CS-1995-12

Scheduling to Reduce Memory Coherence Overhead on Coarse-Grain Multiprocessors[1]

Christopher Connelly   Carla Schlatter Ellis

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

March, 1995

[1]This research was supported in part by NSF grant CCR-9113170.

Scheduling to Reduce Memory Coherence Overhead on Coarse-Grain Multiprocessors*

Christopher Connelly Carla Schlatter Ellis

March, 1995

Abstract

Some Distributed Shared Memory (DSM) and Cache-Only Memory Architecture (COMA) multiprocessors keep processes near the data they reference by transparently replicating remote data in the processes' local memories. This automatic replication of data can impose substantial memory system overhead on an application since all replicated data must be kept coherent. We examine the effect of task scheduling on data replication and memory system overhead due to coherency requirements. We show that simple policies using programmer hints can reduce memory coherence overhead in our workload applications.

1 Introduction

Recent work has shown that the efficiency of shared memory, NUMA (Non-Uniform Memory Access) multiprocessors can be improved considerably by keeping threads and data near each other. This can be accomplished by one of several mechanisms: the OS can migrate or replicate an application's data pages [4, 5, 11, 12]; the OS or user-level thread scheduler may attempt to schedule threads on processors where they have previously executed and built up a certain amount of memory or cache state (e.g. *affinity scheduling* [15, 18, 21]); or programmer hints for task scheduling may be embedded in an object's specification in an object-oriented, task-queue based parallel language [6]. The task-queue model is widely used for parallel programming and is well suited for dynamically changing environments [20].

In contrast to NUMA multiprocessors, some machines based on Distributed Shared Memory (DSM) or Cache-Only Memory Architectures (COMA) guarantee that threads and their data are kept close together by automatically replicating shared pages and requiring memory references to be satisfied from local memory. While this policy helps keep threads and data on the same node,

*This research was supported in part by NSF grant CCR-9113170.

we speculate that it can also lead to increases in memory traffic if threads are scheduled without regard to the location of the data they access.

In this paper we examine the effect of task placement on memory system overhead on DSM platforms. Previous work has shown that even under optimal conditions applications implemented with heavy-weight threads benefit very little from sharing-based placement policies [19, 7]. However, previous work has also shown that applications based on a task-queue model can show appreciable performance gains from policies that place tasks near the data they reference or near other tasks that reference the same data [6, 15]. We have therefore restricted our work to applications conforming to a task-queue model.

Using a task-queue model in a DSM or COMA environment raises several questions that previous work has not addressed:

1. Do task scheduling policies affect an application's data sharing patterns? Do some task placement policies cause data to be more widely shared than others? Does this in turn increase memory system overhead?

2. Is there a role for the operating system in strengthening worker-data locality? For example, can threads be scheduled in such a way as to limit the number of processors sharing a data object?

3. Will increases in worker-data locality translate into reductions in memory coherence overhead, and thus into improvements in application performance?

In the following sections we will address these questions. The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 discusses the methodology and metrics we use in our experiments. Section 4 establishes workload parameters. Results are presented in Section 5, and Section 6 concludes the paper.

2 Related Work

Several recent papers have explicitly or implicitly addressed the impact of task scheduling on multiprocessor memory system overhead, application performance, and data sharing. Thekkath and Eggers [19] use trace-driven simulation to examine the effect of data-sharing based placement on the execution time of applications from the SPLASH [17] and the PRESTO [3] suites running on a multithreaded, multiprocessor architecture. Threads scheduled on the same processor share a cache, and the authors hypothesize that by placing threads sharing large amounts of data on the same processor, cache misses (compulsory and invalidate) and coherency overhead can be decreased and performance improved. However, after using off-line analysis to determine optimal thread placements, the authors observe no significant improvements. They conclude that sharing-based placement does not improve either execution time or cache performance, and that the determining factor in application performance is load balance, even in the presence of infinite caches. The authors attribute this to the uniform access to shared data by application threads and the small fraction of total references that are to shared data.

