and *Matrix* + *Cholesky*, however, affinity scheduling outperforms the original scheduler mostly by extending the effective time slice. The same effect is accomplished by using longer time quanta. As a result, the two schemes yield similar performance.

The performance of *Pmake* + *Matrix* and *Pmake* + *Cholesky* is affected by implementation issues. In these workloads, the better performance of affinity scheduling is due to the altered priority mechanism in the affinity scheduler, which gives more time to interactive applications like *Pmake*. Thus, *Pmake* finishes earlier, reducing the mean execution time of the two applications.

One concern in increasing the time quantum length is that the response time of interactive applications will drop as the interactive applications are forced to wait longer to be scheduled. However, this does not occur in our operating system. When an interactive high-priority process wakes up in response to an interrupt, it will immediately preempt a lower-priority process running on the processor that received the interrupt. A similar approach is used in 4.3BSD UNIX [5].

In summary, longer time quanta are not as powerful as affinity scheduling. Both schemes are equivalent when processes rarely block. This explains why Vaswani and Zahorjan [15] found that affinity scheduling accomplishes little for scientific applications on a system with long time quanta. However, when processes block often (*Oracle*), longer time quanta have no effect; only affinity scheduling can increase cache state reuse.

5.4. More Complex Affinity Functions

While the implementation of affinity scheduling that we studied is effective, we would now like to consider more sophisticated implementations. One general way of exploiting cache affinity is to adjust a process' priority by an amount proportional to the time since the process was last executed on the processor searching the run queue. While this approach may incur more scheduling overhead than the simpler approach we have been using, it can potentially exploit affinity in situations where the simpler approach fails. Consider the case where three processes are running on a processor. When one process blocks, the scheduler has to choose between the other two processes. With the simple algorithm we have been using, the scheduler will pick one at random; the more complex algorithm will select the process that most recently executed.

The implementation of this approach is fairly straightforward. Recall that, in the scheme described in Section 4, the *prio* values of all processes that last ran on the scheduling processor are decreased by one factor, a_p, and the *prio* value of the process that most recently ran is decreased by an additional factor, a_t. In our new system, we keep the a_p adjustment but change the a_t adjustment in the following way. All processes that last ran on the scheduling processor have their *prio* values decreased by a variable factor, equal to a_t minus the elapsed time (measured in terms of 10-ms clock ticks) since they last ran, if the difference is greater than zero. While this approach can exploit affinity in the situation discussed above, it is more costly in that it requires scanning most of the run queue comparing process priorities every time a processor schedules a process.

5.4.1. Performance. Since the change to our algorithm is only significant when there are more than two processes running on each processor, it will not change the results of most of our workloads from previous sections. The main exception is *Oracle,* where the number of processes is more than three times the number of processors. Thus, when a process blocks, the processor will have several processes that last ran on it to choose from. We have also devised two new workloads using higher levels of multiprogramming. The first is *Pmake* + *Matrix* + *Matrix*, a *Pmake* application in conjunction with two *Matrix* applications. The second is *Oracle* + *Matrix* + *Matrix*, an *Oracle* application with two *Matrix* applications. In the latter two cases, we expect to see the advantages of complex affinity when, after an interruption by short-running *Pmake* and *Oracle* processes, the right *Matrix* process resumes execution.

Figure 5 compares the performance of *Oracle*, *Pmake* + *Matrix* + *Matrix*, and *Oracle* + *Matrix* + *Matrix* under the complex affinity function with the performance under the simple affinity function and under the standard scheduler with 100-ms time quanta. *Oracle* and *Pmake* + *Matrix* + *Matrix* receive benefits from simple affinity but no further benefits from complex affinity. In *Oracle,* processes run for very short intervals before blocking. Changing the interleaving of these short-running processes with the complex affinity scheduler affects only slightly their capacity to reuse cache state over the simple affinity scheduler. The result is a small reduction of the workload's execution time.

