Performance Impact of Run Queue Organization and Synchronization on Large-Scale NUMA Multiprocessor Systems[†]

Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

S. P. Cheng[‡]
Bell-Northern Research
Ottawa, Ontario
Canada
pcheng@bnr.ca

ABSTRACT

The goal of this paper is to study the impact of run queue organization on the performance of synchronization methods in multiprocessor systems. Two run queue organizations are considered: distributed and hierarchical organizations. The performance impact of spinning and blocking synchronization methods on these two run queue organizations is studied. We use two canonical workload types that require task synchronization: lock accessing and barrier synchronization workloads. The results presented here show that, when fine grain synchronization is required, the distributed organization is better. However, for large granularity tasks, the performance of the distributed organization is unacceptable and the hierarchical organization should be used. Note that the distributed organization is embedded into the hierarchical organization. Thus, for coarse granularity parallel applications, the hierarchical organization with its load sharing feature can be used; for fine-granularity parallel applications, the hierarchy of queues can be circumvented and the round robin task assignment can be done on processor local queues as in the distributed organization. Therefore, the hierarchical organization is useful in general-purpose large-scale shared-memory multiprocessors.

Index Terms: Multiprocessor systems, Parallel systems, Performance evaluation, Processor scheduling, Run queue organizations.

1. INTRODUCTION

Shared-memory multiprocessors can be classified into either uniform memory access (UMA) or non-uniform memory access (NUMA) systems. UMA systems provide uniform memory access between any pair of processor and memory module. These systems typically use a system bus to interconnect processors and memory modules. Such UMA systems are limited to small number of processors due to bus bandwidth limitations. Examples of such systems include the Sequent Symmetry [18] and the DEC Firefly [28].

When building large-scale shared-memory systems, it is inevitable to move from UMA to NUMA architecture where memory access is not uniform. In NUMA systems, the global shared memory of the system is a collection of the processor's local memories. In such systems, accessing a processor's local memory is much faster than accessing a remote memory. The advantages of NUMA architecture have been recognized by several researchers [13,16,34]. Some example NUMA systems are the BBN Butterfly, Cedar [13], and Dash [16] systems. This paper is concerned with large-scale NUMA shared-memory systems.

There are two basic run queue organizations: *centralized* organization and *distributed* organization. In the centralized organization, there is a single global run queue that is accessible to all processors in the system. On the other hand, in the distributed organization, a local run queue is associated with each processor. In the absence of run queue access contention, centralized organization provides the best performance mainly because of its load sharing characteristic. Centralized organization is widely used in several small systems including the Dynix on the Sequent Symmetry [18] and Mach on the Encore Multimax [5]. However, due

[†]Appears in *Euromicro Journal of Systems Architecture*, 1996

[‡]Part of this work was done while the author was at the School of Computer Science, Carleton Univesity.

to mutually exclusive access requirement, the global run queue becomes a system bottleneck as the number of processors increases [2,23,24].

The distributed organization, on the other hand, avoids the problem of run queue access contention but introduces additional problems. The main problem with this organization is to find an appropriate run queue for the threads (i.e., the thread assignment problem). A simple random assignment strategy, in which threads are simply assigned to random run queues, causes load imbalance resulting in performance degradation. Using the round robin assignment strategy, in which threads are assigned to run queues in a cyclic fashion, improves the performance substantially. Further improvements in performance can be obtained by using the join the shortest queue strategy. Theoretically, this strategy is very attractive. But this strategy introduces system overhead that increases with the number of processors and thus causes implementation problems.

A hierarchical organization has been proposed to avoid the shortcomings associated with the previous proposals [10]. It has been shown that, when frequent synchronization is not required (i.e., for large granularity threads), hierarchical organization performs better than the centralized and distributed organizations while eliminating the run queue contention problem. Section 3 gives a brief overview of the performance advantages of the hierarchical organization.

While synchronization is not frequent in some scientific computations, there are several applications that do require frequent synchronization (i.e., fine-grain synchronization). Previous studies into the performance impact of synchronization have assumed a centralized task queue organization. This organization, however, cannot be used in large-scale NUMA machines. Therefore, in this paper, we use the hierarchical organization as the representative task queue organization for the centralized organization. Our goal, then, is to evaluate the impact of the run queue organization on the performance of large-scale NUMA shared-memory systems where threads of a parallel application require frequent synchronization. The relative performance of these two run queue organizations is studied here using two canonical workload types: *lock accessing* workload and *barrier synchronization* workload.

In a multiprogrammed shared-memory multiprocessor system, a thread of control may have to wait for some event before proceeding. Two potential situations that could lead to this waiting are: (1) a thread waiting to get mutually exclusive access to shared data protected by a lock; and (2) a thread waiting for the other threads to reach the barrier point (barrier synchronization). The first situation is due to competition among the threads and the latter situation arises due to cooperation among the threads [32]. We consider these two situations in this paper.

There are two common ways to handle a waiting thread. The first is *spinning*, in which the thread continues to run on its processor busy-waiting i.e., repeatedly checks to see if the waiting condition has been met. For example, if a thread is waiting to acquire a lock, in the spinning policy, the thread busy-waits until the lock is free. The second is *blocking*, in which the thread relinquishes the processor. In the above example, the thread could enqueue itself to be awakened when the lock becomes free.

There is a performance trade-off associated with spinning and blocking schemes. Spinning causes waste of system resources. These resources include the processor on which the thread is spinning, communication bandwidth, and creates contention for memory. The latter two disadvantages can be minimized by spinning on a local copy. Blocking, on the other hand, introduces context switch overhead that accounts for saving the thread's state information, enqueueing the blocked thread, and scheduling a ready thread on the processor.

In this paper, task and thread are used interchangeably. The rest of the paper is organized as follows. Section 2 discusses the related work while details on the hierarchical and distributed run queue organizations are provided in Section 3. Section 4 describes the workload and system models we have used in performing this study. Spinning and blocking policies are described in Section 5. Sections 6 and 7 discuss the performance of these policies under the lock accessing and barrier synchronization workloads, respectively. Section 8 discuses the cache effects and conclusions are given in Section 9.

2. RELATED WORK

Zahorjan et al. [32] studied the tradeoff between spinning and blocking. Similar to the two workload models used in this paper, they used the lock accessing and barrier synchronization models (somewhat different in structure than the ones used here). They also consider two sources of uncertainty: multiprogramming and data-dependent behaviour. Using analytical models they concluded that for the case of lock accessing model, neither source of uncertainty significantly increases the expected spin time. For the barrier synchronization workload, both types of uncertainty lead to sharply increased spin times. They also investigated the potential benefit of not unscheduling a task that has the lock. Our work differs in several aspects. First, we have implemented both spinning and blocking and use the job response time as the performance measure. This measure implicitly takes all overheads and synchronization time into account. In contrast, Zahorjan et al. [32] use the mean number of spinning processors and analyze only the spinning policy. Second, we compare the performance of the spinning and blocking policies in the hierarchical and distributed task queue organizations rather than in the centralized organization, which has been used in the previous studies. We also model explicitly task queue access contention and blocking of a task. This is important because, in the hierarchical organization, blocking involves variable context switch overhead that depends on the number of queues visited. Third, we do not use the preemptive round robin scheduling. Instead, a run-to-completion (with voluntary blocking, if blocking is used) scheduling policy is used. Since a running task is not preempted involuntarily we do not have worry about unscheduling a task that holds the lock. Similar to their model, we include the effects of multiprogramming as well as data-dependent behaviour (through service time variance).

Feitelson and Rudolf [12] consider the performance benefits of coscheduling (they call it gang scheduling) when threads require fine grain synchronization. They consider only the barrier synchronization workload model. Three methods of synchronization have been compared. In addition to the two methods considered in this paper, busy-waiting with gang scheduling has also been considered in their study. Their workload model is similar to the barrier synchronization workload used in here. It has been shown that gang scheduling is effective only when fine grain interactions are needed. While their study proves that gang scheduling is definitely useful when fine grain interactions are required, we do not model gang scheduling for the following reasons: (1) Implementing gang scheduling requires hardware support. In their implementation, inter-processor broadcast of interrupts was necessary. (2) Gang scheduling also causes processor fragmentation if the gang size does not fit the number of available processors. Feitelson and Rudolf have shown that, under reasonable conditions, processor fragmentation may lead to a loss of up to 25% of the processing power. For large-scale NUMA machines, this represents a significant loss. This could potentially increase the average job response time by increasing the waiting time to get processors. Their study assumed a perfectly balanced system (difficult to achieve on a large-scale system consisting of hundreds of processors) and did not include the effects of job waiting times and processor fragmentation. Clearly, as acknowledged by them, more research is needed to efficiently support gang scheduling on larger machines.

Karlin et al. [15] compare the performance of several variations of a hybrid policy (i.e., spin-before-blocking) for the lock accessing type of workload. They used lock waiting time distributions obtained from five parallel programs to compare the performance of seven strategies including the two strategies considered here (i.e., always-spinning, always-blocking, and five hybrid policies). Their workload consists of a set of programs that exhibits very short lock waiting times (typically requiring less than 32 spins) and another set of programs that exhibits very short and very long lock waiting times. They also conclude that always-blocking performs poorly. Since their workload consists of a mix of programs that exhibits very short and very long lock waiting times, they found variations of the hybrid policies provide better performance.

Zahorjan et al. [33] discuss how spin times are affected by scheduling policy. One of their results indicates that for the lock accessing workload, the scheduling policy used in this paper (they call it application-based blocking policy) provides the best performance.