Stanford's COOL project [6] and Markatos and LeBlanc [14] examine the effect of task placement on performance for applications running in a task-queue environment. COOL, a parallel object-oriented environment, uses application hints to the run-time scheduler to enqueue a task on a node for which it has affinity. Markatos and LeBlanc simulate synthetic task-queue programs under a range of architectural parameters. Both papers report substantial performance gains when tasks are scheduled on compute nodes already containing data that they reference. Although both [6] and [14] allow for the caching of remote data, both platforms implement a fixed home node for VM pages, meaning that pages neither migrate nor are replicated. In this context, the ability to place a task on the home node of its data can eliminate a substantial number of remote cache misses and greatly improve execution times.

Chandra *et al.* [5] have investigated the effect of OS scheduling and page migration policies on the cache behavior and execution time of applications running on the Stanford DASH [13], thus examining the interaction between data sharing patterns and locality management policies. They do not investigate the interaction of page migration or replication (which is under implementation) and the task scheduling policies of COOL (also implemented on the DASH).

Our work differs from the previous work in that we examine a DSM multiprocessor in which data is migrated and replicated automatically, and all memory references are satisfied from local memory. In this context, the question of keeping threads near the data they reference becomes a question of managing memory coherence overhead.

3 Methodology

In the following sections we will attempt to answer the questions outlined in the introduction. To examine the role of the OS in task placement and determine if task placement affects the sharing of data objects, we measure the number of processors mapping a data page, the *worker-set size*, at a large number of post-initialization sample points under each placement policy. Results are presented as a series of worker-set size distribution plots. To examine the effect of task placement on memory coherence overhead, we measure coherency traffic (bytes transfered in update messages to maintain internode coherency and page transfers to replicate data) under each policy. Finally, we look at the impact of task placement on parallel execution time.

All results presented in this paper were obtained using a generic DSM memory hierarchy simulator running with Stanford's Tango multiprocessor simulator

[9].

3.1 Task-queue model

Our programming model is loosely modeled after the Uniform System [2] and assumes that an application spawns a single, non-migratory *server process* for each processor assigned to the application. Each server process executes for the duration of the application, fetching *tasks* from user-level ready queues, executing them, and searching for new tasks. These tasks tend to be short and are run to completion. Tasks may create new tasks, and may be nested.

To minimize contention, task queues are distributed throughout the multiprocessor. Each processor has both private and public task queues. Tasks in the private queue are always executed before tasks in the public queue, and are guaranteed to be executed by the owner of the private queue. Private-queue tasks may be enqueued by any process. Load balance is maintained by allowing idle server processes to steal tasks from the public queues of other server processes. To reduce the overhead of fetching a potentially large number of short-lived tasks individually, a server-process may dequeue several tasks at a time. The number of tasks dequeued in a single fetch operation is known as the *chunk size*, and is a function of application characteristics and task placement policy (see Section 4.1).

3.2 Workload applications

The target workload consists of four applications from Stanford's SPLASH suite rewritten to conform to a task-queue environment. *Barnes*, *mp3d* and *water* are large-scale scientific applications, while *cholesky* is a scientific kernel. Barnes is an implementation of the Barnes-Hut algorithm for an n-body gravitational problem; mp3d simulates rarefied fluid flow; and water is an n-body molecular dynamics application. Cholesky performs parallel Cholesky factorization on a sparse matrix. Detailed descriptions of each application are available in [17].

The difficulty of porting the workload to a task-queue environment varied widely from application to application. On the one hand, the SPLASH version of cholesky already conformed to our model, so we were able to simply replace calls to the cholesky task management functions with calls to our more general functions. On the other hand, the original versions of water and mp3d both statically partition work, so rewriting them to conform to a task-queue model required somewhat more work.

