Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling

Mark S. Squillante, *Member, IEEE*, and Edward D. Lazowska, *Senior Member, IEEE*

Abstract—**In a shared-memory multiprocessor system, it may be more efficient to schedule a task on one processor than on another if relevant data already resides in a particular processor's cache. In this paper we study the effects of this type of processor affinity. We make the observation that tasks continuously alternate between executing at a processor and releasing this processor due to I/O, synchronization, quantum expiration, or preemption. These factors suggest that ignoring processor-cache affinity, which is typically the case in existing multiprocessor operating systems, can degrade performance. On the other hand, fixing tasks to run on specific processors may not be an appropriate alternative due to the potential load imbalance and the transitory nature of processor-cache affinity.**

We formulate queueing network models of different abstract scheduling policies, spanning the range from ignoring affinity to fixing tasks on processors. These models are solved via Mean Value Analysis, where possible, and by simulation otherwise. An analytic cache model is developed and used in these scheduling models to include the effects of an initial burst of cache misses experienced by tasks when they return to a processor for execution. A mean-value technique is also developed and used in the scheduling models to include the effects of increased bus traffic due to these bursts of cache misses. In comparing the different policies, our analysis shows that exploiting even the simplest forms of processor-cache affinity can potentially provide significant improvements over ignoring this affinity. On the other hand, scheduling decisions cannot be based solely on processor-cache affinity else other scheduling criteria, such as fairness, will be sacrificed. We show that only a small amount of affinity information needs to be maintained for each task, and demonstrate the importance of having a policy that adapts its behavior to changes in system load. Given the insights provided by our analysis, we develop a variety of practical scheduling policies ranging from simple management of local queues to augmenting a priority discipline with affinity information.

Index Terms—**Cache, performance analysis, queueing network models, scheduling, shared-memory multiprocessors.**

I. Introduction

In a shared-memory multiprocessor system, it may be more efficient to schedule a task on one processor than on another. Using such processor affinity information in shared-memory multiprocessor scheduling can improve performance, particularly if this information is inexpensive to obtain and exploit.

Manuscript received April 3, 1990; revised February 10, 1991.

M. S. Squillante is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. Part of this work was performed while he was a graduate student at the University of Washington, Seattle, WA.

E. D. Lazowska is with the Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.

IEEE Log Number 9204621.

The affinity of a specific task for a particular processor may arise from many sources. For example, the affinity may be based on how fast the task can run on a particular processor in an environment of heterogeneous processors. Scheduling problems of this type have received considerable attention in the literature; e.g., see [5], [9], [12]. These studies have produced exact and approximate algorithms to optimize a variety of performance metrics under a variety of conditions. They have also provided insight and understanding of basic principles that underlie the performance of scheduling policies in these environments.

Another form of processor affinity concerns the resources associated with the processors. For example, each processor may have a set of resources available to it and each task must execute on a processor that is associated with the set of resources it requires. In this environment, the goal of the scheduler is to optimize some performance metric while satisfying the constraints imposed by the system. This class of scheduling problems has also received considerable attention in the literature; e.g., see [5], [6], [28].

A third form of processor affinity is based on the contents of processor caches. Specifically, it may be more efficient in a shared-memory multiprocessor system to schedule a task on a particular processor than on any other if relevant data already resides in the processor's cache. This type of processor affinity, which is particularly interesting because it is time-dependent, has received far less attention. Moreover, most current multiprocessor operating systems have implemented schedulers that use simple priority schemes which completely ignore processor-cache affinity; examples include Mach [1], Digital Equipment Corporation's experimental Topaz system [21], and Sequent's DYNIX [22].

The typical behavior of tasks in computer systems suggests that ignoring processor-cache affinity may have significant performance implications. Tasks continuously alternate between executing at some processor and releasing this processor. The processor may be released to perform I/O or synchronization operations, in which cases the task is not eligible for scheduling until the operation is completed, or because of quantum expiration or preemption, in which cases the task is suspended to allow execution of another task. When the task returns for execution and is scheduled on a processor, it experiences an initial burst of cache misses. The duration of this burst depends, in part, upon the number of blocks belonging to the task that are already loaded in the processor's cache. Given current increases in cache sizes, a significant portion of a task's working set may reside in the cache of a

particular processor. With continuing increases in the relative cost of a cache miss [10], disregarding this cache reload time in scheduling decisions may cause significant increases in the execution times of individual tasks. Performance degradation in the system as a whole is also likely because of the increased bus traffic due to cache misses. This effect may be compounded by an increased number of cache invalidations due to the modification of a task's data still resident in another processor's cache.

One simple approach to alleviate this problem is to fix tasks to run on specific processors. This approach has serious limitations, however. The performance degradation due to the resulting load imbalance can be significant. Furthermore, the benefit of rescheduling a task on the processor where it most recently executed decreases with time, due to eventual cache-block replacements.

In this paper we analyze these issues of processor-cache affinity, with the objective of providing a better understanding of this form of processor affinity in shared-memory multiprocessor environments. We first demonstrate that cache reloading can have a significant effect on the time a task spends at a processor. We then formulate queueing network models of different abstract scheduling policies, which span the range from ignoring affinity to fixing tasks on processors. Mean Value Analysis [13], bounding techniques and discrete-event simulation are used to evaluate these models and compare the effectiveness of the various policies. An analytic cache model is developed and used in these scheduling models to include the effects of cache reloading. We also develop a simple mean-value technique to represent the effects of increased bus traffic due to cache reloading. Our analysis illustrates and quantifies the potentially considerable benefit of exploiting processor-cache affinity in shared-memory multiprocessor scheduling. Furthermore, our analysis illustrates the benefits and limitations of the different abstract scheduling policies considered, shows that only a small amount of affinity information needs to be maintained for each task, and demonstrates the importance of having a policy that adapts its behavior to changes in system load. Given the insights provided by our analysis, we develop a variety of practical scheduling policies ranging from simple management of local queues to augmenting a priority discipline with affinity information.

II. Problem Motivation

When a task is scheduled to run on a processor, it initially experiences a large number of cache misses as its working set is brought from main memory into the cache. Using the terminology of Thiebaut and Stone [24], we refer to the time delay of this initial burst of misses as the *cache-reload transient* and we call the group of cache blocks in active use by a task its *footprint* in the cache.

We have suggested that the cache-reload transient can be a significant factor, and that its significance will increase in the future. In fact, a key assumption of our analysis is that the time required to load a task footprint in the cache is not negligible. However, one might still believe that this is a secondary effect. If this is true, then the time a task spends at the processor is not significantly affected by the reload transient, and exploiting processor-cache affinity will have little or no performance benefit. Our objective in this section is simply to show that the cache reload time in existing systems can be significant. Although the relative cost of the reload transient depends upon many factors, including the task footprint size, the task computing demand per visit to the multiprocessor and the system architecture, we present the results of a few simple measurements to show that these reload penalties can be large.

All of the measurements presented in this section were obtained by experimentation on a Sequent Symmetry [16]. These experiments were written in the C++ programming language [20] and used the PRESTO runtime system [4], which was modified to store/restore the contents of floating-point registers during a context switch and to bind a DYNIX process to a processor. The application used in our experiments is composed of a set of tasks, each of which is implemented as a PRESTO thread. A task repeatedly references each block of its private "data" footprint (in addition to instruction references), performing some minor computations and releasing the processor. The measure of interest is the mean time to execute a task per visit to a processor.

Our first set of results considers the cache reloading time. In Fig. 1 we compare the mean execution time of a single task, as a function of the footprint size, for the cases of starting with a warm and cold cache. Both cases are run on a single processor, and the cache footprint is referenced via reads. In the warm cache case, the task returns immediately after releasing the processor, whereas the cache is flushed between task visits in the cold cache case. The curves plotted in Fig. 1 clearly show that the cache reloading time can be significant. In the worst case considered, cache reloading causes a 69% increase in the task execution time. Beyond this point, i.e., for larger footprint sizes, the significance of cache reloading begins to decrease somewhat as the task starts to replace portions of its own footprint.