Martkatos et al. [20] have investigated the effect of multiprogramming on barrier synchronization when the scheduling policy partitions the hardware and gives each application a set of dedicated processors. In contrast to the previous studies, they consider the distributed task queue organization. They compare barriers with spinning as well as blocking. But the task scheduling policy used is the preemptive round-robin policy. Three barrier types are considered: centralized barrier, tree barrier, and combination barriers. Centralized barriers use a global counter and each task that arrives at the barrier increments the counter. Centralized barrier is simple to implement but it is not scalable to large systems because it creates a "hot-spot". Tree barriers, assumed in this paper, use a combination tree and the barrier completion time is logarithmic [30]. Combination barriers incorporate both tree and centralized barriers. They conclude that using an appropriate blocking barriers gives good performance.

Anderson [1] considers the performance of algorithms for software spin-waiting. Anderson et al. [2] study the performance implications of several data structures and algorithm alternatives for thread management. They consider, among others, centralized and distributed task queue organizations. Their results show that the centralized organization becomes a performance bottleneck even for small number of processors in bus-based systems.

Thread scheduling can be done either by the operating system kernel (kernel threads), as in the Dynix [18] and Mach [5] operating systems, or by the application itself (user-level thread), as in the Presto system [4]. Operating system-based thread scheduling is oblivious of the thread "state" (e.g., spinning or doing useful work) whereas user-thread can take thread state into account in making scheduling decisions. Anderson et al. [3] show that performance of kernel threads is inherently worse than that of user-level threads and propose kernel support to effectively support user-level threads.

3. TASK QUEUE ORGANIZATIONS

In the hierarchical organization, a set of task queues is organized as a tree with all the processors attached to the bottom level of the tree (i.e., as leaf nodes). Figure 1 shows an example hierarchical organization for $N = 8$ processors with a branching factor $B = 2$. Each task queue can then be viewed as a task queue in the centralized organization serving only the tree nodes (these nodes could themselves be task queues or processors) directly below it. The centralized organization can be considered as a special case of this structure with just one level (just the root node) having a branching factor equal to the number of processors in the system.

In such a hierarchical organization, all incoming tasks are added to the root task queue. Note that the tasks in the root node can be scheduled on any processor in the system. Let L be the leaf node level (i.e., processor level) in the tree. When a processor is looking for work, it first checks its associated task queue at level $(L-1)$. If that queue is empty it checks the parent node of this node at level $(L-2)$ and the process is repeated up the tree until it finds a task to be scheduled. However, in order to reduce access contention at the higher levels, when a task queue is accessed, a set of tasks rather than just one, is moved one level down the tree. The size of the set decreases progressively as one goes down the tree (taking into account the availability of increasing number of task queues). At the bottom of the tree this set size is reduced to just one task (corresponding to scheduling that task on the associated processor). We use a parameter called the transfer factor Tr to indicate the number of tasks transferred from a parent queue to its child queue. The parameter Tr is defined as follows:

$$Tr = \frac{\text{number of tasks moved one level down the tree}}{\text{number of processors below the child task queue}}$$

This type of organization avoids task queue bottleneck because branching factor can be adjusted. However, removing a set of tasks rather than just one task does increase task queue access time but only marginally. Of course, reducing the branching factor increases the height of the tree introducing (in addition to the task queue imbalance associated with the distributed organization) a different type of overhead in that more task queues will have to accessed. The impact of these factors is discussed in [10].

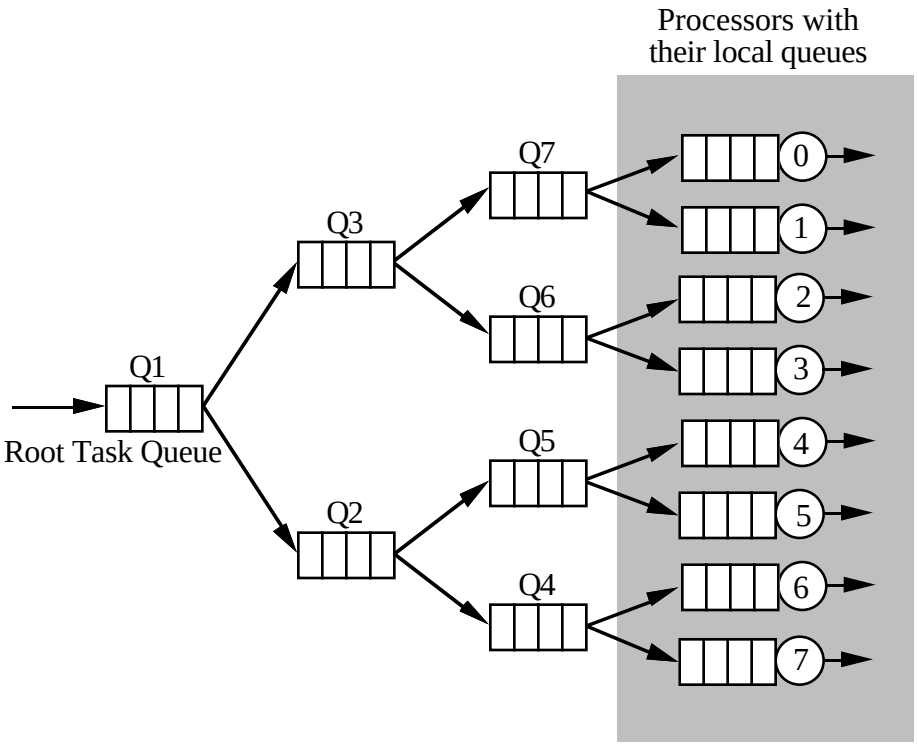


Figure 1 Hierarchical organization for $N = 8$ processors with a branching factor $B = 2$

The tree branching factor is another design issue in such an organization. We wish to keep the tree as "bushy" as possible (i.e., a tree with a large branching factor) because it allows better load sharing and results in better performance. Taken to the extreme this represents the centralized organization. However, the access contention for the task queue imposes a limit on the maximum branching factor of the tree. The impact of the branching factor on the performance of the hierarchical organization has been studied in [10].

In the distributed organization, task queues are distributed among the memory modules (see Figure 2). When a new task arrives into the system, the task will be placed in one of the task queues depending on the task assignment strategy used. Task assignment strategies can be broadly divided into two classes: *oblivious* strategies and *adaptive* strategies. Strategies belonging to the oblivious group make their assignment decisions independent of the system state; adaptive strategies use information about the current system state. Random task assignment, in which each arriving task is placed in a queue that is selected at random, and the round robin task assignment, in which each arriving task is assigned to a task queue in a round robin fashion, are examples of the oblivious strategies. Shortest queue assignment, which routes each arriving task to the queue that has the shortest queue length, is an example of the adaptive strategy. The adaptive strategies obviously improve system performance but there is an associated cost/overhead involved [8]. For instance, in the shortest queue strategy, current queue length information should be collected. For large systems with hundreds of processors this overhead may potentially offset the performance advantages obtained with these adaptive strategies. In this paper, we assume the round robin task queue assignment strategy. Our results show that, even with this simple task assignment strategy, the distributed organization performs as well as or better than the hierarchical organization for fine grain synchronization. Also note that the distributed organization is embedded into the hierarchical organization (compare Figures 1 and 2).

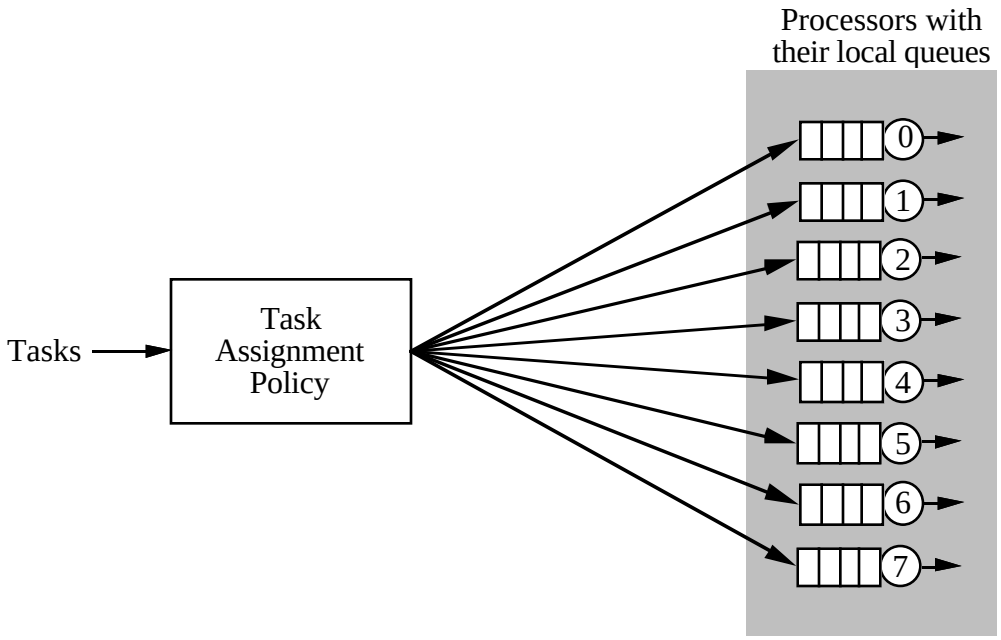


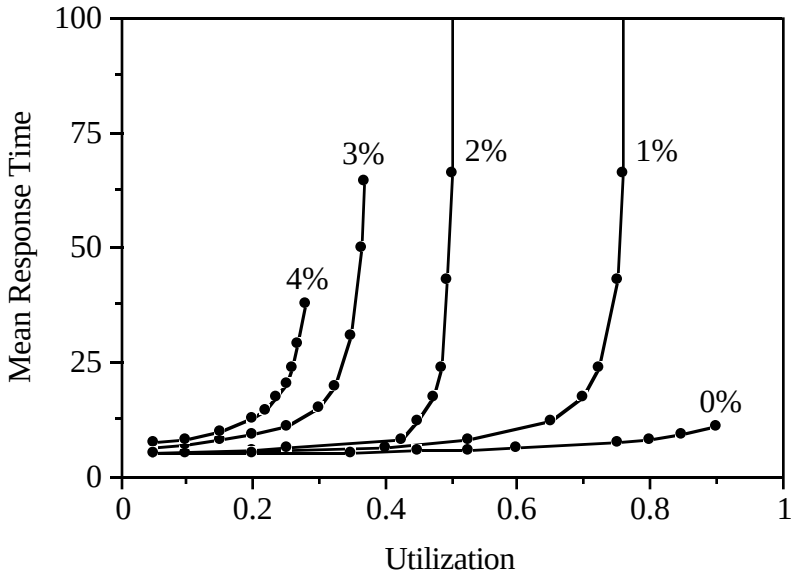
Figure 2 Distributed organization for $N = 8$ processors