To avoid confusion concerning application versions, the original versions of each application are hereafter referred to with the suffix "-splash" (e.g. barnes-splash), while the task-queue versions are referred to with the suffix "-tq" (e.g. barnes-tq).

3.2.1 Important application characteristics

Barnes. Two types of tasks are used in the barnes-tq application. First, the master thread creates a single "top-level" task for each processor in the partition; these tasks run for the duration of the application. Each of the top-level tasks is responsible for moving a set of bodies through a gravitational n-body simulation through a time-step consisting of several distinct phases. Approximately 90% of parallel execution time is spent in the force computation phase [17]. Top-level tasks generate a new task to correspond to the force computation for each body in each time step. A 4096 byte VM page will hold data structures corresponding to approximately 40 tasks. Since each top-level task creates a force-computation task for each body in its set and these bodies reside in contiguous portions of the virtual address space, we expect placement policies that enqueue tasks on the generating node to work relatively well.

Cholesky. Tasks in the cholesky-tq application have been optimized to increase the amount of work done between task fetch operations, at the expense of some potential load imbalance [17]. Tasks may create one or more new tasks, though unlike barnes-tq, these are not executed until the creator task has terminated. Because a new task often shares data with its creator task, we expect local task placement policies to work fairly well.

Mp3d. As with barnes-tq, two types of tasks are used to implement mp3d-tq: a top-level task running for the duration of the application, and a large number of short-lived tasks used to implement the *move* phase of each time step, which accounts for roughly 90% of execution time [17]. We expect policies that place tasks on the generating node to perform relatively well.

Water. Water-tq uses a master-slave task queue model in which a single "master" thread creates all tasks, and many "slave" server processes fetch the tasks and execute them. The master process also executes tasks, in addition to controlling the flow of computation. Because water-tq contains a relatively small number of tasks, this model achieves acceptable performance. However, unlike cholesky-tq and barnes-tq, we expect local placement policies to create load imbalance and large worker-set sizes as all tasks are enqueued on a single node. On the other hand, policies that distribute the tasks more evenly and allow tasks to build state on a given node may achieve better performance.

All the applications from the SPLASH suite were originally written for small-scale UMA multiprocessors. Though this limits their scalability, especially for the large scale multiprocessors which we study, the UMA programming model is widely viewed as the most convenient model for parallel programming. Providing the appearance of an UMA environment on NUMA, DSM and COMA machines is the goal of much current research. We acknowledge the likelihood of artifact due to the small-scale nature of the original target; however, more contemporary benchmarks are not widely available or accepted.

3.3 The target architecture

We simulate a generic DSM multiprocessor based loosely upon the Galactica Net [22]. We assume the multiprocessor consists of a number of compute nodes connected by a high speed network. Compute nodes consist of a memory module, network interface, and a processor. Internode data consistency is maintained with a distributed directory update protocol: for each shared page in a node, the interface module maintains a pointer to the next node in a *virtual sharing ring* of processors with copies of the page. When a shared page is written, a copy of the new value is sent around the ring to all participating nodes [23]. Release consistency [10] is supported to reduce update latency. A competitive update policy is used to invalidate stale pages: when the number of remote updates to a particular page between local references exceeds a threshold value, the local copy of the page is invalidated and the node is removed from the virtual sharing ring for that page. Pages are assumed to be 4096 bytes.

We model a write-update protocol with competitive invalidation to minimize the effect of the particular memory coherence policy on our results. Specifically, previous work has shown that a write-invalidate policy can make worker-sets appear artificially small, while a pure write-update policy can make worker-sets appear artificially large, especially if data tends to migrate from processor to processor [8]. A write-update, competitive invalidate policy will allow data to be actively shared, but prevent stale data from lingering on a node where it is no longer needed. It should prevent any gross distortions of worker-set size.

Communication and CPU timings are based on Galactica Net prototype figures [24]. Page invalidation thresholds and task chunk size are set on a per-application basis such that execution time is minimized for each application (see Section 4 and [8]).