The above measurements actually understate the effects of cache-reload transients because they do not include bus interference and write invalidations due to false data sharing. To consider these effects, we compare the mean execution time of multiple tasks running on a single processor with an equivalent workload on multiple processors. Our results are plotted in Fig. 2. In both cases, the cache footprint is referenced via writes. The tasks in the single processor case will execute with a warm cache (since the tasks always execute at the same processor, and the task footprints considered are small enough to allow them to essentially remain in the cache), whereas the tasks in the multiple processor case will typically run with a cold cache (since the tasks essentially do not execute at a processor where they recently ran) and will experience the effects of bus interference and write invalidations (since the footprint is referenced via writes). To eliminate any effects due to contention for the PRESTO scheduling queue, each task spins in a null loop for a random amount of time before releasing the processor. The measures plotted in Fig. 2 demonstrate even more forcefully that the cache reloading time can be significant.

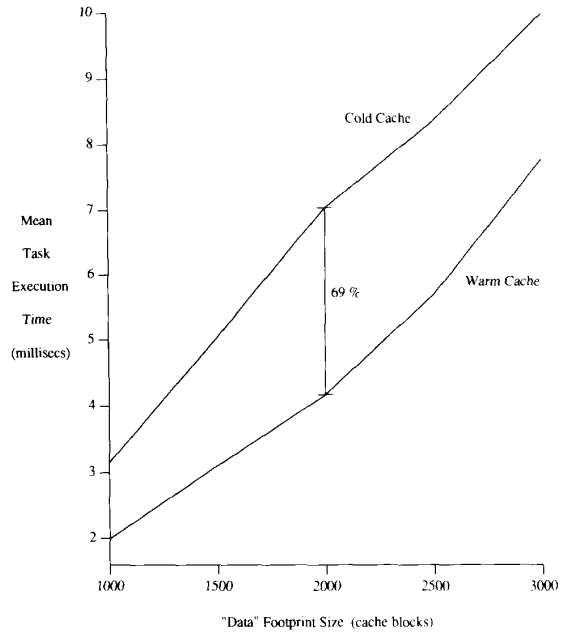


Fig. 1. Comparison of cache reloading times for warm and cold starts.

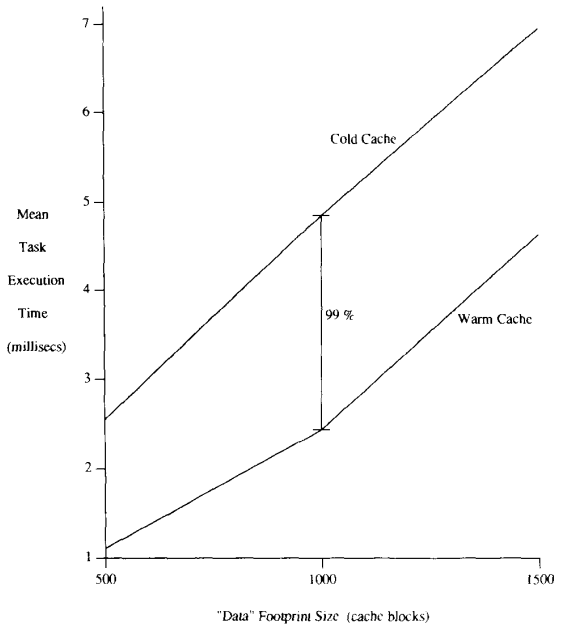


Fig. 2. Comparison of cache reloading times for warm and cold starts, including bus interference and write invalidations.

The measurement results presented in this section clearly show that the time required to load a task footprint in the cache can be a considerable factor, even in contemporary conservative multiprocessors such as the Sequent Symmetry. Furthermore, we expect this factor to become even more significant on newer architectures as cache sizes and the relative cost of a cache miss continue to rise.

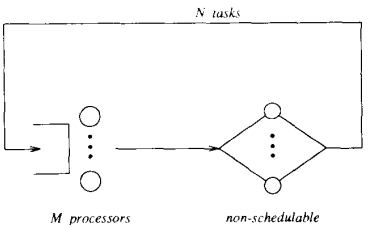


Fig. 3. System model.

III. The Models

We consider two formal models in our study of processor-cache affinity. The first is a queueing network model that represents the general multiprocessor system. The second is an analytic model of a processor's cache and is used within the system model to compute the expected initial burst of cache misses experienced by a task when it returns to a processor for execution.

A. The System Model

Our model of system behavior is that tasks alternate between executing at one of the processors and releasing this processor due to I/O, synchronization, quantum expiration, or preemption. Note that we are focusing on a micro level of scheduling, i.e., within a multiprogramming level. Such a system is modeled naturally by a closed queueing network [13].

Our system model, depicted in Fig. 3, consists of two service centers. The first is a *multiple server queueing center*, i.e., the center can service up to a fixed number of tasks simultaneously, and represents the multiprocessor composed of M identical processors. We refer to the queue of this center as the *ready queue* because it contains those tasks that are ready to execute but are waiting for an available processor. The second center is an *infinite capacity server*, i.e., there is no queueing because all tasks are serviced simultaneously, and represents the time spent by tasks while nonschedulable.

We consider a single customer class, i.e., all tasks are assumed to be identical. Let N denote the number of tasks in, or the *population* of, the model. The scheduling policy defines the manner in which tasks are removed from the ready queue; the policies we consider are defined in Section IV. The computing demands of tasks per visit to the multiprocessor are assumed to be independent and identically distributed as exponential random variables with mean D. Tasks become nonschedulable (due to I/O, synchronization, quantum expiration, or preemption) for random periods of time which are assumed to be independent and exponentially distributed with mean Z. Because the stationary distribution of a product form network is insensitive to service distributions at infinite capacity servers, we expect that the exponential assumption for nonschedulable periods is not critical to our results. By further assuming exponential computing demands, we are able to obtain analytically tractable results. Moreover, given the body of experience indicating that the accuracy of queueing network models is extremely robust with respect to violations

of separability assumptions (especially regarding throughput values) [13], we expect our results to be sufficiently accurate to be of general interest, particularly given the objectives of this paper.

Studies of cache miss behavior, such as [26] and [27], show that cache misses tend to occur in bursts which are situated between relatively long periods virtually free of misses. Note that this behavior is similar and related to phase transitions between working sets [7], and that it can be relatively easy to detect transitions between distinct footprints. For these reasons we assume that each task is operating within a particular footprint phase, i.e., phase transitions are not considered directly. We note, however, that tasks in the midst of a footprint transition phase (as well as new tasks and tasks whose footprints have been replaced) can be marked as such and used to reduce anticipated load balance problems in real systems.

B. The Cache Model

The cache-reload transient experienced by a task when it returns to a processor for execution depends upon the portion of its footprint already loaded in the processor's cache. With current increases in cache sizes, and since tasks may be nonschedulable for relatively short periods of time, this fraction could be quite large. However, as more tasks execute at the processor before the task returns, this fraction is decreased and therefore the reload transient experienced by the task increases. To illustrate and quantify this behavior, we developed a cache model that computes a task's expected *cache-reload miss ratio*, i.e., the portion of a task's footprint that must be reloaded when it returns to a processor for execution, as a function of the number of tasks that have executed on the processor since the task of interest last executed there. Our model is an extension of the cache model by Thiebaut and Stone [24].

The Thiebaut and Stone model is based on the notion of a task's footprint in the cache (the group of cache blocks in active use by the task). Assuming that each set is equally likely to be the destination of a block, and letting S denote the number of sets in the cache, the probability p that a single reference is to set j is equal to $1/S$. Further assuming that successive references to the cache are independent, the number of blocks assigned to a random set is binomially distributed. Therefore, if we let X be the random variable denoting the number of blocks in a set that are part of a task's footprint, the probability distribution of X is given by

$$\Pr[X = i] = \binom{F}{i} p^i (1-p)^{F-i} \tag{1}$$

for $0 \leq i \leq K-1$ and

$$\Pr[X = K] = \sum_{j=K}^{F} \binom{F}{j} p^j (1-p)^{F-j} \tag{2}$$

where K is the set associativity of the cache and F is the size of the task's footprint.