Performance with Coarse Grain Synchronization

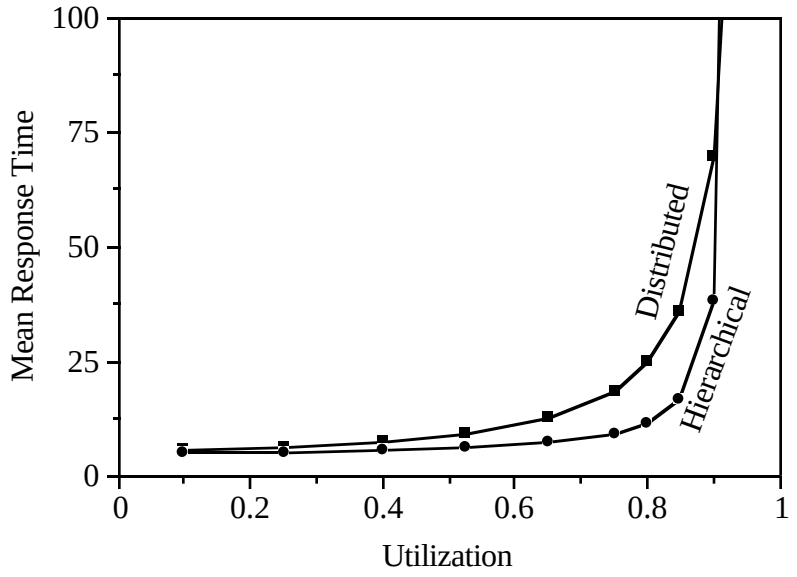
To incorporate coarse grain synchronization, a simple fork-and-join job model has been used. In this model, a fork-join job is assumed to be composed of a set of independent tasks that can be run on the system concurrently. The job completes when all of its component tasks are completed. Tasks within the job do not communicate with each other. Tasks participate in a single barrier synchronization at the end.

This job structure is reasonable for the class of problems with a solution structure that iterates through a communication phase and a computation phase. This class is exemplified by the n-body simulations of stellar or planetary movements, in which the movement of each body is governed by the gravitational forces produced by the system as a whole [14]. In this case, the model used here can be considered to represent one computation phase [11]. Similar job structure has been used in [7,8,9,10,17,19,34]. The job arrival process is assumed to be Poisson with parameter λ (i.e., job interarrival times are exponentially distributed). The number of tasks per job is exponentially distributed with a mean value of T. Tasks are characterized by processor service demand with mean $1/\mu$ and a coefficient of variation C_S. The amount of time needed to access (not including the waiting time to get exclusive access to the ready queue) the run queue is modelled as a fraction f of the average task execution time. In a lock-based implementation, this corresponds to the lock hold time. Performance of the centralized, distributed and hierarchical organizations has been reported in [10]. Here we will briefly discuss the performance advantages of the hierarchical organization under the fork-and-join workload.

Figure 3 presents the impact of run queue access time as a function of utilization for the three organizations. These are obtained for a system with $N = 64$ processors and the average number of tasks per job $T = 64$. The service time required by a task is assumed to be exponentially distributed with parameter $\mu = 1$. Since $\mu = 1$, f value can be interpreted as the ready queue access time. For example, $f = 1\%$ indicates that it takes 0.01 time units to remove/insert an entry from/into the ready queue. Note here the time unit corresponds to the mean task service time.



(a) Centralized organization (*f* is varied from 0% to 4%)



(b) Hierarchical and distributed organizations (*f* = 4%)

Figure 3 Performance as a function of utilization (N = 64, T = 64, μ = 1, B = 4, Tr = 1)

The results in Figure 3a show that the centralized organization is extremely sensitive to the access contention. For example, when the task queue access time is 2% of the average task execution time, the saturation utilization (and hence the throughput) falls to less than about 50%. In the centralized organization, since there is only one run queue global to all processors, it becomes a 'hot-spot' leading to a behaviour similar to that observed in interconnection networks [25,30].

Figure 3b presents the performance of the hierarchical and distributed organizations for f = 4%. In contrast to the sensitivity of the centralized

organization to f values, the distributed and hierarchical organizations exhibit a robust behaviour. The job response time for the distributed organization increases substantially as the system load increases. This is mainly due to the lack of load sharing in this organization. The hierarchical organization provides a substantially better performance than both the centralized organization and the distributed organization. This shows that the hierarchical organization achieves the load sharing of the centralized organization without creating task queue bottlenecks. More details can be found in [10].

Figure 4 shows the average response time versus service time coefficient of variation (CV) for the three organizations. The graphs are plotted for f = 0% and λ = 0.75. The hypothetical no contention case (f = 0%) is used basically to compare the performance of the hierarchical organization with that of the centralized organization in the absence of access contention. The results show that the hierarchical organization performs as well as the centralized organization. This shows that the hierarchical organization inherits the load sharing property of the centralized organization. The distributed organization is sensitive to the service time CV because of its lack of load sharing that is present implicitly in the centralized and hierarchical organizations.

Summary: In this section, we have presented a brief overview of the performance advantages of the hierarchical organization over the centralized and distributed organizations. For coarse grain synchronization, the hierarchical organization provides substantially better performance than both the centralized and distributed organizations. Therefore, the hierarchical organization is useful in large-scale parallel systems. The remainder of the paper addresses the performance issues when fine-grain synchronization is required by the tasks of an application program.

4. THE WORKLOAD AND SYSTEM MODELS

Simulation is used to study the performance of the hierarchical and distributed organizations. We consider a system with N identical processors. We model this system with an abstraction consisting of a set of queues and N servers. For the results reported here, we fix N at 64 because of the time needed to run the simulation experiments. We assume that these N processors are connected to N memory modules using a multistage interconnection network with an average communication delay of Cd_{in} which is expressed as a percentage of the average task execution or service time.

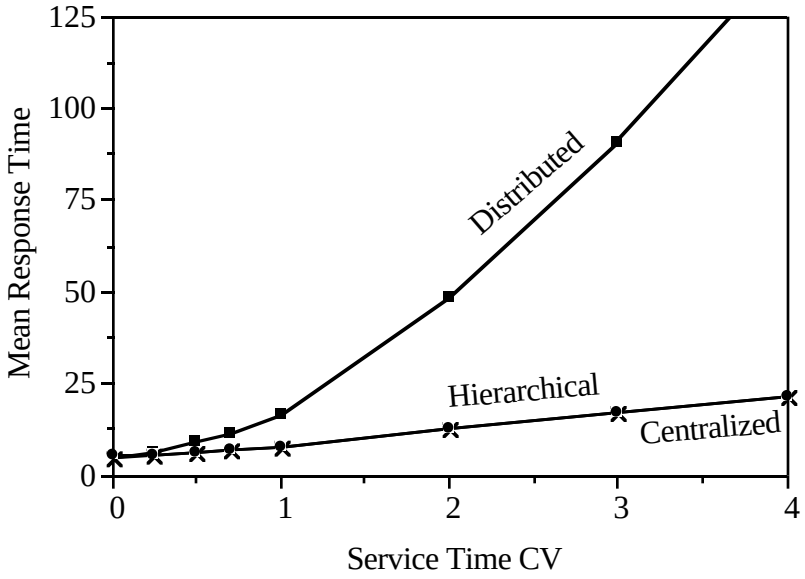


Figure 4 Performance sensitivity to task service time variance (N = 64, T = 64, μ = 1, B = 4, Tr = 1, λ = 0.75, f = 0% (hypothetical no contention case). Note that the lines for the hierarchical and centralized organizations are very close together.

We also assume that there is a front-end sub-system that is responsible for generating system load by creating jobs, splitting them into tasks, and inserting the tasks into an appropriate task queue. It is assumed that the front-end is connected through a communication network with an average communication delay Cd_{fe}. Since our interest is in the performance of the task queue organizations, we assume that the front-end is not a bottleneck in the system and that the front-end communication network is faster.

We consider two types of workload: *lock accessing* and *barrier synchronization*. In the *lock accessing* workload, a single lock is assumed to be shared by the tasks within a job [22]. In both workloads, job arrival process is assumed to be Poisson with parameter λ (i.e., job interarrival times are exponentially distributed). Figure 5 shows the basic structure of the generic program executed by each task in the lock accessing workload. In each iteration, tasks go through the following sequence: local computation, accessing the shared memory, requesting the lock, entering the critical section and releasing the lock. The first two steps are grouped as region 1 and the last three steps are grouped as region 2. Region 1 represents the independent computation phase and region 2 represents the critical section phase.

In the *barrier synchronization* workload, tasks iterate through a computation phase and a communication phase. In the computation phase, tasks are executed independently. Tasks wait for the other tasks to complete their computation phase and may communicate their results. This is represented by the communication phase. Figure 6 shows the basic structure of the program executed by each task in the barrier synchronization workload. In each iteration, tasks first perform some local computation and then communicate with others. This type of job structure is exemplified by the n-body simulations of stellar or planetary movements, in which the movement of each body is governed by the gravitational forces produced by the system as a whole [14]. The next movement of each body cannot start unless the movement of all the bodies in the current state have been completed.