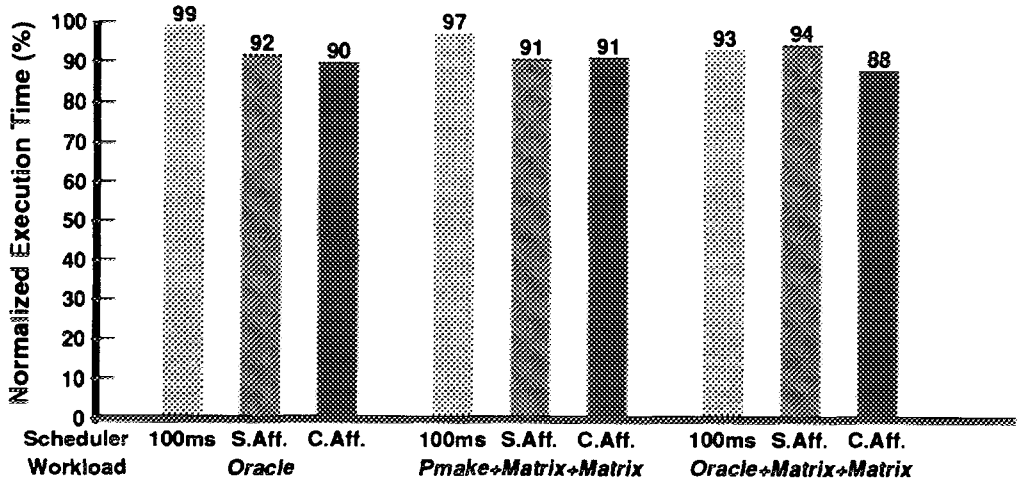

FIG. 5. Workload execution time under standard scheduling with 100-ms time quanta, under simple affinity (S. Aff.), and under complex affinity (C. Aff.). The bars are normalized with respect to standard scheduling with 30-ms time quanta.

In the case of *Pmake* + *Matrix* + *Matrix* the result is due to the way I/O is implemented on our system. While *Pmake* processes generate frequent I/O interrupts, most of these interrupts are directed at a single designated processor. Consequently, *Matrix* processes running on other processors run relatively undisturbed and the simple affinity function works nicely. In addition, the *Matrix* processes running on the designated I/O processor are interrupted too frequently to build up cache state. With little stored cache state, it does not matter which process is scheduled next. Therefore, the complex affinity function accomplishes no further gains.

With *Oracle* + *Matrix* + *Matrix*, complex affinity does make a slight difference. Like *Pmake* processes in the *Pmake* + *Matrix* + *Matrix* workload, *Oracle* processes are high-priority and run for short periods of time. Unlike *Pmake* processes, though, *Oracle* processes run on all four processors equally. In addition, they do not cause interrupts, so *Matrix* processes will always execute for at least one time quantum. The result is that when an *Oracle* process runs and then blocks, enough cache state was stored by the previously-running *Matrix* process that it becomes advantageous to run that process over other *Matrix* processes.

In summary, we see that a form of cache-affinity scheduling that adjusts priority based on the time since a process last ran may in some cases (high loads and particular workload characteristics) work better than a simpler system. The gains vary from 0 to 7%, and must be traded off against a more complex implementation.

6. CONCLUSIONS

In this paper, we explored the benefits that applications can achieve through the use of cache-affinity scheduling. We found several characteristics inherent to applications that influence their potential for benefits: the amount of cache state an application reuses, the time its processes run without interruption on a processor, and the frequency of process blocking. We also found important characteristics of the ways applications interact when run concurrently in a workload: the effective time slice of their processes and the effect of other applications in the workload in removing state from the cache. We studied realistic workloads from a variety of domains and found that, while moderate gains are possible, few have characteristics that allow for large gains from exploiting cache affinity.

We found that a simple and efficient modification to a standard UNIX scheduler, discouraging process migration and extending the effective time slice of processes, accomplishes affinity scheduling without load imbalance or fairness problems. The implementation improves the performance of our workloads by up to 10%, fulfilling most of their potential for reusing cache state. We also considered a more complex approach to affinity scheduling, and found slight advantages over the simpler approach for workloads with large numbers of processes.

We compared affinity scheduling to even simpler approaches for increasing the reuse of cache state, namely, permanently attaching processes to processors and extending the system time quantum. Neither technique is as effective as affinity scheduling. Attaching processes to processors creates load imbalance and, in addition, does not decrease the rate of context switches. Extending the time quantum accomplishes most of the gains possible for processes that rarely block (making affinity scheduling fairly useless for such workloads as show by Vaswani and Zahorjan [15]), but, unlike affinity scheduling, cannot benefit processes that run only for a short time before blocking.

Some obvious hardware factors affect our results. The Silicon Graphics machine has two data caches, a primary and a secondary, that are fairly close in size (64 and 256 Kbytes) but not in speed (1 and 15 cycles to retrieve a word). This means that cache-tuned applications will solely use the primary cache. Although the primary cache is fairly large, it is small enough that refilling it every time quantum does not severely hurt performance. If applications could effectively use larger caches, with a fill time comparable to the time quantum length, then affinity scheduling would have a larger impact.