Consider the situation where the execution of a task A is followed by the execution of a task B, which is followed by the execution of task A. Let V be the random variable denoting the number of task A blocks, in a random set, that are replaced by the execution of task B. Assuming LRU replacement of cache blocks within a set, the probability distribution of V is given by

$$\begin{aligned}\Pr[V = i] &= \Pr[X_A = i]\Pr[X_B = K] \\ &\quad + \Pr[X_A = i+1]\Pr[X_B = K-1] \\ &\quad + \cdots + \Pr[X_A = K]\Pr[X_B = i] \\ &= \sum_{j=i}^{K} \Pr[X_A = j]\Pr[X_B = K+i-j], \end{aligned} \tag{3}$$

for $1 \leq i \leq K$, where X_A and X_B are random variables denoting the number of blocks in a set belonging to tasks A and B, respectively, and their distributions are as given above for X. The average number of reload misses that task A experiences when it regains control of the processor is therefore equal to

$$F - S(E[X_A] - E[V]) \tag{4}$$

where $E[X_A]$ denotes the expectation of X_A and $E[V]$ denotes the expectation of V. We note that a similar equation for this case was independently derived by Agarwal, Horowitz, and Hennessy [2].

Instead of focusing on the number of blocks replaced by a single task, we consider the number of blocks replaced by I tasks that execute during the absence of task A. We assume that these I tasks are unique and that they execute long enough to load their footprints in the cache. (Note that any violation of these assumptions will only decrease task A's cache-reload transient, thus increasing its affinity for the processor of interest.) Let X_i, $1 \leq i \leq I$, be the random variable denoting the number of blocks in a set belonging to task i, and define $W_I = X_1 + X_2 + \cdots + X_I$. Due to the independence of the X_i's, the probability distribution of W_I is given by the convolution of their distributions. Assuming that the X_i's are identically distributed and that this distribution is as given above, the distribution of W_I is more precisely given by the I-fold convolution of the distribution of X with itself.

We now define V_I to be the random variable denoting the number of task A blocks, in a random set, that are replaced by the execution of I intervening tasks. We assume that all tasks are identical, thus the sizes of their footprints are equal, and we use F to denote the size of a task's footprint in the cache. The probability distribution of V_I is then given by (3), where X_B is replaced by W_I, and the average number of reload misses that task A experiences when it regains control of the processor is given by (4), where V is replaced by V_I. The cache-reload miss ratio experienced by task A, as a function of the number of intervening tasks I, is given by

$$R(I) = \frac{F - S(E[X_A] - E[V_I])}{F}. \tag{5}$$

We refer to $R(I)$ as the cache-reload miss ratio function.

The solid curves in Fig. 4 illustrate the expected cache-reload miss ratio, as a function of the number of intervening tasks that are scheduled, for a 4-way set associative cache with $8K$ sets. Similarly, the dotted curves are for an equivalently

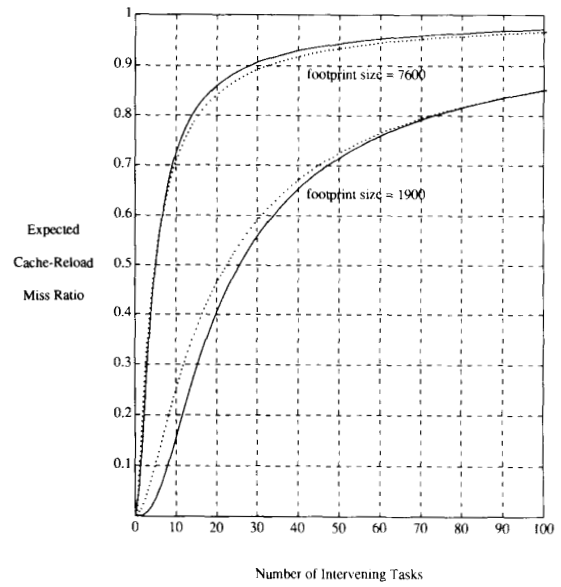


Fig. 4. Expected cache-reload miss ratio for a 4-way (2-way) set associative cache with 8K (16K) sets.

large cache with a set associativity of 2. A key observation is that the reload miss ratio increases rapidly with increasing numbers of intervening tasks. This suggests that there may be considerable benefit in exploiting processor-cache affinity in multiprocessor scheduling if the opportunity arises after only a relatively small number of intervening tasks have been scheduled on the processor, but that this benefit decreases rapidly.

Numerous validation studies by Thiebaut and Stone show excellent agreement between their cache model and observations based on program-address traces, with the only exceptions being instances where the model overestimated the cache-reload transient due to a greater clustering of references than assumed by the model [24], [19], [23]. These results together with our additional assumptions suggest that, with the exception of certain pathological cases, any inaccuracies in our cache model will tend to yield larger task reload transients than realized in practice. We therefore expect our model to be sufficiently accurate, noting that any bias in our miss ratio function is inclined to understate the benefit of exploiting processor-cache affinity information in scheduling decisions.

Given the expected cache-reload miss ratio function provided by our cache model, we can augment our system model to more accurately reflect the time a task spends at the multiprocessor. Assuming that the average time required to reload an entire task footprint in the cache is C, the mean time a task spends in service per visit to the multiprocessor is given by

$$D + CR(i) \tag{6}$$

where D is the task's inherent computing demand and $CR(i)$ is the cache-reload transient (i denotes the number of tasks that have executed on the processor since the task of interest last executed there). Maintaining consistency with the previous section, we assume these per-visit processing demands to be independent and identically distributed as exponential random variables.

C. Bus Interference

The above equation for the mean time spent at the multiprocessor by a task, during a single visit, ignores bus interference. In this section we describe a simple mean-value technique to include in our system model the effects of bus interference due to cache-reload transients. Similar approaches have been used elsewhere, e.g., [25].

Our method consists of inflating the cache-reload transient by the average bus utilization. We ignore bus traffic due to the execution of tasks, and only consider that due to the reloading of task footprints in system caches. We note that the former basically affects all of the scheduling policies in a similar manner, inflating the time a task spends at the multiprocessor by a certain amount. The additional cost of bus interference due to cache-reload transients, however, depends heavily upon the scheduling policy employed and is therefore of primary interest in our study. Thus, the mean time a task spends in service per visit to the multiprocessor is revised to be

$$D + \frac{CR(i)}{1 - U_{bus}} \tag{7}$$

where U_{bus} is the average bus utilization and the other variables are as defined previously.

If the processor never becomes idle, then the average time between bursts of bus activity due to the reloading of task footprints is $C\bar{R} + D$, where $\bar{R}$ is the mean cache-reload miss ratio. Hence, the mean total time between successive initiations of bus activity by a processor to reload task footprints is estimated as

$$\frac{C\bar{R} + D}{U_{cpu}}$$

where U_{cpu} is the mean utilization of a processor. Since the average duration of these bursts on the bus is $f_{ba}C\bar{R}$, where f_{ba} denotes the fraction of the reload transient that reflects bus activity, the portion of time each processor uses the bus is approximated by

$$\frac{f_{ba}C\bar{R}U_{cpu}}{C\bar{R} + D}. \tag{8}$$

Bus utilization is estimated as the product of the number of processors in the system and the fraction of time each processor uses the bus. Using (8) yields

$$U_{bus} = MIN\left(\frac{Mf_{ba}C\bar{R}U_{cpu}}{C\bar{R} + D}, 1\right). \tag{9}$$

Since the equation for U_{bus} is in terms of the unknown quantities $\bar{R}$ and U_{cpu}, the scheduling models are solved iteratively. We initially set U_{bus} to zero and solve the model using (7). The model solution yields values for $\bar{R}$ and U_{cpu}, which are used to obtain a new value for U_{bus} via (9). The model is solved again using this new value for U_{bus}. This iterative procedure continues until the difference between the

U_{bus} value of an iteration and that of the previous iteration is small, or until the bus becomes saturated.

The only remaining detail to discuss is choosing an appropriate value for f_{ba}. Actual values for f_{ba} depend upon the specific system being considered; of particular importance are the characteristics of the caches and main memory, the processing speed of the processors, and the bus protocol. However, our objective is to demonstrate, in a general fashion, the effects on system performance of increased bus traffic due to cache-reload transients. Moreover, the larger the value for f_{ba}, the more significant the degradation in performance due to reload transients. For these reasons, we do not want to overestimate the effects of increased bus activity; i.e., we would rather err on the side of underestimating this effect. We therefore use the conservative value of 0.1 for f_{ba}.