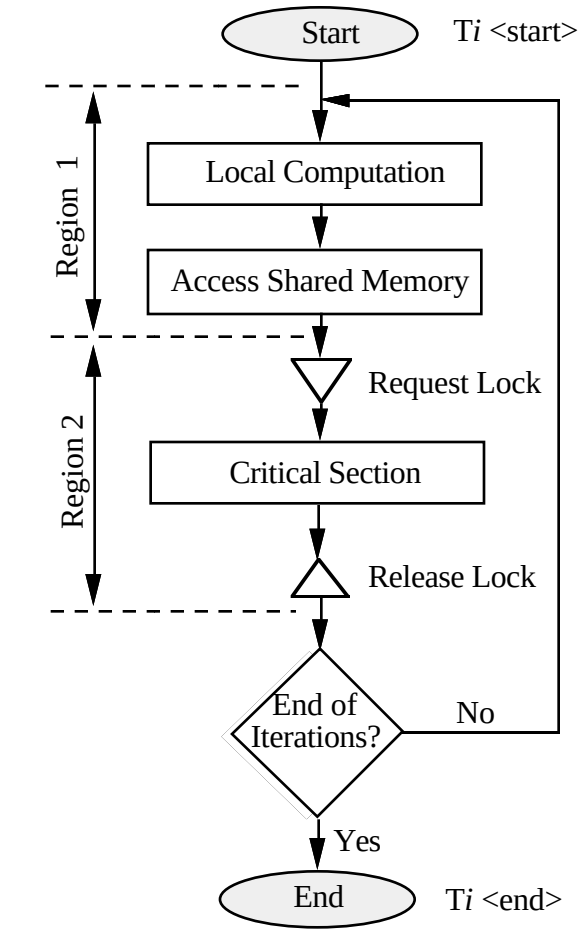


Figure 5 Generic lock accessing workload task structure for task T_i

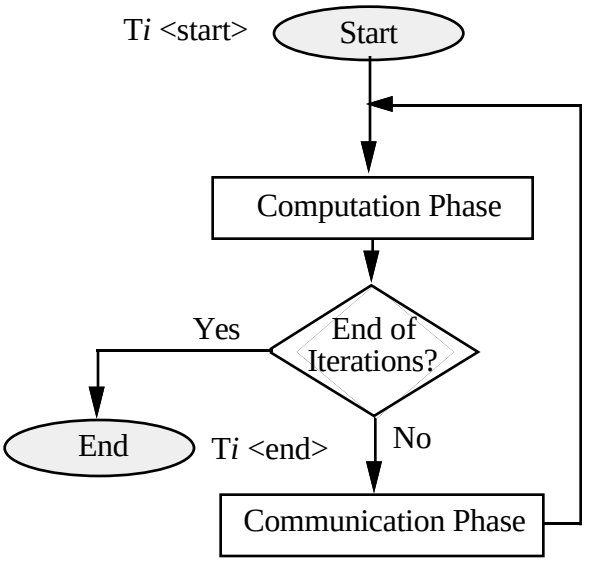


Figure 6 Generic barrier synchronization workload task structure for task T_i

The simulation model assumes that processors spin-wait on a local copy of the task queue lock (for example, using the MCS lock [21]). This mechanism successfully eliminates network contention due to spin-waiting [21]. (For details on spin lock alternatives, see [1].) A task queue access, therefore, involves testing the lock of the associated task queue (and copying it to its local memory in case the lock is not free) and accessing the task queue when the lock is free. Each of these accesses is assumed to require an average communication delay of Cd_{in}. In the case of the front-end processor accessing the root node, the corresponding delay is Cd_{fe}. Since the hierarchical and distributed organizations do not create hot-spots, sensitivity analysis of these parameters is not considered.

We model the amount of time needed to access (not including the waiting time to get exclusive access to the task queue) the task queue as a fraction f of the average task execution time. In a lock-based implementation, this corresponds to the lock hold time. We further assume that each time an idle processor accesses a task queue, it spends a constant amount of time independent of the actual number of entries in the queue i.e., task queue access time is deterministic. Deterministic access time has also been used in [17,24]. This is true for the scheduling algorithms considered in this paper.

The task scheduling policy used is a variation of the run-to-completion policy. When spinning strategy is used for synchronization, tasks relinquish processors when they are completed. Thus, in this case, the task scheduling policy is the non-preemptive run-to-completion policy. When blocking is employed, a task voluntarily relinquishes the processor. Such policy can be implemented by user-level threads (for example, see [4]). Thus, this scheduling policy avoids the undesirable situation of unscheduling a thread that holds the lock. A similar policy is shown to provide good performance [33].

5. SPINNING AND BLOCKING POLICIES

This section describes the spinning and blocking policies used in this study. The lock mechanism proposed in [21] is assumed; thus, *FCFS* order in the lock acquisition is considered.

(a) Spinning policy: When a task requires a lock or waits at a barrier, the processor executing the task sits in a tight-loop and waits. A deadlock may exist in the barrier synchronization workload if the number of tasks within a job inserted into a queue is greater than the number of processors linked to the queue (i.e., a task is placed in a queue and all the processor linked to the queue are spinning and waiting for the completion of the task). In order to prevent the deadlock, the number of tasks within a job moved down to a child queue is restricted to the number of processors linked to the queue. This restriction is a trade-off between deadlock prevention and the degree of load sharing.

(b) Blocking policies: In the hierarchical organization, blocking policies can be implemented in several ways. The simplest one treats the blocked tasks as new tasks when they are activated. The tasks are then inserted into the root queue. Similar to the global round robin policies, queue access contention for the root queue could cause performance problems. Furthermore, these tasks in the root queue will have to wait for the processors to complete all the tasks in the child queues. The blocking time (the time interval between when a task is blocked and when it is re-scheduled) will be considerably large. Preliminary results have indicated that this approach is not feasible for the hierarchical organization.

In the other approach, each node in the hierarchical structure maintains two task queues: a new ready task queue and a blocked task queue. All the new tasks transferred from the parent queue are inserted into the first queue. The latter queue contains all the tasks that have been suspended. When a processor accesses a node for a ready task, it places its suspended task (if there is one) into the blocked task queue of the node. When a suspended task is activated, the next available processor accessing the node scans through the blocked task queue first and then removes the task to process. Since the suspended tasks are inserted into the blocked queue during the acquisition of the ready task queue, a single lock can be shared by both task queues in the node. This approach is considered in this paper for the hierarchical organization.

In the distributed organization, the blocked tasks are inserted into the associated local blocked task queue. In the hierarchical organization, however, there

is a choice as to the selection of the task queue into which the suspended task is to be inserted. In the *static* blocking policy, all the blocked tasks are inserted back into the leaf queue node to which the processor is linked. In the *dynamic* blocking policy, blocked tasks are inserted into a task queue at a higher level from which the processor gets ready tasks. This policy utilizes the load sharing property of the hierarchical organization. Results reported in [6] show that the dynamic policy almost always provides a better performance than the static policy. Therefore, only the dynamic blocking policy results are presented in this paper for the hierarchical organization.

6. LOCK ACCESSING WORKLOAD RESULTS

6.1 The Workload Model

The generic structure of a task in the lock accessing model is shown in Figure 5. It is assumed that the job arrivals form a Poisson process with parameter λ (i.e., job interarrival times are exponentially distributed). The jobs are partitioned into a set of tasks *T*, which is uniformly distributed between 1 and the system size *N*. A *T* value of one implies that the user lock is not needed to execute the task. Note that in spinning with non-preemptive task scheduling policy, it makes no sense to have a number of tasks in a job that is greater than the number of processors. Tasks are characterized by the number of iterations *i*, which is uniformly distributed between 1 and Max_i, the region 1 service demand (S_C) and the lock holding time (S_l) in each iteration. Tasks within a job are assumed to have similar behaviour, i.e., identical number of iterations and the same mean service demand in each iteration. In the model, the number of iterations and the mean service demand (S_C) in each iteration are determined when a job is created. The service demand (S_C) of each iteration is generated by an exponential distribution. The region 1 mean service time in each iteration is uniformly distributed (among the tasks of a job) between $(1-V_{Sc})$ and $(1+V_{Sc})$ of S_C. This kind of variability in service demand has also been used in [31,33]. The variance represents the variation in accessing the shared memory (i.e., second step in Figure 5).

The lock holding time is assumed to be deterministic and is a fraction of the average task service demand in a single iteration (S_C+S_l). The deterministic lock holding time has also been used in [33,17]. The reader should bear in mind that there are two types of locks: one type of lock is associated with the task queues and is used to gain access to the task queues and the other type refers to the user lock accessed by the user program.

6.2 Simulation Results

Table 1 summarizes the default parameter values used in the lock accessing workload experiments. With these values, the actual system load or useful utilization (percentage of the time spent by the processor actually executing the tasks, excluding the queue accessing time and waiting time) is close to the arrival rate λ. The lock holding ratio is given by $\frac{S_l}{S_C+S_l}$. In the experiments, the values of S_C and S_l are set equal to 0.24 and 0.01, respectively. Thus, the locking holding ratio is 4%.