3.4 Task placement policies

The placement policies we evaluate fall in two principal categories, depending on the structure of task queues. Policies which are *unblocked* associate a single public task queue with each processor; tasks assigned to the processor are simply enqueued in FIFO order. *Blocked* policies associate an array of queues with each processor. Under a blocked policy, a particular queue within the array of queues is chosen in such a way as to group a certain set of tasks together (e.g. tasks operating on a certain object). The difference between blocked and unblocked policies is important given that tasks can be stolen or fetched in chunks, rather than individually. This means that an unblocked chunk of tasks is likely to contain a number of unrelated tasks, while tasks in a blocked chunk are more likely to share data or contend for certain objects. We hypothesize that by executing related tasks on a single processor, data replication and the related coherency overhead can be reduced.

Task placement policy determines which processor will receive a newly generated task. If an array of queues is used, the placement policy also determines which of the processor's queues receives the task. The placement policies that we investigate in this paper are:

- **Random (RAND):** a new task is placed on a queue chosen at random. This serves as a baseline for our other placement policies.

- **Round Robin (RR):** task queues are selected in a round robin fashion. If more than one process creates tasks, then each creator maintains a separate pointer indicating the next queue to receive a task. This policy tends to balance computational load, and will schedule iterative tasks (e.g. time stepping a body) on the same processor over several time steps if the number of tasks in each time step is divisible by the number of processors. However, RR will spread tasks in a single coherency block over a large number of processors, so we expect worker-set sizes to be relatively large.

- **Memory Affinity (MEM):** task queues are chosen based on an application hint indicating the address of an object for which the task has affinity (e.g. a data structure the task references). An attempt is made to place the task on a node possessing a local copy of the object. (Full queues cause a task to be placed elsewhere.) This policy can reduce unnecessary data replication by placing tasks near the data they will use, and will allow data sharing patterns to evolve gracefully over time. When several nodes are found to possess copies of the affinity page, an attempt is made to balance load by distributing tasks among all such nodes. This policy requires a programmer hint in the form of a pointer to the affinity object, and the ability to query the OS for the location of a copy of the page containing the affinity object.

- **Local Affinity (LOCAL):** all tasks are placed on the queue of the processor that generates them. For applications such as barnes-tq and cholesky-tq, where tasks can create new tasks, we intuitively expect this policy to perform competitively. For applications such as water-tq, where a single master process creates all tasks, we expect this policy to lead to load imbalance.

- **Home Node Affinity (HOME):** application provides affinity hints, as in memory affinity placement, but tasks are placed on the "home node" of the object for which they have affinity.[1] Although this policy does not ensure that the affinity object is present on the selected node, it will allow iterative applications to build up state on a particular node. Home node affinity tends to schedule tasks on nodes where they previously executed,

[1] The concept of a home node on a DSM platform may correspond to the node maintaining directory and state information for a particular coherency block, or it may simply be a convenient way to distribute load.

and will balance computational load if affinity objects are uniformly distributed.

- **Blocked Home Node Affinity (BLOCK-HOME):** task queues are implemented as an array of queues. Like home node affinity, tasks are scheduled on the "home node" of the affinity object. Furthermore, tasks with affinity to objects in the same coherency block are placed on the same queue. This policy requires a programmer hint in the form of a pointer to the affinity object.
- **Blocked Memory Affinity (BLOCK-MEM):** as with memory affinity placement, tasks are scheduled on a node with a local copy of the affinity object. As with blocked policies, tasks operating on objects in the same coherency block are placed on the same queue. This policy requires a programmer hint in the form of a pointer to the affinity object, and the ability to query the OS for the location of a copy of the page containing the affinity object.
- **Blocked Local Affinity (BLOCK-LOCAL):** tasks are placed on the local queues. If an affinity hint is available, tasks are further grouped according to affinity object.