IV. The Scheduling Policies

In this section we describe the six scheduling policies that will be evaluated. As previously mentioned, these policies span the range from ignoring affinity to fixing tasks on processors. Each policy, with the exception of the first two policies to be described, updates the affinity information for a task after completion of its service demand at a processor. It should be noted that these are abstract policies used to identify fruitful approaches; thus, unless noted otherwise, each of the policies uses a single ready queue. Practical scheduling policies and implementation issues are discussed in Section VI.

A. First Come First Served (FCFS)

The First Come First Served ($FCFS$) scheduling policy completely ignores a task's affinity for a particular processor. When a processor becomes idle, $FCFS$ executes the task at the head of the ready queue, and when a task becomes schedulable $FCFS$ places it at the tail of this queue.

B. Fixed Processor (FP)

Under the Fixed Processor (FP) scheduling policy, each processor has a dedicated local queue and tasks are permanently assigned to these queues, i.e., the tasks are never migrated to other processors. Thus the N tasks are evenly divided among the M processors, and each processor works independently serving its N/M tasks in an $FCFS$ manner.

C. Last Processor (LP)

The Last Processor (LP) scheduling policy attempts to use the simplest form of "intelligent" task affinity, namely scheduling a task where it last executed. When a processor becomes idle, it searches the ready queue for the first *matching* task, i.e., one that last executed there. If a matching task is found, the processor runs it; otherwise, the processor executes the first task on the ready queue. When a task is placed on an empty ready queue and there are idle processors, the matching processor, if it is idle, is awakened and given the newly arrived task; otherwise, one of the idle processors, chosen arbitrarily, is given the task.

D. Minimum Intervening (MI)

The Minimum Intervening (MI) scheduling policy attempts to schedule the task with the greatest affinity for a processor whenever a scheduling decision has to be made. When a processor becomes idle, MI computes, for each task x on the ready queue, the number of tasks executed since x last executed there. It then runs the task with the minimum value. When a task arrives at the ready queue and idle processors exist, it is given to the processor for which the task has the greatest affinity.

E. Limited Minimum Intervening (LMI)

The Limited Minimum Intervening (LMI) scheduling policy schedules in a manner similar to MI, but limits the number of processors for which a task maintains affinity information. A task is *associated* with a processor if it maintains affinity data for that processor; let A denote the number of processor associations per task. When a processor becomes idle, LMI computes, for each associated task x on the ready queue, the number of tasks executed on that processor since x last executed there. If the queue contains associated tasks, LMI runs the task with the minimum value. Otherwise, it executes the first task on the ready queue and, upon completion of the service burst, replaces the task's worst processor association with an association to the processor on which the task just ran. When a task is placed on the ready queue and there are idle processors, the task is given to the associated processor with the greatest affinity for the task, if such a processor is idle; otherwise, the task is given to an arbitrary nonassociated processor in the manner described above.

F. Limited Minimum Intervening Routing (LMR)

The Limited Minimum Intervening Routing (LMR) policy is similar to LMI except that the decision of where a task will execute is made when the task becomes schedulable instead of when a processor becomes idle. When a task becomes schedulable, it is *matched* with the associated processor that currently has the minimum value for the following sum: the number of intervening tasks plus the number of tasks on the ready queue already matched with the processor. When a processor becomes idle, LMR runs the first matching task on the ready queue. If the queue does not contain a matching task, LMR executes the first task associated with the processor. If this also fails, LMR runs the first task on the ready queue and, upon completion of the service burst, replaces the task's worst processor association. This sequence of decisions is followed when a task is placed on an empty ready queue and there are idle processors.

V. Results

In this section we compare the performance of the scheduling policies described in Section IV, within the context of the models of Section III. The exact Mean Value Analysis (MVA) technique [13] in combination with our analytic cache model was used to solve the system model and obtain performance measures for the FP policy; the convolution algorithm [15]

was employed when numerical problems were encountered with MVA. Bounding techniques were used to obtain results under the $FCFS$ and LP policies, although discrete-event simulation in combination with our cache model was used when these bounds were not tight. Performance measures for the remaining policies were obtained via the latter approach. When simulation was used, values were obtained within 4% of the mean at 98% confidence levels.

The performance measures considered in this paper are system throughput ($\bar{X}$), mean task response time ($\bar{T}$), and the second moment of task response time ($\bar{T}^2$). In this section, $\bar{T}$ is plotted only when it illustrates information other than that shown by the $\bar{X}$ curves. Similarly, $\bar{T}^2$ is plotted only when it is qualitatively different, or provides additional insights, for the policies under consideration. We initially ignore the effects of bus interference due to cache-reload transients, i.e., (6) is used for the mean time a task spends in service per visit to the multiprocessor, thus understating the benefits of exploiting processor-cache affinity. Results including these effects, i.e., those obtained using (7), are presented in the final subsection. We do not address the effects of cache invalidations due to task migration, but note that these factors can only argue more favorably for exploiting affinity information in scheduling decisions.

Throughout this section the mean computing demand of tasks per visit to the multiprocessor is normalized to one unit (i.e., $D = 1$), the footprint size of each task is set to 1900 cache blocks, the remaining cache model parameters are as described in Section III-B, and the values considered for the mean time required to load task footprints in the cache range from 0 to D (i.e., $0 \leq C \leq 1$). The results presented in this section are for 32 processors (i.e., $M = 32$) and tasks that spend an average amount of time nonschedulable that is equivalent to their mean per-visit computing time (i.e., $Z = 1$). We present results here for various numbers of tasks (N), since this is a critical parameter that reflects system load.

In addition to the parameters for the results presented here, different parameter values were studied. The effects of these parameters can be briefly summarized as follows. When M or Z is increased, and all other parameters are fixed, system load is decreased and the results are similar to those presented for smaller populations. Similarly when M or Z is decreased, the load is increased and the results are similar to those presented for larger populations. In the same vein, when the cache (resp. footprint) size is increased (resp. decreased), and all other parameters are fixed, the cache-reload transient experienced by tasks decreases; the cache-reload transient increases when the direction of either of these parameters is reversed.

A. Comparison of Scheduling Policies

Our first set of results, shown in Fig. 5, compares using no affinity information (the $FCFS$ policy), using a very simple form of processor affinity (the LP policy), and fixing tasks to run on specific processors (the FP policy). These graphs clearly show the importance of including processor-cache affinity in scheduling decisions, particularly at higher system loads. Under light loads, and when C (the cache footprint reload time) is not large, $FCFS$ outperforms FP. This is due

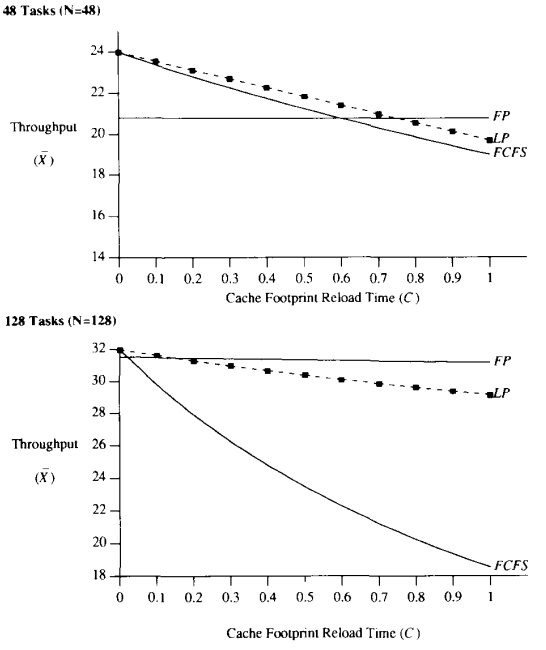


Fig. 5. Throughput comparison of First Come First Served (FCFS), Last Processor (LP), and Fixed Processor (FP) for a 32 processor system ($M = 32$).

to the fact that there is considerable load imbalance under the FP policy (i.e., tasks waiting at a busy processor while other processors are idle) and the cost of this imbalance outweighs the penalty for scheduling a task on a processor where its footprint is not fully loaded. However, as C increases, the penalty for scheduling a task on a processor where it has little or no affinity outweighs the cost of load imbalance, and FP outperforms $FCFS$. Under heavy loads, FP outperforms $FCFS$ at all but very small values of C. Since there are plenty of tasks assigned to each of the processors under high load, there is little or no load imbalance under the FP policy. (We note that although the FP curve appears to remain constant, it does decrease with increasing C.)