Table 1 Default parameter values used in the lock accessing workload experiments

Para-meter	Description	Default Value
λ	Arrival rate	0.7
Max_i	Maximum number of iterations	15
Cd*in*	Communication delay in IN	0.005
Cd*fe*	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
S_C	Mean service demand in region 1 per iteration	0.24
S_l	Lock holding time	0.01
V_{Sc}	Variation in mean service demand (uniformly distributed)	0.5

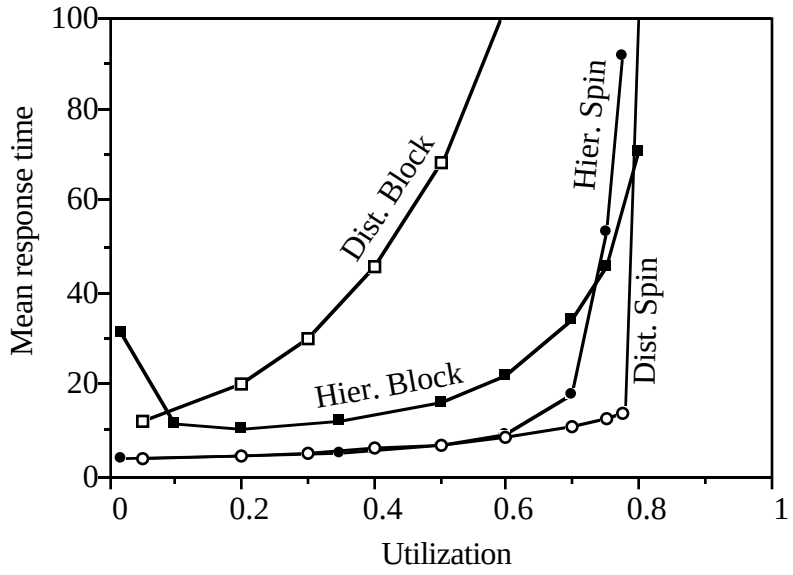


Figure 7 Performance of spinning and blocking policies as a function of useful utilization (ρ) - lock accessing workload

6.2.1 Principal Comparison

The performance of the spinning and blocking policies as a function of system utilization (ρ) is shown in Figure 7. Note that the utilization used here is the useful utilization in the sense that it does not include the contribution of the overhead activities such as accessing the task queues, spinning/blocking etc. In other words, utilization refers to processor busy time contributed by executing the tasks.

(a) *Spinning versus blocking:* The results show that, for both the hierarchical and distributed organizations, the spinning policy outperforms the blocking policy. This is an expected result based on the previous studies [32,15]. Briefly, this is due to the following phenomenon. In the blocking policy, rescheduling a suspended task will have two components of delay: first, the waiting time to acquire the lock, and second, waiting time to get a processor. Thus, in these policies, the lock holding time includes the second component of delay i.e., waiting time to get a processor. This, in turn, leads to substantial lock waiting times resulting in performance deterioration for all system loads except when the system is near saturation (see Figure 7). The spinning policy successfully avoids this problem because tasks are run to completion without preemption.

(b) *Performance of the blocking policy:* The blocking policy performs better in the hierarchical organization than in the distributed organization. This is mainly due to the load sharing property of the hierarchical organization. Note that, in the hierarchical organization, a blocked task is placed in a higher level queue which can be scheduled, when ready, on any of the processors below this queue node of the task queue tree. The results imply that the suspended tasks should be inserted into the queue at the highest level. Unfortunately, the queue access contention may occur at the root queue if all the suspended tasks are to be inserted back in the root queue. This demonstrates the trade-off between load-sharing and queue access contention. Also note that attempting to increase the scope of load sharing destroys cache affinity in that a suspended task may not be rescheduled on the same processor. The distributed organization maintains cache affinity by rescheduling a task on the processor it ran before suspension. The cache effects are discussed in Section 8.

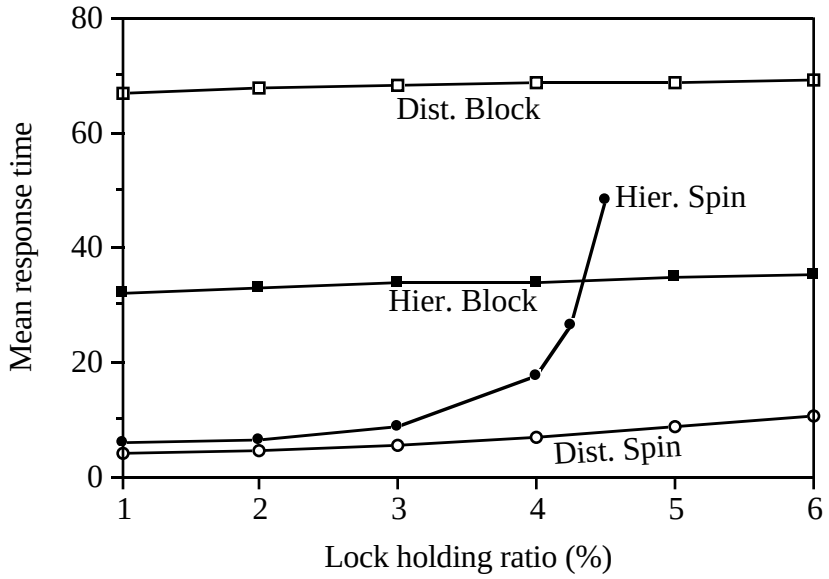


Figure 8 Impact of the lock holding ratio

Interestingly, the job response time of the blocking policy in the hierarchical organization increases dramatically at very low utilizations (<20%). At such low system utilizations, the processors frequently wait for a ready task at the higher levels of the task queue tree and these processors will have to be notified to resume a suspended task that is at a lower level of the tree. This overhead can be large (at least one communication delay is induced). Thus, the performance of the blocking policies deteriorates as the system load reduces close to zero. This problem can be remedied but such low utilizations are of no interest and hence are not considered any further.

(c) Performance of the spinning policy: The performance of the spinning policy is similar in both the task queue organizations from low to moderate system load (up to about 60% useful utilization in Figure 7). The reason for this is that, at these system loads, the load sharing property of the hierarchical organization does not really affect the performance due to the following: (i) jobs are restricted to have a maximum of N tasks per job, where N is the number of processors in the system, and (ii) the round robin task assignment policy used in the distributed organization tends to load balance the tasks at these low moderate system loads like in the hierarchical organization.

At higher system loads, the hierarchical organization tends to load balance the task execution in which all tasks of a job tend to be scheduled concurrently (similar to coscheduling). This leads to increased contention for the user lock; this in turn results in increased spin time wasting CPU cycles. For example, consider a job with 32 tasks (which is the average value used in the simulation experiments). Note that each lock access takes S_l time and therefore keeps the user lock busy for $32*S_l$ time per iteration. For the parameter values used here, this is equal to 0.32 time units. Thus, assuming that these 32 tasks are simultaneously scheduled, a task would have to wait at least $31*S_l = 0.31$ time units before accessing the critical section during the next iteration. This is because the other 31 tasks would have to access the critical section due to the FCFS policy used in granting requests for the lock on the critical section. Since each task spends, on the average, 0.24 time units in performing the local computation (region 1), the remaining time would have to be spent spinning, which is substantial. The hierarchical

organization increases the probability of concurrently executing all tasks belonging to a job because of the load sharing provided by this task queue organization. Thus the hierarchical organization performs poorly at high system loads. The disadvantage of the distributed organization in not providing good load sharing turns out to be an advantage when scheduling tasks that access a shared critical section by reducing contention for the critical section lock.

This is further demonstrated in Figure 8, which shows the performance of the spinning and blocking policies as a function of the lock holding ratio (for the useful utilization of ρ = 70% and $S_C + S_l = 0.25$). As discussed in Section 6.2, the lock holding ratio is defined as $\frac{S_l}{S_C+S_l}$. The spinning policy in the hierarchical organization is sensitive for the lock holding ratios greater than about 3%. The reason for this is that for values less than or equal to 3%, the lock busy time per iteration is 0.24 time units, which is approximately equal to the average region 1 execution time. Note that, as the lock holding ratio increases, the corresponding region 1 execution time decreases because $S_C + S_l$ is maintained constant at 0.25 per iteration. For higher lock holding ratios, spin time increases as explained in the last paragraph. Since processors can execute other ready tasks, the blocking policies are less sensitive to the lock holding time.

(d) *Hierarchical organization versus distributed organization:* The results presented in Figure 7 show that the distributed organization performs as well as or better than the hierarchical organization when the spinning policy is used. For small lock holding times, both organizations provide similar performance when the spinning policy is used. When the blocking policy is used, the hierarchical organization performs better. However, for the lock accessing workload, using the blocking policy results in substantial performance deterioration. Thus the distributed organization with the spinning policy is the preferred choice for the lock accessing type of workload.

6.2.2 Performance Sensitivity to Service Time Variance

As defined before, the variability of the region 1 task service time (V_{Sc}) represents the variation in accessing the shared memory. Note that the region 1 service demand during the same iteration is uniformly distributed between $S_C*(1-V_{Sc})$ and $S_C*(1+V_{Sc})$ among the tasks of a job. Figure 9 shows the perform-

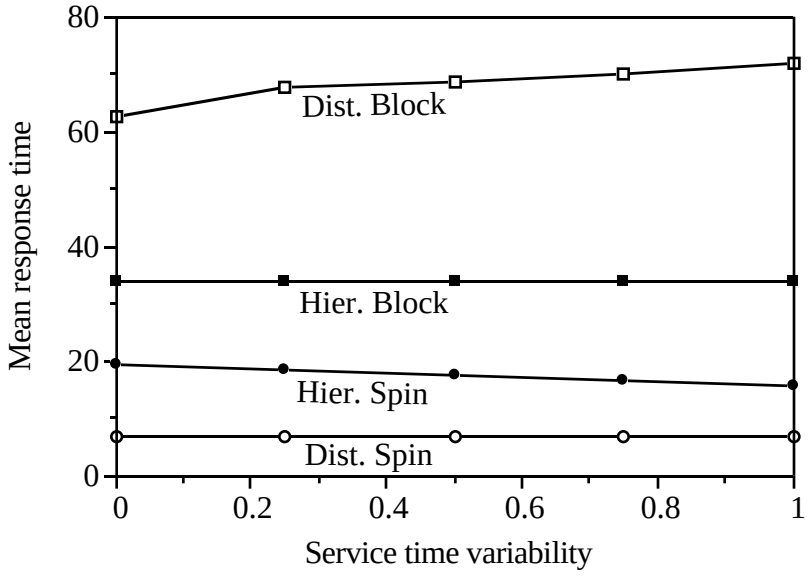


Figure 9 Performance sensitivity to the service time variability V_{Sc} - lock accessing workload

ance sensitivity of the spinning and blocking policies to the service time variability (V_{SC}). When the blocking policy is used, the performance of the hierarchical organization is insensitive to the service time variance while that of the distributed organization deteriorates. This is because of the lack of load sharing in the distributed organization that is implicitly present in the hierarchical organization.