4 Setting parameters

Results in [8] show that for our architectural parameters, the most appropriate multiprocessor sizes for simulations of SPLASH suite applications are 32 processors for barnes-splash and water-splash, and 16 processors for cholesky-splash and mp3d-splash. The applications do not scale well on the target architecture for larger partitions. In the following sections, we focus on simulation results for these machine sizes.

Results in [7] show that finding an appropriate value for the page invalidation threshold is important for two reasons. First, the invalidation threshold can impact performance. If the threshold is too low, application pages will tend to be invalidated while they are still in use, and performance will suffer as the application spends cycles refetching pages. Conversely, a threshold that is too high will allow stale pages to linger on nodes where they are no longer used, imposing unnecessary coherency overhead on tasks that are using them. Second, the threshold value affects worker-set distribution, our primary metric for data sharing. Low thresholds make worker-sets appear artificially small while high thresholds make worker-sets appear artificially large. The threshold values used in this paper have been determined empirically such that execution time is minimized and worker-set sizes are stable. Invalidation thresholds for the workload applications are summarized in Table 1.

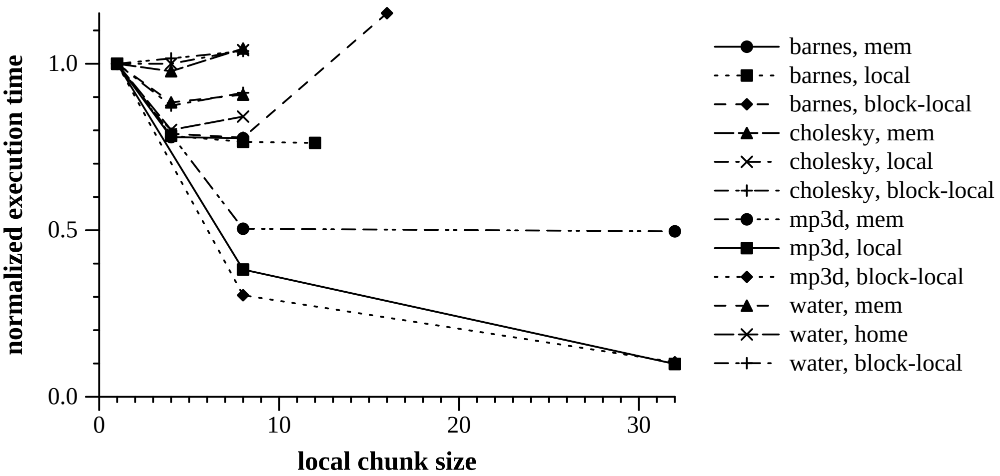

Figure 1: Execution times for workload, as a function of task chunk size.

4.1 Task chunk size

Chunk size refers to the number of tasks that may be dequeued by a server process in a single fetch operation. Server processes may want to fetch several tasks at a time to amortize the cost of shared queue operations, to reduce contention for shared queues, and to avoid splitting up groups of tasks with affinity for the same object. However, large chunk sizes may lead to load imbalance since fetched but unexecuted tasks cannot be stolen by idle server processes.

Figure 1 shows execution time for the workload applications as a function of chunk size. Times are normalized for each application with respect to the execution time at $chunksize = 1$. For reasons of clarity, a single placement policy is presented for each application; other policies do not differ qualitatively. Task chunking works well for barnes-tq, mp3d-tq, and water-tq, which have small tasks. Execution times decrease markedly as chunk size increases from one to four. At this point execution time increases for water-tq, due to increasing load imbalance, levels off for barnes-tq, and continues to fall for mp3d-tq. This is largely due to differences in the granularity of the applications: water-tq generates 512 tasks for each phase of computation (for our problem sizes), while barnes-tq and mp3d-tq generate 4096, and 64K, respectively. Task chunking performs poorly for cholesky-tq, which suffers from increasing load imbalance as chunk sizes grow.