Fig. 5 also illustrates the benefits of exploiting simple processor affinity information in an intelligent but straightforward manner. Under light loads, LP performs essentially identically to $FCFS$, which dominates FP for small values of C (the cache footprint reload time) due to the load imbalance problems of the FP policy. Under heavy loads, LP performs somewhat similarly to FP, which dominates $FCFS$ for essentially all values of C due to the cache reload overhead of the $FCFS$ policy.

Our next set of results considers MI, a greedy algorithm that chooses the best task, based on the cache-reload transient penalty, whenever a scheduling decision is made. MI makes *locally optimal* decisions. In Fig. 6 we add performance measures for the MI policy to those for the $FCFS$, LP, and FP polices. These curves illustrate the benefits and limitations of using a greedy scheduling algorithm based solely on processor-cache affinity. Under light loads MI outperforms LP, and the difference between these curves widens with increasing

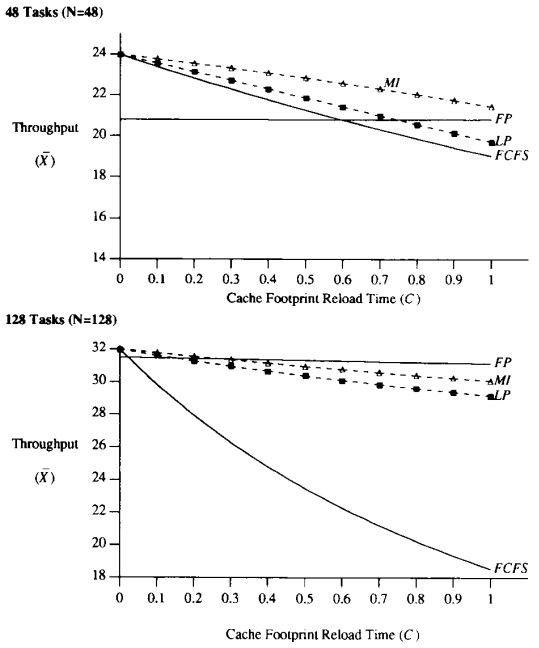


Fig. 6. Throughput comparison of Minimum Intervening (MI), First Come First Served (FCFS), Last Processor (LP), and Fixed Processor (FP) for a 32 processor system ($M = 32$).

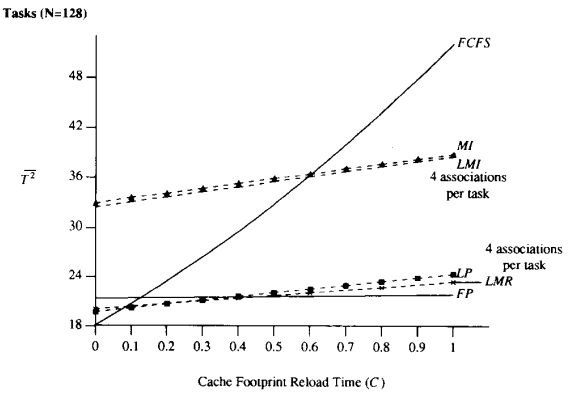


Fig. 7. $\bar{T}^2$ comparison of Minimum Intervening (MI), Limited Minimum Intervening (LMI), Limited Minimum Intervening Routing (LMR), First Come First Served (FCFS), Last Processor (LP), and Fixed Processor (FP), for a 32 Processor System ($M = 32$).

C. These performance differences are due to the use of more detailed affinity information under the MI policy. When load is heavy, MI offers a more modest advantage over LP.

While MI outperforms LP, a disadvantage of the MI policy is that it requires maintaining a large amount of affinity information for each task. We therefore consider the LMI scheduling policy, which attempts to approximate MI but limits the amount of affinity information maintained for each task. As expected, the performance of LMI approaches that of MI as the number of processor associations per task, A, approaches M. For this reason we do not plot LMI performance measures here, but instead summarize how rapidly LMI approaches MI with increasing A. Under light loads, LMI with $A = 8$ is essentially identical to MI, and the $A = 4$ case is not significantly different. The convergence is even more rapid under heavy loads, where LMI with $A \geq 2$ is essentially equivalent to MI. We also note that LMI with $A = 1$ exhibits first moment results that are essentially equivalent to those of LP. These results suggest that the system only needs to maintain affinity information for a small number of processors per task.

The MI and LMI policies both have the disadvantage that the variance in task response times is large relative to that of LP. This is illustrated in Fig. 7. A subset of the tasks are receiving excellent service while other tasks are receiving poor service. This is due to the fact that MI makes locally optimal scheduling decisions based solely on the cache-reload transient penalty, and completely ignores fairness. The subset of tasks receiving excellent service are assigned to processors shortly after becoming schedulable. The remaining tasks are executed only when none of the "favored" tasks are available. An unstable feedback causes some tasks to be indefinitely postponed at high loads.

To address this problem we consider the LMR policy by comparing its performance measures with those of LMI. In general, LMR provides a lower $\bar{T}^2$ (e.g., see Fig. 7) and similar first moment performance statistics, for the same values of A. Under light loads, the two policies provide essentially identical values for $\bar{X}$ and $\bar{T}$. This is due to the similar behavior of both policies when a processor, upon completion of a task, fails to find a task associated with it on the ready queue, and when a task arrives at an empty ready queue and the processor with the strongest affinity for the task is idle. Under heavy loads, LMR performs slightly worse than LMI and this difference in performance increases with C. By delaying scheduling decisions as long as possible, the LMI policy realizes a lower cache-reload transient penalty; i.e., the best scheduling decision made when a task becomes schedulable may no longer be the best decision when a processor needs to dispatch a task. These performance differences, however, are negligible.

The effects of increasing system load beyond what has been presented thus far was also considered. As the population of the model is increased, the effects presented above become magnified. However, as the system load continues to rise, the results illustrate significant performance degradation in all scheduling policies. When the load is very high, a large number of tasks are executed at each processor before a task returns to a processor, and therefore every task experiences a large reload transient penalty. The greedy policies (i.e., MI and LMI) do not exhibit as great a degradation in $\bar{X}$ and $\bar{T}$ as do the other policies, but there is a huge increase in $\bar{T}^2$ and a large number of tasks are *never* executed. These results emphasize the importance of limiting a system's multiprogramming level.

B. Inserting Idle Time

Each of the scheduling policies considered thus far, with the exception of FP, is work conserving [11], i.e., a processor

never remains idle when there are tasks waiting to be executed. In the results presented above, however, there are instances where it would be advantageous to have an idle processor *pause*, even though there are waiting tasks, until a task with reasonable affinity arrives; this is particularly the case when the reload transient penalty is large. As idle processors are inevitable in shared-memory multiprocessor environments due to factors such as the transient nature of system load and the variability of task service times, there is an important scheduling tradeoff between keeping the workload balanced and adhering to processor affinities.

An adaptive pause mechanism is needed that determines, as a function of system load and the cache-reload time, whether or not a processor should remain idle after completion of its last matching task on the ready queue. In this section we develop a very simple (abstract) version of such a pause mechanism, and present corresponding performance measures. Our objective here is solely to demonstrate the potential utility of an adaptive policy that determines the appropriate balance between the extremes of strictly sharing the workload among all processors and abiding by processor affinities blindly. We refer the interested reader to the study presented in [18] for a general investigation of this scheduling tradeoff.

A processor should not pause if the next task with reasonable affinity will arrive more than C units of time in the future, because in this case the processor could load the entire footprint of some task and start its execution before the arrival of a matching task (C is the cache footprint reload time). We can estimate the time for a task with affinity to arrive as follows. Assuming that processor affinities are fairly evenly divided and that a processor has finished servicing its last matching task on the ready queue, there are roughly N/M nonschedulable tasks with reasonable affinity for this processor. The aggregate arrival rate of these tasks to the ready queue is their departure rate from the delay center, and is equal to $(N/M)(1/Z)$. Thus the expected time for a task with affinity to arrive is $(M/N)Z$. This suggests the following *pause heuristic*: when a processor completes its last matching task on the ready queue, it pauses if $C \geq (M/N)Z$; otherwise, it runs the task at the head of the ready queue.