When the spinning policy is used, the job response time of the hierarchical organization decreases marginally with increasing variability. This is due to the fact that the contention for the user lock reduces with an increase in service time variability. For the distributed organization, the performance of the spinning policy is insensitive to the variability in service time.

6.2.3 Impact of Granularity

The previous results in Section 6.2 assumed that the maximum number of iterations is 15. This section presents the impact of the maximum number of iterations per task (the mean value is $\frac{1+ \text{Max}_i}{2}$) on the performance of the spinning and blocking policies for ρ = 70%. The task service demand is fixed at 2 time units. The larger the value of the maximum number of iterations is, the smaller is the service demand per iteration (i.e., the system moves toward finer granularity).

Figure 10 demonstrates that the maximum number of iterations has a negative impact on the performance of the blocking policies. The spinning policies exhibit relatively marginal sensitivity to the number of iterations. This is natural to expect because, with increasing Max_i, tasks access lock more frequently. The same conclusion was arrived at in [26] in the context of scheduling parallel loops. In blocking policies, the suspended task acquires a lock and is then re-scheduled. There exists a time delay between when a task acquires a lock and when it is re-scheduled that is implicitly included in the lock holding time. Therefore, blocking policies are more sensitive to this parameter than spinning policies.

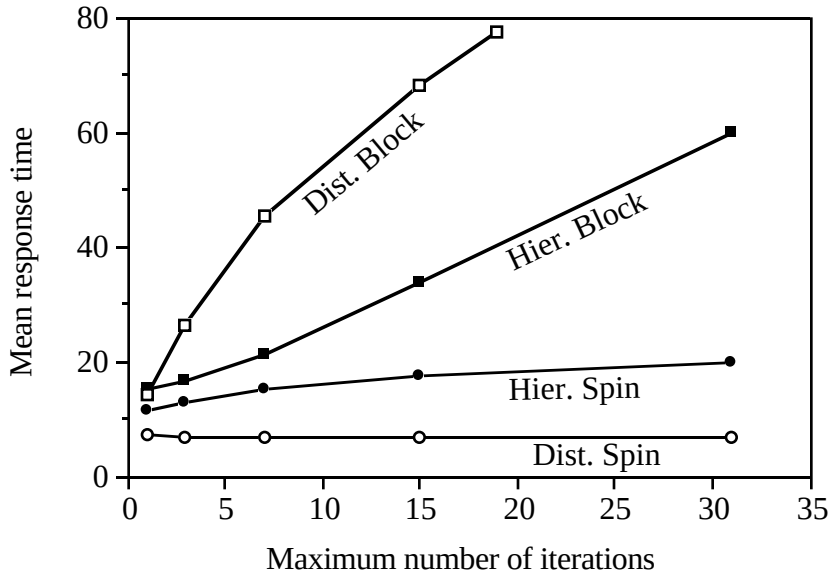


Figure 10 Performance as a function of the maximum number of iterations (Max_i)

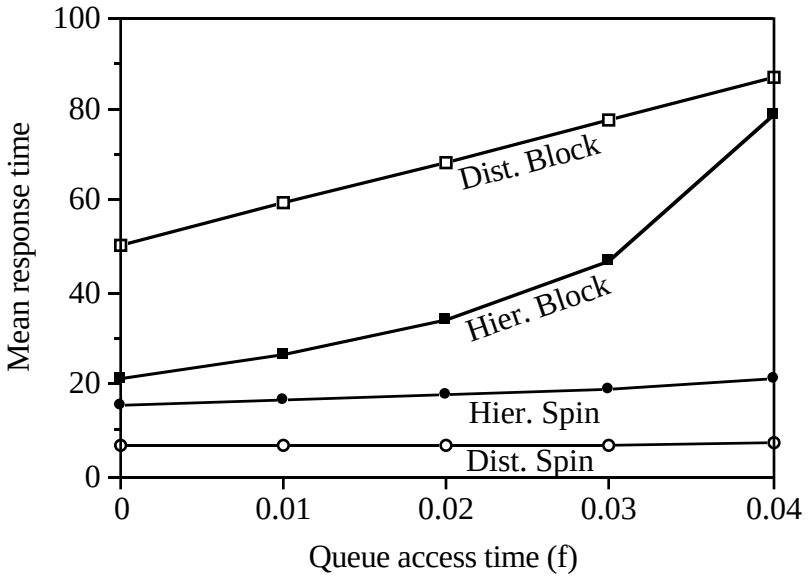


Figure 11 Performance sensitivity to the queue access time (*f*)

6.2.4 Impact of the Queue Access Time

This section investigates the impact of the queue access time (*f*) on the performance of the spinning and blocking policies. Figure 11 presents the performance sensitivity to the queue access time at ρ=70%. As discussed in Section 6.2.3, there exists a time delay that is implicitly included in the lock holding time. This time delay increases with increasing queue access time. Therefore, the blocking policies exhibit increased sensitivity to the queue access time.

In the hierarchical organization, a suspended task may be inserted into a queue at a higher level requiring additional queue accesses. Thus, the blocking policy in the hierarchical organization is more sensitive (than that in the distributed organization) to the queue access time. Because of this, increasing the queue access time deteriorates the performance of the blocking policy in the hierarchical organization.

Note that tasks in the system with the spinning policy only require a single queue access; therefore, the queue access time has only a marginal effect on the performance of this policy.

7. BARRIER SYNCHRONIZATION WORKLOAD RESULTS

7.1 Workload Model

The generic structure of tasks in the barrier synchronization job model is shown in Figure 6. As in the lock accessing job model, it is assumed that the job arrivals form a Poisson process with parameter λ. The jobs are partitioned into a set of tasks *T*, which is uniformly distributed between 1 and the system size *N*. Tasks are characterized by the number of iterations *i*, which is also uniformly distributed between 1 and Max_i, the computation phase service demand in each iteration S_C. Again, tasks within a job are assumed to have identical behaviour, i.e., identical number of iterations and computation phase service demand in the same iteration. In the model, the number of iterations and the mean computation phase service demand (S_C) in each iteration are determined at the time of job creation. The computation phase service demand of tasks in i^{th} iteration is uniformly distributed (among the tasks of a job) between ($1-V_{SC}$) and ($1+V_{SC}$) of S_{Ci} which is generated by an exponential distribution. This represents the variation in accessing the shared memory during the local computation phase.

The system is assumed to use the combining-tree barrier mechanism discussed in [21] to implement the barrier. Thus, the barrier synchronization time (S_b) is modelled as $Cd_{in} * \lceil \log T \rceil$ where T is the number of tasks in the job. In the spinning policy, the next computation phase of the application starts at S_b time units after the completion of the last task in the current computation phase. Similarly, in blocking policies, the suspended tasks are activated after a delay of S_b time units after the completion of the last task in the previous computation phase.

7.2 Simulation Results

Table 2 summarizes the default parameter values used in this set of experiments. The task service demand is fixed at 2 time units (excluding the time for executing the barrier mechanism). Section 7.2.1 presents impact of the system load. Section 7.2.2 shows the performance sensitivity to the variability in service time in the computation phase. Section 7.2.3 investigates the effect of the task granularity. Section 7.2.4 discusses the impact of the queue access time (*f*).

7.2.1 Impact of System Load

Figure 12 presents the performance of the spinning and blocking policies as a function of system utilization (ρ). Note that the utilization used in this paper is the useful utilization (see Section 6.2.1).

Table 2 Default parameter values used in the barrier synchronization workload experiments

Parameter	Description	Default Value
λ	Arrival rate	0.5
Max_i	Maximum number of iterations	15
Cd_{in}	Communication delay in IN	0.005
Cd_{fe}	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
S_C	Mean service demand in computation phase	0.25
V_{Sc}	Variation in mean service demand (uniformly distributed)	0.25

(a) Spinning versus blocking: At low system loads, the spinning policy performs better than the corresponding blocking policy because, at these system loads, blocking a task would likely result in leaving the corresponding processor idle. For higher system loads (for the parameters conside-

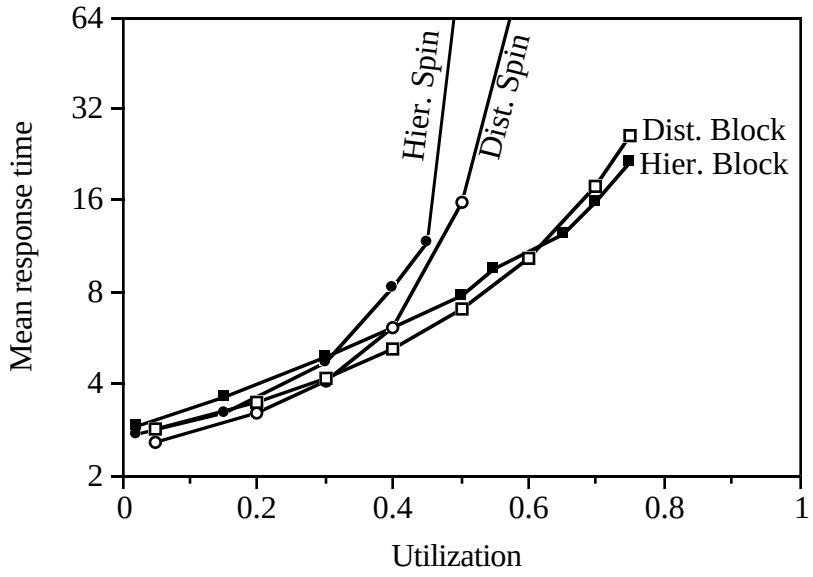


Figure 12 Performance as a function of useful utilization

red here, for system loads greater than 30%), however, blocking outperforms the spinning policy in both hierarchical and distributed organizations. The reasons for this are explained in the following paragraphs.