The advantages of affinity scheduling would also be more pronounced in a machine with longer miss latencies. This is the case for the new generation of NUMA machines, where remote accesses are costly. For example, on the cluster-based DASH machine developed at Stanford [6], a remote miss takes as long as 130 cycles to be serviced. As the latency increases, the time taken to fill the cache increases, and thus affinity becomes more important. On these machines, issues of geographical affinity for a processor (remote versus local cluster memories) also come into play.

In the past, there has been some question as to the usefulness and cost of providing affinity scheduling in an operating system. Based on the findings in this paper, we conclude that affinity scheduling is a worthwhile addition. While few workloads may be able to benefit substantially from exploiting cache affinity, affinity scheduling can be easily implemented, can easily provide moderate gains for most of the workloads, and does not hurt the performance or response time of the rest.

ACKNOWLEDGMENTS

This research was funded by DARPA Contract N00039-91-C-0138. Josep Torrellas was supported by La Caixa d'Estalvis per a la Vellesa i de Pensions and the Ministerio de Educacion y Ciencia, both of Spain. Anoop Gupta was also supported by an NSF Presidential Young Investigator award. We acknowledge the help and encouragement provided by Luis Stevens. We thank the referees for their comments.

REFERENCES

1. F. Baskett, T. Jermoluk, and D. Solomon, The 4D-MP graphics superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of the 33rd IEEE Computer Society International Conference—COMPCON 88*, Feb. 1988, pp. 468–471.
2. M. Devarakonda and A. Mukherjee, Issues in implementation of cache-affinity scheduling. In *Proceedings Winter 1992 USENIX Conference*, Jan. 1992, pp. 345–357.
3. J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, San Mateo, CA, 1991.
4. A. Gupta, A. Tucker, and S. Urushibara, The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 120–132.
5. S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.
6. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 148–159.
7. E. Lusk *et al.*, *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, New York, NY, 1987.
8. J. D. McDonald and D. Baganoff, Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.
9. J. Mogul and A. Borg, The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 75–84.
10. Oracle Corporation, *Oracle Database Administrator's Guide*. Oracle Corp., Belmont, CA, 1989.
11. E. Rothberg and A. Gupta, Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, Nov. 1990, pp. 232–243.
12. M. Squillante and E. Lazowska, Using processor-cache affinity in shared-memory multiprocessor scheduling. Technical Report 89-060-01, Department of Computer Science, University of Washington, June 1989.
13. M. Squillante and R. Nelson, Analysis of task migration in shared-memory multiprocessor scheduling. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 143–155.
14. S. S. Thakkar and M. Sweiger, Performance of an OLTP application on symmetry multiprocessor system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 228–238.
15. R. Vaswani and J. 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*, Oct. 1991, pp. 26–40.

Received April 5, 1993; accepted September 3, 1993

JOSEP TORRELLAS is an assistant professor of computer science at the University of Illinois at Urbana–Champaign, where he is associated with the Center for Supercomputing Research and Development. He received a Ph.D. in electrical engineering from Stanford University in 1992. While at Stanford, he worked in the team that designed the DASH scalable shared-memory multiprocessor. Professor Torrellas' research interests focus on hardware and software techniques to improve the performance of memory hierarchies in scalable shared-memory multiprocessors. Professor Torrellas received the NSF Young Investigator Award in 1994.

ANDREW TUCKER received a B.S. in computer science at the University of California, Santa Barbara in 1984 and a Ph.D. in computer science at Stanford University in 1993. He is now employed at Sun Soft, a division of Sun Microsystems Inc., working on scheduling issues for the Solaris operating system.

ANOOP GUPTA is an assistant professor of computer science at Stanford University. Prior to joining Stanford, he was on the research faculty of Carnegie Mellon University, where he received his Ph.D. in 1986. Professor Gupta's primary interests are in the design of hardware and software for large-scale multiprocessors. Along with John Hennessy, he co-led the design and construction of the DASH shared-memory multiprocessor at Stanford. He has also worked extensively in the area of parallel applications. Professor Gupta was the recipient of a DEC faculty development award from 1987–1989, he received the NSF Presidential Young Investigator Award in 1990, and he currently holds the Robert Noyce faculty scholar chair in the School of Engineering at Stanford. He is currently on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems* and *Journal of Parallel and Distributed Computing*.