Various (steady-state) studies were performed to determine how long the pause period should be. Our results suggest that there is no benefit in terminating a pause period, within the context of our scheduling models, before a matching task arrives; i.e., if it is beneficial for a processor to pause then the processor should wait until a matching task arrives. For this reason, we consider pauses that are terminated only by the arrival of a matching task.

Our simple pause heuristic can be easily added to the LP, LMI, and LMR policies. In Fig. 8 we plot performance measures for the modified LP and LMI policies, along with those for the MI and FP policies. We note that our heuristic achieves the desired effect. In particular, LP is equivalent to FP once processors start to pause (compare Fig. 8 to Fig. 6). It is also worth noting the interesting interaction between LMI and the pause heuristic. Recall that the performance of LMI approaches that of MI as the number of processor associations per task, A, approaches M. The added pause

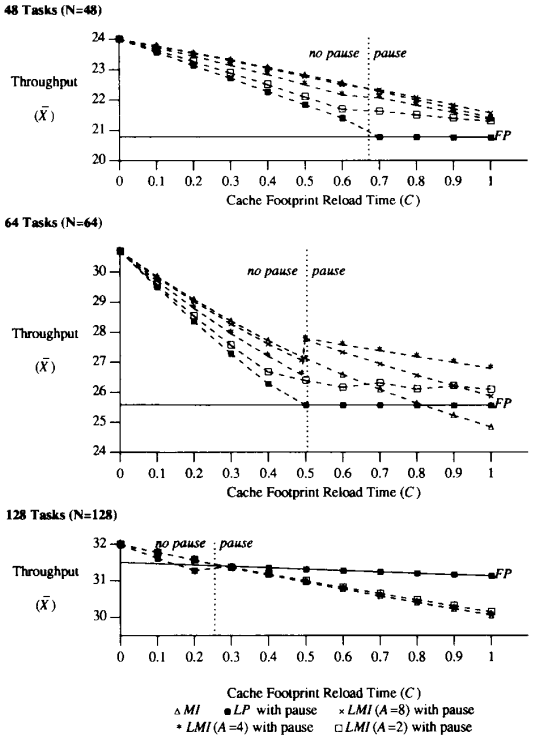


Fig. 8. Throughput comparison of Last Processor (LP) with pause, Limited Minimum Intervening (LMI) with pause, Minimum Intervening (MI), and Fixed Processor (FP) for a 32 processor system ($M = 32$).

causes this to occur more rapidly under light loads. As system load increases somewhat the pause heuristic actually causes LMI to outperform MI, as well as FP which always pauses. When $A = 2$ the curve essentially flattens and always remains above the FP curve; this is due to the fact that pauses are terminated more quickly with "good" matches. As A is increased to 4, the graph immediately jumps above the MI curve as soon as processors start to pause; by increasing A further, pauses are terminated even sooner, again with "good" matches. When A is raised to 8, LMI continues to jump above MI but degrades much more quickly with increasing C and actually drops below LMI with $A = 2$ at high C; this is due to the fact that pauses are being terminated too quickly with matches that are not as "good." Continuing to increase A causes LMI to perform more and more like MI. As load increases further the LMI results for different values of A are identical, all performing slightly better than FP when C is small and somewhat worse than FP otherwise.

We also compared the performance measures for the modified LMR policy with those of modified LMI. As previously described, the two policies provide essentially identical average performance before processors start to pause. When processors start to pause, though, LMR performs somewhat worse than LMI with identical values for A. This is due to the lazy decision process under the LMI policy, which causes it to realize a lower cache-reload transient penalty. We note, however, that these performance differences are

relatively minor. Since an implementation of the LMI policy would be more expensive than an implementation of the LMR policy, and since LMR does not have the task response time variance problems of MI and LMI (see Fig. 7), use of LMR as a practical policy may be worth consideration (see Section VI-A).

C. Bus Interference

The results presented thus far understate the benefits of exploiting processor-cache affinity in the sense that they ignore the degradation in system performance that occurs because of the increased bus traffic due to cache reloading. We now briefly consider this additional factor, noting that a system with a very high bandwidth bus (or interconnection network) will probably, at best, exhibit performance similar to that presented in Section V-A, and that a system with a (nearly) saturated bus can only benefit from using processor-cache affinity information in scheduling decisions. Our approach consists of inflating each task's reload transient by dividing it by one minus the average bus utilization, thus reflecting delays attributable to bus interference. We ignore bus traffic due to the execution of tasks, only considering that due to the reloading of task footprints in system caches. A more detailed description of our approach was given in Section III-C.

Our main goal here is to demonstrate the extent to which performance degradation is magnified by the effects of bus interference. In Fig. 9 we plot performance measures for the $FCFS$, LP without pause, and FP policies. (The dotted curves illustrate the same performance measures when bus interference is ignored.) As expected, the added effects of increased bus traffic due to cache reloading degrades system performance significantly. This is particularly the case with the $FCFS$ policy, which ignores processor-cache affinities. $FCFS$ continues to dominate FP under light loads, due to FP's load imbalance problems, but the value of C (the cache footprint reload time) below which this occurs is reduced considerably. System performance is affected more severely as C increases and the bus becomes saturated under light load when C is greater than or equal to some saturation point between 0.8 and 0.9. Under heavy loads, the degradation in performance is even more significant and the bus saturates when C is greater than or equal to a point between 0.6 and 0.7. These effects are due to the large cache-reload transient penalties, which are compounded by the resulting increase in bus traffic, realized under the $FCFS$ policy.

The differences between LP and $FCFS$ are due to the fact that the LP policy lessens the realized cache-reload miss ratio, and thus the effects of bus interference, and this effect increases with C. When load is light, these differences are not very significant because of the work conserving nature of LP. Thus, under light load, LP suffers from problems similar to those of $FCFS$. We note, however, that these problems with the LP policy can be reduced significantly with the addition of the pause heuristic discussed in the previous subsection.

The FP policy, as well as the LP policy under heavier system loads, is not affected significantly by the added effects of bus traffic. This is due to the small cache-reload overhead realized under the FP policy, and under the LP policy when load is heavy.

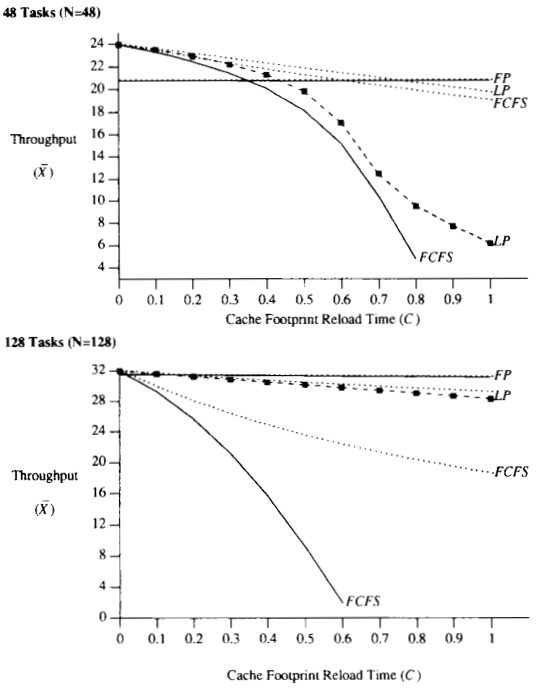


Fig. 9. Throughput comparison of First Come First Served (FCFS), Last Processor (LP), and Fixed Processor (FP), including bus interference.

The performance measures plotted in Fig. 9 demonstrate even more forcefully the importance of including processor-cache affinity in scheduling decisions. They also emphasize the point that exploiting even the simplest forms of processor affinity can provide significant improvements over ignoring affinity, particularly when load is heavy.

VI. Implementation Issues

The results described in the previous section provide a better understanding of processor-cache affinity in shared-memory multiprocessor systems. In this section, we use these insights to develop practical scheduling policies that take advantage of available affinity information.

We divide our discussion of these scheduling policies into two categories. The first class of policies, called *queue-based*, considers the organization and use of data structures to incorporate processor-cache affinity in scheduling decisions. The second class, called *priority-based*, considers augmenting the system's priority discipline with processor-cache affinity information; including this information in the priority calculation allows the scheduler to balance a task's affinity with other scheduling criteria. We note that combining these queue-based and priority-based techniques in the system scheduling policy could be advantageous in many shared-memory multiprocessor environments.