(b) *Performance of the blocking policy:* With the blocking policy, the distributed organization provides marginally better performance than the hierarchical organization when the system load is low to moderate. The reason for this is that, at these system loads, the load sharing property of the hierarchical organization does not really affect the performance because of the following: (i) jobs are restricted to have a maximum of *N* tasks per job, where *N* is the number of processors in the system, (ii) and the round robin task assignment policy used in the distributed organization tends to load balance the tasks at these low to moderate system loads like in the hierarchical organization. Since the distributed organization introduces smaller task scheduling overhead compared to that in the hierarchical organization, the distributed organization provides marginally better performance at these system loads. At higher system loads, the hierarchical organization performs marginally better than the distributed organization. This because the hierarchical organization provides better load sharing than the distributed organization and this factor becomes important at high system loads (and this load sharing feature more than compensates for the additional overhead involved in the hierarchical organization). Thus, in Figure 12, the hierarchical organization outperforms the distributed organization for system loads greater than 60%.

(c) *Performance of the spinning policy:* In contrast to the lock accessing workload, the spinning policy exhibits increased sensitivity to system load. This sensitivity can be explained as follows. In the lock accessing model, each task is independent except for accessing the critical section. Thus the spin times are not influenced by the system load. In the barrier synchronization model, the situation is different. When a task arrives at a barrier, it has to spin-wait until all of its siblings arrive at the barrier. This substantially results in wasted processor cycles (thus results in reduced useful utilization). The system load influences the spin times due to the following inherent restriction that spinning strategy imposes on task allocation. That is, tasks of a job are to be allocated to processors so that at most one task is assigned to a processor. If two tasks are assigned to a processor, one task busy waits and the other task would never be scheduled, leading to a deadlock. This deadlock situation arises because of run-to-completion scheduling employed here. This restriction on task allocation leads to load imbalances in the hierarchical and distributed task queue organizations. For example, assume that three consecutive applications with a size of 60, 49 and 47 are inserted in the root queue. The deadlock restriction forces the applications to be executed in a serial order. Therefore, the system saturates at a low system useful utilization.

The performance of the spinning policy in the distributed organization is better than that in the hierarchical organization for all system loads. The reason for this is that that load sharing property of the hierarchical organization is not exploited by the spinning policy with the barrier synchronization workload because of the restrictions imposed on task distribution, the number of tasks in a job etc. Since the hierarchical organization introduces additional overhead in scheduling tasks the distributed organization provides a better performance.

(d) *Hierarchical organization versus distributed organization:* For the barrier synchronization workload, blocking is the preferred choice (unless the system load is low) for the workload parameters considered here. When the blocking policy is used, both the hierarchical and the distributed organizations provide similar performance. For reasons explained before, for low to moderate system loads, the distributed organization performs marginally better. For higher system loads, the hierarchical organization performs better. As shown in the

following sections, higher variance in service demand tends to favour the hierarchical organization over the distributed organization. However, increased number of iterations and increased queue access times tend to favour the distributed organization because the hierarchical organization involves accessing more task queues.

7.2.2 Performance Sensitivity to Service Time Variance

Figure 13 presents the impact of the computation phase service demand variability on the performance of the spinning and blocking policies (ρ =50%). In the spinning policy, the total task spinning time in an application in each communication phase is $T * \mathrm{Max}(S_{Cij.}) - \sum_{i=1}^{T} S_{Cij}$ where S_{Cij} is the service demand of i^{th} task of the job in j^{th} computation phase. When the variability becomes large, the total spinning time dramatically increases. Thus, this policy is very sensitive to the service time variance. Since processors do not waste any time spinning, the service time variability (V_{Sc}), on the other hand, has only a marginal impact on the performance of the blocking policies. The blocking policy in the distributed organization exhibits more sensitivity mainly because of the lack of load sharing in distributed organization.

7.2.3 Impact of Granularity

Figure 14 shows the performance of the policies as a function of the maximum number of iterations in a task (Max_i) for a system load (ρ) of 50%. The task service demand is fixed at 2 time units. In contrast to the lock accessing workload, the job response time of the spinning policy increases dramatically with increasing number of iterations. This is due to the fact that, with increasing number of iterations, the spinning time of processors increases. As stated in Section 7.1, each communication phase requires an additional S_b time delay before the beginning of the next computation phase for the barrier mechanism, and all but one processor are spinning during this time period.

The blocking policies, however, are less affected (compared to the spinning policy) by the granularity as shown in Figure 14. The amount of time spent in barrier synchronizations increases in direct proportion to the number of iterations. The hierarchical organization is relatively more sensitive to the maximum number of iterations because is involves more task queue accesses.

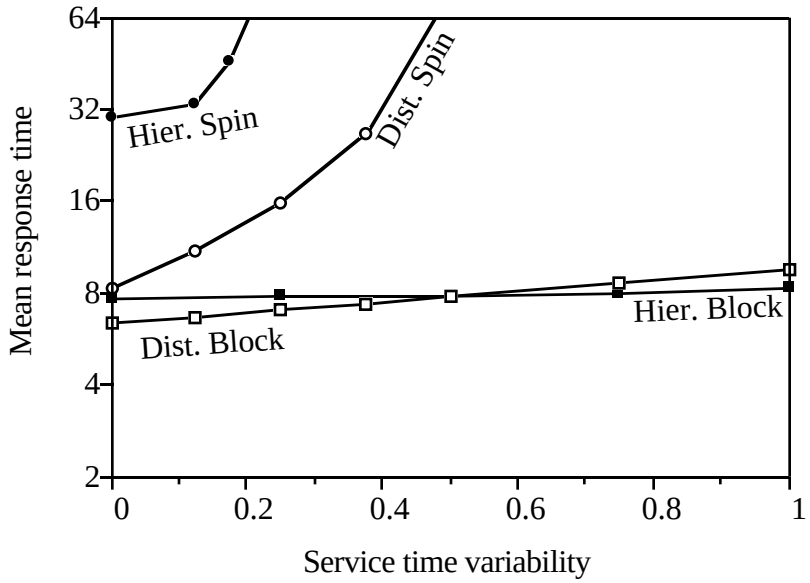


Figure 13 Impact of the service time variability V_{Sc}

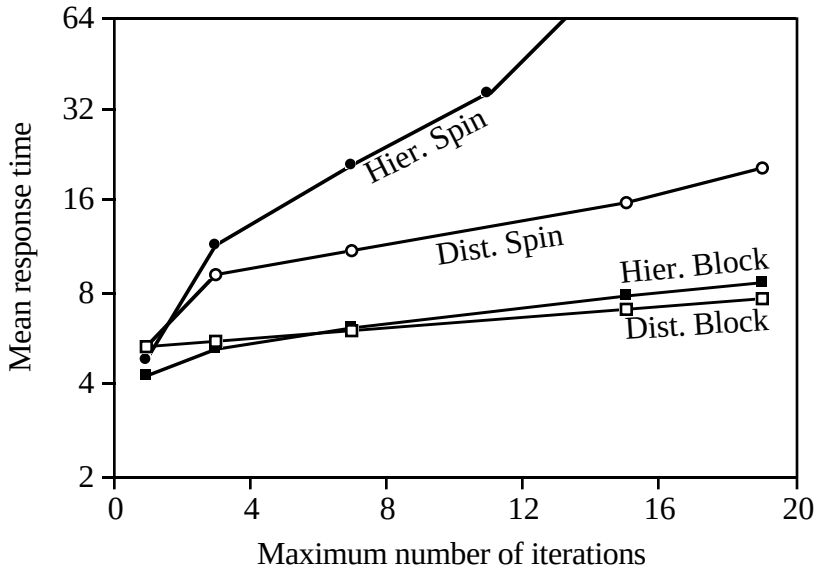


Figure 14 Performance sensitivity to the maximum number of iterations

7.2.4 Impact of the Queue Access Time

Figure 15 presents the impact of the queue access time (*f*) on performance of the policies for the barrier synchronization workload. The extreme sensitivity of the spinning policy in the hierarchical organization is due to the fact that, even though the useful utilization is only about 50%, the system is near saturation due to wasted processor cycles spinning at the barrier. Since the delay - introduced by the mutually exclusive access to the task queue - between when the first and last tasks of the same job are scheduled increases with queue access time, the value of the queue access time (*f*), has a direct impact on the performance of the spinning policy. The blocking policies and the spinning policy in the distributed organization, on the other hand, are not seriously affected by the queue access time because the system is not near saturation.

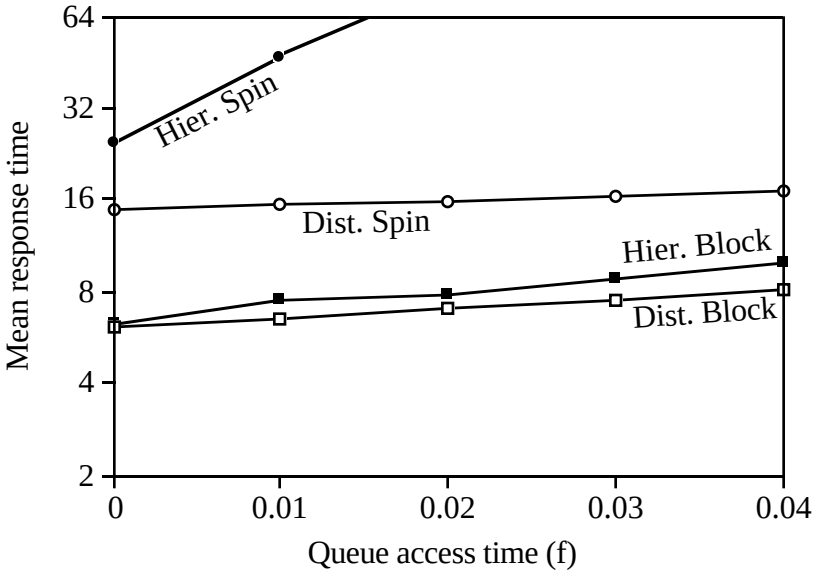


Figure 15 Impact of the queue access time (*f*)

8. CACHE EFFECTS

The results presented in this paper have ignored the possible cache effects on the performance. Previous work has shown that cache effects have significant impact on performance when preemptive scheduling policies are used [27]. A subsequent study by Vaswani and Zahorjan [29] have shown that voluntarily relinquishing processors will cause, on the other hand, cache effects to have only a minor impact on performance for reasons discussed in [29]. Briefly, preemptions cause more frequent processor allocations and each task runs for a shorter time before being preempted. This leads to increased cache data survivability by reducing interference caused by other tasks. On the other hand, voluntarily relinquishing processors causes tasks to run for longer before giving up the processors and this increases the interference by other tasks. Thus cache effects on performance are substantially reduced.