A. Queue-Based Policies

The simplest queue-based scheduling policy is an implementation of a variation of the LP policy defined in Section IV. Many existing shared-memory multiprocessor operating systems, such as Mach [1], make use of a global ready queue. This queue is used to store tasks without affinity for a particular processor, such as tasks that have not previously executed and tasks that have lost their affinity for a processor due to the replacement of their footprint or the transition to a different footprint. In addition to the global queue, a *local queue* is associated with each processor. When a task becomes schedulable, it is placed at the end of the queue associated with the processor where it last executed. After servicing a task, the processor runs the task at the head of its local queue. When the processor's queue is empty, it executes a task from the global ready queue. When this queue is also empty (or if its use is eliminated, e.g., see [17]), the processor can probe, in some fashion, the local queues of other processors and migrate a waiting task. If fairness is an issue, the policy could place tasks on the global queue after a number of returns to a local queue. When system load is heavy, the scheduling policy could wait for a task to arrive, i.e., pause, if its local queue is empty. This would cause the policy to function like the FP policy at heavy load, and by dynamically adjusting the scheduler's behavior with load the benefits of both LP and FP could be realized at relatively little runtime cost. A general study of such adaptive scheduling policies is presented in [18].

As demonstrated by our results, even this simple scheduling policy can potentially provide significant improvements over ignoring affinity. However, some performance may be gained, at the expense of some increases in complexity, by modifying the above policy to correspond to the LMR policy. Under this policy, a task is placed on one of A local queues after it becomes schedulable. If $A = 1$, the policy is identical to that described in the previous paragraph. Otherwise, the task is placed on the local queue of the associated processor for which it has the greatest affinity. Since our results suggest that A does not have to be very large, and because using too large a value for A would be expensive and could have adverse effects, values of A between two and four seem appropriate.

In a multiprocessor environment where processors are clustered, i.e., a single ready queue is shared among a small number of processors [14], an implementation of a variation of the LMI policy, where A is set to the number of processors in a cluster, may be appropriate. This policy functions in the same manner as the above LP policy, with the exception that tasks return to the queue associated with the cluster where it last executed. Each priority queue is *multi-linked*, i.e., its elements are linked together in multiple ways, based on the A different processor affinity orderings. When a processor finishes servicing a task, it runs a task from the head of its cluster's queue as defined by its affinity ordering.

B. Priority-Based Policies

There are two basic classes of priority scheduling policies: time-dependent and nontime-dependent. The latter is fairly static and is typically based on the type of task and its current state. The former provides additional degrees of freedom in the priority queueing discipline via a set of variable parameters, which are at the disposal of the designer to adjust relative task waiting times.

In each case, a simple and efficient mechanism is needed to estimate a task's cache-reload transient at a particular processor. One simple approach is based on our cache model. The system maintains the number of intervening tasks (or the number of intervening task *units*, where the number of units per task is some measure of the task's footprint size) between task executions by incrementing a per-processor counter whenever a context switch occurs at a processor and having each task record this value when it leaves the processor. The number of intervening tasks since a task executed at a particular processor is simply the difference between the processor's current execution count and the value recorded by the task when it last executed there. Given this value for a processor and task under consideration, the expected cache-reload transient experienced by the task when it returns to the processor is computed via our cache model (see Section III-B).

This approach may underestimate a task's affinity for a particular processor due to the implicit assumption that the intervening tasks, as computed above, are unique. Further problems may be caused by the assumption that these tasks execute long enough to load their entire footprint in the cache before relinquishing the processor. To eliminate these potential problems, the system could keep track of the number of cache-block replacements between task executions instead of maintaining the number of intervening tasks. This can be accomplished easily by incrementing a per-processor counter whenever a cache block is replaced at a processor, which can be done while the block is being fetched from main memory, and having each task record this value when it leaves the processor. Although this approach may be preferable to the one above, it will require a minor hardware modification in most systems. Given the support necessary to keep track of the number of cache-block replacements at a processor, the processor's activity during the absence of a task can be represented as the execution of a single program with footprint size equal to the difference between the processor's current replacement count and the value recorded by the task when it last executed there. Using this footprint parameter, the cache-reload transient experienced by the task when it returns to the processor is estimated via the original Thiebaut and Stone cache model (see Section III-B).

Having the scheduler compute a task's expected cache-reload transient at runtime in either of these ways would be too expensive in most environments. Moreover, accurately computing a task's footprint size is nontrivial. Therefore we pre-compute expected reload transients for a set of FT different footprint sizes and a set of different replacement counts or numbers of intervening tasks, depending upon which of the above methods is employed. These results are stored in a table, which we denote by TB_{cr}, and used by the system scheduler at runtime. Let N_{cr} denote the number of cache-block replacements, or intervening tasks, since a task executed at a particular processor, and categorize each task as having a footprint falling into one of the FT classes. To compute a

task's expected reload transient at a particular processor, the table is indexed by N_{cr} and the task's footprint class f. We now describe the use of TB_{cr} in practical priority disciplines.

One example of time-dependent priority scheduling, called *delay-cost* scheduling, is due to Franaszek and Nelson [8]. This policy has been implemented and used successfully on VM/HPO systems at the IBM T. J. Watson Research Center. Delay-cost scheduling is characterized by its use of a priority function that includes $\bar{x}_i$, the expected class i service time, and the time this class i task has spent in the system. Because the time spent at the processor by a task depends upon its cache-reload transient as well as its service demand, affinity information is easily incorporated into the delay-cost scheduling policy by replacing $\bar{x}_i$ with $\bar{x}_i + TB_{cr}(f, N_{cr})$ in the priority calculation.

A nontime-dependent priority function is one defined by some measurable task attribute(s) that does not include time as a parameter. Let $BP_i(x)$ denote the base priority assigned to a class i task in state x. We can easily augment such a nontime-dependent priority with a function of the number of cache-block replacements, or intervening tasks, at the processor of interest. Once again, we pre-compute the table TB_{cr} but instead of storing cache-reload transients, we store the corresponding reload miss ratios. We define, for each task class, the allowable range of increase in priority due to processor-cache affinity, and let RP_i denote this range for tasks of class i. The priority of a class i task is then computed as $BP_i(x) + (1 - TB_{cr}(f, N_{cr}))RP_i$. Actual values for the RP_i's will depend upon the intended system environment, and may require experimentation. We also note that the effect of affinity could be adjusted with system load.

Finally, we consider the combined use of time-dependent and nontime-dependent functions in priority calculations; such a priority discipline is used, for example, in UNIX [3]. Using the above notation, the priority of a task is computed via the formula $BP_i(x) + AGE_i(t)$, where $AGE_i(t)$ is an *aging* function used to gradually increase the priority of class i tasks waiting for service. To augment this discipline with processor-cache affinity, we define a function $AFFINITY_i()$ that maps the expected reload transient of a class i task to some priority factor in the same way that $AGE_i()$ maps the waiting time of a class i task to a priority factor. This affinity function is simply added to the existing priority formula, i.e., the priority of a class i task is computed as $BP_i(x) + AGE_i(t) + AFFINITY_i(TB_{cr}(f, N_{cr}))$.

The interaction between the aging and affinity functions is important. If the aging function grows too slowly, fairness is sacrificed. If the aging function grows too fast, task affinities will be effectively ignored. Similar arguments could be made for the affinity function. There clearly needs to be a nice balance between these two functions. The proper balance depends upon the intended system environment, and may require experimentation.

VII. Conclusions

In this paper we explored the importance of using processor-cache affinity information in shared-memory multiprocessor scheduling, with the goal of providing insight and understanding of basic principles that underlie this multiprocessor scheduling issue.

The results of several measurements on an existing system were first presented, demonstrating that cache reloading can have a significant effect on the time a task spends at a processor. We designed several abstract scheduling policies, which span the range from ignoring affinity to fixing tasks on processors, we formulated queueing network models of these policies, and we compared their effectiveness. A cache model was developed and used in these scheduling models to include the effects of cache reloading. A simple mean-value technique was developed to include the effects of increased bus traffic due to cache reloading.