In this paper, tasks voluntarily relinquish processors when the blocking strategy is used. Therefore, cache effects will not have a significant performance impact on performance. Since the spinning strategy provides perfect affinity (i.e., once a task runs on a processor it continues to run on the same processor until completion) taking the cache effects into consideration will cause improvement in performance to be much more than that associated with the blocking policies. For the lock accessing workload, we have shown that spinning is the preferred choice in most cases and considering cache effects would only strengthen this conclusion.

For the barrier synchronization workload, the influence of caching on the performance would be even weaker than in the lock accessing workload. This is because, tasks relinquish processors only after completing a computation phase (i.e., when waiting at the barrier). In the next computation phase, the tasks will most likely work on a new set of data and therefore that data that remains in cache may not be useful. Thus the effect of cache is expected to be marginal and, therefore, do not substantially change the conclusions derived for this workload.

9. CONCLUSIONS

We have considered the trade-offs between spinning and blocking in the context of hierarchical and distributed run queue organizations. With the exception of Markatos et al. [20], all previous studies have assumed a centralized task queue organization. In addition, round robin task scheduling policy has been used in the previous studies. In contrast, we use the run-to-completion (with voluntary blocking, if blocking is used) scheduling policy. Previous studies indicate that this policy provides a better performance than the round robin policy.

We have considered two canonical workload types: lock accessing and barrier synchronization. For the lock accessing workload model, the spinning policy performs better than the blocking policies except when the lock holding time is large. When the spinning policy is used, the distributed organization performs as well as or better than the hierarchical organization . For small lock holding times, both organizations provide similar performance when the spinning policy is used. When the blocking policy is used, the hierarchical organization performs better. However, for the lock accessing workload, using the blocking policy results in substantial performance deterioration. Thus the distributed organization with the spinning policy performs better for the lock accessing type of workload.

For the barrier synchronization workload, blocking is the preferred choice (unless the system load is low) for the workload parameters considered here. When the blocking policy is used, both the hierarchical and the distributed organizations provide similar performance. For low to moderate system loads, the distributed organization performs marginally better. For higher system loads, the hierarchical organization performs better. It is also shown that higher variation in service demand tends to favour the hierarchical organization over the distributed organization. However, increased number of iterations and increased queue access times tend to favour the distributed organization.

In conclusion, the results presented here show that, when fine grain synchronization is required, the distributed organization with round robin task assignment is preferred. However, for large granularity tasks, the performance of the distributed organization is unacceptable and the hierarchical organization should be used. Note that the distributed organization is embedded into the hierarchical organization. Thus, for

coarse granularity parallel applications, the hierarchical organization with its load sharing feature can be used; for fine-granularity parallel applications, the hierarchy of queues can be circumvented and round robin task assignment can be done on processor local queues as in the distributed organization. Therefore, the hierarchical organization is useful in general-purpose large-scale shared-memory multiprocessors.

ACKNOWLEDGEMENTS

We thank Dr. Giacomo Sechi for handling the paper in a timely manner and the referees for their constructive comments. Most of Section 3 is based on [10]. We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University. Part of this work was done while S. P. Cheng was at the School of Computer Science, Carleton University.

REFERENCES

[1] T. E. Anderson, The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Parallel and Distributed Systems*, 1(1990) 6-16.

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy, The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Computers*, C-38(1989) 1631-1644.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Trans. Computer Systems*, 10(1992), 53-79.

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

[5] D. L. Black, Scheduling Support for Concurrency and Parallelism in the Mach Operating System, *IEEE Computer*, 23(1990) 35-43.

[6] S. P. Cheng, *Hierarchical task queue organization for multiprocessor systems*, M.C.S. Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1992.

[7] S. P. Cheng and S. P. Dandamudi, Scheduling in parallel systems with a hierarchical organization of tasks, in: *Proc. ACM Int. Conf. on Supercomputing*, (Washington, D.C., 1992) 377-386.

[8] S. P. Dandamudi, A Comparison of task scheduling strategies for multiprocessor systems, in: *Proc. IEEE Symp. Parallel and Distributed Processing*, (Dallas, Texas, 1991) 423-426.

[9] S. P. Dandamudi, Performance implications of task routing and task scheduling strategies for multiprocessor systems, in: *Proc. IEEE Int. Conf. Massively Parallel Computer Systems*, (Ischia, Italy, 1994) 348-353.

[10] S. P. Dandamudi and S. P. Cheng, A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems, *IEEE Trans. Parallel and Distributed Systems*, 6(1995) 1-16.

[11] A. Duda, On the Tradeoff between Parallelism and Communication, in Modelling Techniques and Tools for Computer Performance Evaluation, R. Puigjaner and D. Potier (Eds.) (Plenum, 1988).

[12] D. G. Feitelson and L. Rudolf, Gang Scheduling Performance Benefits for Fine-Grain Synchronization, *J. Parallel and Distributed Computing*, 16(1992) 306-318.

[13] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, Cedar – a large scale multiprocessor, in: *Proc. Int. Conf. Parallel Processing*, (1983) 524-529.

[14] P. Jones and A. Murta, Practical Experience of Run-Time Link Reconfiguration in a Multi-Transputer Machine, *Concurrency - Practice and Experience*, 1(1989) 171-189.

[15] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, Empirical studies of competitive spinning for a shared-memory multiprocessor, in: *Proc. of ACM Symp. Operating System Principles*,(Pacific Grove, 1991) 41-55.

[16] D. Lenoski et al. , The Stanford Dash Multiprocessor, *IEEE Computer*, 25(1992) 63-79.

[17] S. T. Leutenegger and M. K. Vernon, The performance of multiprogrammed multiprocessor scheduling policies, in: *Proc. of ACM Sigmetrics Conf.*, (Boulder, Colorado, 1990) 226-236.

[18] T. Lovett and S. Thakkar, The symmetry multiprocessor system, in: *Proc. Int. Conf. on Parallel Processing*, Vol. I, (1988) 303-310.

[19] S. Majumdar, D. L. Eager, and R. B. Bunt, Scheduling in multiprogrammed parallel systems, in: *Proc. ACM Sigmetrics Conf.*, (Santa Fe, 1988) 104-113.

[20] E. Markatos, M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, The effects of multiprogramming on barrier synchronization, in: *Proc. IEEE Symp. Parallel and Distributed Processing*, (Dallas, 1991) 662-669.

[21] J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. Computer Systems*, 9(1991) 21-65.

[22] A.K. Nanda, H. Shing, T.-H. Tzen, and L.M. Ni, Resource Contention in Shared-Memory Multiprocessors: A Parameterized Performance Degradation Model, *J. Parallel and Distributed Computing*, 12(1991) 313-328.

[23] R. Nelson and M. Squillante, Analysis of contention in multiprocessor scheduling, in: *Proc. Performance 90 - Int. Symp. Computer System Modelling, Measurement and Evaluation*, (1990) 391-405.

[24] L. M. Ni and C. E. Wu, Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems, *IEEE Trans. Software Engng.*, SE-15(1989) 327-334.

[25] G. F. Pfister and V. A. Norton, 'Hot Spot' Contention and Combining in Multistage Interconnection Networks, *IEEE Trans. Computers*, C-34(1985) 943-948.

[26] C.D. Polychronopoulous and D.J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. Computers*, C-36(1987) 1425-1439.

[27] M. S. Squillante and E. D. Lazowska, *Using processor-cache affinity information in shared memory multiprocessor scheduling*, Tech. Rep. 89-06-01, Department of Computer Science, University of Washington, 1989.

[28] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr., Firefly: A Multiprocessor Workstation, *IEEE Trans. Computers*, C-37(1988) 909-920.

[29] R. Vaswani and J. Zahorjan, The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors, in: *Proc. ACM Symp. Operating System Principles*, (Pacific Grove, 1991) 26-40.

[30] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, Distributing Hot-Spot Addressing in Large-Scale Multiprocessors, *IEEE Trans. Computers*, C-36(1987) 388-395.

[31] J. Zahorjan and C. McCann, Processor scheduling in shared memory multiprocessors, in: *Proc. of ACM Sigmetrics Conf.*, (Boulder, Colorado, 1990) 214-225.

[32] J. Zahorjan, E. Lazowska, and D. Eager, Spinning versus blocking in parallel systems with uncertainty, in: *Proc. Int. Symp. Performance Distributed Parallel Systems*, 1988. (Tech. Rep. 88-03-01, University of Washington)

[33] J. Zahorjan, E. Lazowska, and D. Eager, The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems, *IEEE Trans. Parallel and Distributed Systems*, 2(1991) 180-198.

[34] S. Zhou and T. Brecht, Processor pool-based scheduling for large-scale NUMA multiprocessors, in: *Proc. ACM Sigmetrics Conf.*, (1991) 133-142.