Our analysis illustrates and quantifies the potentially considerable benefit of exploiting processor-cache affinity information in scheduling decisions. In particular, we show that under contemporary scheduling policies, which completely ignore processor-cache affinity, the cache-reload transient penalty may have a significant effect on both individual task performance and system performance, and this effect increases with system load although eventually degrades at very high loads. When the analysis includes increased bus traffic due to cache reloading, the bus can actually become saturated as the large reload transient penalties are compounded by bus interference. We also show that the other extreme of fixing tasks to run on specific processors suffers from load balance problems under light loads, as expected. These and our other results demonstrate the importance of having a policy that adapts its behavior to changes in system load.

Our analysis also illustrates the benefits and limitations of policies that lie between the extremes of ignoring affinity and abiding by it blindly. A simple work conserving policy that attempts to schedule tasks where they last executed can potentially provide considerable performance improvements over ignoring affinity. Increasing the amount of affinity maintained for each task may further improve performance, but the benefit gained decreases rapidly suggesting that only a small amount of affinity information per task is needed. We also showed that greedy algorithms provide high throughputs and low response times, but suffer from a high variance in task response times and even indefinite postponement under heavy loads.

The actual importance of using processor-cache affinity information in shared-memory multiprocessor scheduling depends upon many factors. Preliminary studies, both at the IBM T. J. Watson Research Center and elsewhere, show that there are system architectures and workloads for which exploiting processor-cache affinity provides significant improvements in performance, while there are others for which the use of this affinity will have little or no performance benefit. The key point is to identify the class of applications that can benefit from affinity scheduling, define the circumstances under which this benefit can be realized, and provide the scheduling policies with the ability to take advantage of this information when appropriate. Some of the important factors in making these decisions are the size of processor caches, the locality of the task memory references, the size of the task footprint in the cache (i.e., F), the ratio of the footprint loading time to the

computing time of the task per visit to the multiprocessor (i.e., C/D), the time spent nonschedulable by the task between processor visits (i.e., Z), and the system load.

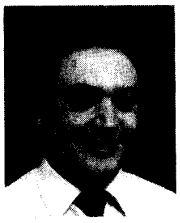
Given the insights provided by our analysis, we suggested several practical scheduling policies that take advantage of available affinity information, ranging from simple management of local queues to augmenting a priority discipline with affinity information. Planned future research includes further experimentation with various real workloads running under implementations of some of these policies.

ACKNOWLEDGMENT

We would like to thank H. Stone and D. Thiebaut for discussions about their cache model, our extension of this model, and the use of these models in multiprocessor scheduling. We would also like to thank P. Flemming, A. Greenberg, P. Heidelberger, S. Laha, R. Nelson, W.-H. Wang, and J. Zahorjan for discussions about the research described in this paper. D. Eager, H. Stone, J. Zahorjan and the anonymous referees provided helpful comments on an earlier version of this paper.

REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proc. USENIX Assoc. Summer Tech. Conf.*, June 1986, pp. 93–112.
[2] A. Agarwal, M. Horowitz, and J. Hennessy, "An analytic cache model," *ACM Trans. Comput. Syst.*, vol. 7, no. 2, pp. 184–215, May 1989.
[3] M. J. Bach, *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986.
[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "PRESTO: A system for object-oriented parallel programming," *Software: Practice and Experience*, vol. 18, no. 8, pp. 713–732, Aug. 1988.
[5] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling.* Reading, MA: Addison-Wesley, 1967.
[6] D. H. Craft, "Resource management in a decentralized system," in *Proc. Ninth ACM Symp. Oper. Syst. Principles,* Oct. 1983, pp. 11–19.
[7] P. J. Denning, "Working sets past and present," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 64–84, Jan. 1980.
[8] P. A. Franaszek and R. D. Nelson, "Delay cost scheduling for timesharing systems," IBM Res. Rep. RC 13777 (#61800), June 1988.
[9] E. Horowitz and S. Sahni, "Exact and approximate algorithms for scheduling nonidentical processors," *J. ACM* vol. 23, no. 2, pp. 317–327, Apr. 1976.
[10] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. Third Int. Conf. Architectural Support for Programming Languages and Oper. Syst.*, Apr. 1989, pp. 272–282.
[11] L. Kleinrock, *Queueing Systems, Vol. II: Computer Applications.* New York: Wiley, 1976.
[12] E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Inform. Processing Lett.*, vol. 12, no. 1, pp. 9–12, Feb. 1981.
[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
[14] L. M. Ni and C. E. Wu, "Design tradeoffs for process scheduling in shared memory multiprocessor systems," *IEEE Trans. Software Eng.*, vol. SE-15, no. 3, pp. 327–334, Mar. 1989.
[15] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[16] Sequent Computer Systems, Inc. *Symmetry Technical Summary*, 1988.
[17] M. S. Squillante, "Issues in shared-memory multiprocessor scheduling: A performance evaluation," Ph.D. dissertation, Dep. Comput. Sci. and Eng., University of Washington, Sept. 1990.
[18] M. S. Squillante and R. D. Nelson, "Analysis of task migration in shared-memory multiprocessor scheduling," in *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, May 1991, pp. 143–155.
[19] H. S. Stone, personal communication, Nov. 1990.
[20] B. Stroustrup, *The C++ Programming Language.* Reading, MA: Addison-Wesley, 1986.
[21] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr., "Firefly: A multiprocessor workstation," *IEEE Trans. Comput.*, vol. C-37, no. 8, pp. 909–920, Aug. 1988.
[22] S. S. Thakkar, P. R. Gifford, and G. F. Fieland, "Balance: A shared memory multiprocessor system," in *Proc. Second Int. Conf. Supercomput.*, May 1987, pp. 1–9.
[23] D. Thiebaut, personal communication, June–Sept. 1989.
[24] D. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Trans. Comput. Syst.*, vol. 5, no. 4, pp. 305–329, Nov. 1987.
[25] M. K. Vernon, E. D. Lazowska, and J. Zahorjan, "An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols," in *Proc. The Fifteenth Annu. Int. Symp. Comput. Architecture,* June 1988, pp. 308–315.
[26] J. Voldman and L. W. Hoevel, "The software-cache connection," *IBM J. Res. Develop.*, vol. 25, no. 6, pp. 877–893, Nov. 1981.
[27] J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. Rosenfeld, "Fractal nature of software-cache interaction," *IBM J. Res. Develop.*, vol. 27, no. 2, pp. 164–170, Mar. 1983.
[28] A. Weinrib and G. Gopal, "Decentralized resource allocation for distributed systems," in *Proc. INFOCOM*, 1987.



Mark S. Squillante (S'85–M'86) received the A.B. (First Hons.) degree in mathematics from Washington College, Chestertown, MD, in 1982, the M.S. degree in computer science from Columbia University, New York, NY, in 1985, and the Ph.D. degree in computer science from the University of Washington, Seattle, in 1990.

He has been a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1990 and an adjunct faculty member of the Department of Computer Science at Columbia University, New York, NY, since 1991. The summers of 1988 and 1989 were also spent at the Watson Research Center in the Systems Theory and Analysis group. From 1982 to 1985, he was a Member of the Technical Staff at Bell Telephone Laboratories, Murray Hill, NJ. His research interests concern the design, analysis and theory of computer systems, including performance modeling, scheduling algorithms, and distributed and parallel systems.

Dr. Squillante is a member of Tau Beta Pi, Omicron Delta Kappa, the Association for Computing Machinery, and the Operations Research Society of America.

Edward D. Lazowska (M'90–SM'91) received the A.B. degree from Brown University in 1972 and the Ph.D. degree from the University of Toronto in 1977.

He is a Professor of Computer Science at the University of Washington, Seattle. He has been at the University of Washington since 1977, with the exception of 9 months spent at the DEC Systems Research Center in 1984–1985. His research concerns the design and analysis of distributed and parallel computer systems.

Dr. Lazowska has served as Chair of the Graduate Record Examination Board Computer Science test committee, Chair of SIGMETRICS (the Association for Computing Machinery's special interest group concerned with computer system performance), and General Chair of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. He is Program Chair for the 13th ACM Symposium on Operating Systems Principles, an editor of IEEE TRANSACTIONS ON COMPUTERS, and a member of IFIP Working Group 7.3 on computer system performance. Twelve Ph.D. and nineteen M.S. students have completed their degrees under his supervision.