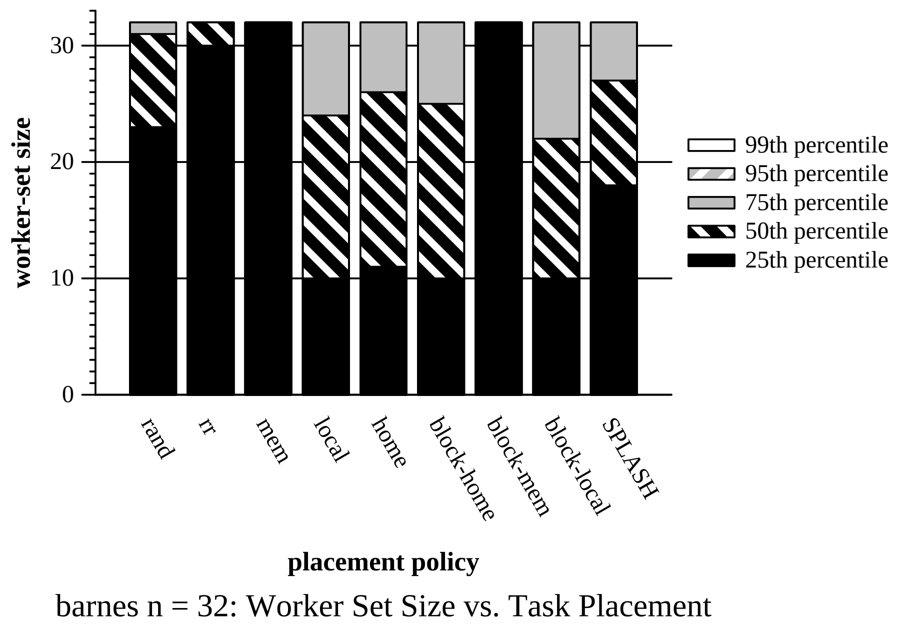

Figure 2: Barnes-tq: worker set size distribution versus task placement policy.

4.1.1 Summary of application parameters

Values for page invalidation threshold and task chunk size were determined empirically on a per application, per placement basis. Table 1 summarizes parameter values assumed in the remainder of this paper.

5 Results

In this section we analyze results from our process placement study. In Figures 2 – 5 we present worker-set size distribution graphs for the workload applications and also for the original SPLASH implementations. In the discussion below we will focus on worker-set size distribution of application pages, excluding pages used by the task-queue layer in order to separate the implementation specific effects of the latter.[2]

[2]Our shared memory allocation routines guarantee that task-queue and application pages do not overlap.

app	n	placement policy	inval. thresh.	chunk size
barnes	32	RAND	152	8
		RR	152	8
		MEM	304	8
		LOCAL	152	8
		HOME	152	8
		BLOCK-HOME	152	8
		BLOCK-MEM	225	8
		BLOCK-LOC	76	8
cholesky	16	RAND	76	1
		RR	76	1
		MEM	152	1
		LOCAL	76	1
		HOME	76	1
		BLOCK-HOME	76	1
		BLOCK-MEM	76	1
		BLOCK-LOC	76	1
mp3d	16	RAND	608	128
		RR	-	-
		MEM	608	64
		LOCAL	608	128
		HOME	608	128
		BLOCK-HOME	608	128
		BLOCK-MEM	-	-
		BLOCK-LOC	608	128
water	32	RAND	76	4
		RR	76	4
		MEM	152	4
		LOCAL	76	4
		HOME	76	4
		BLOCK-HOME	38	8
		BLOCK-MEM	38	8
		BLOCK-LOC	38	4

Table 1: Summary of parameters used in evaluating task placement policies. n indicates the number of processors simulated for each application. No parameters were set for mp3d-tq under RR or BLOCK-MEM because simulations of these policies did not complete.

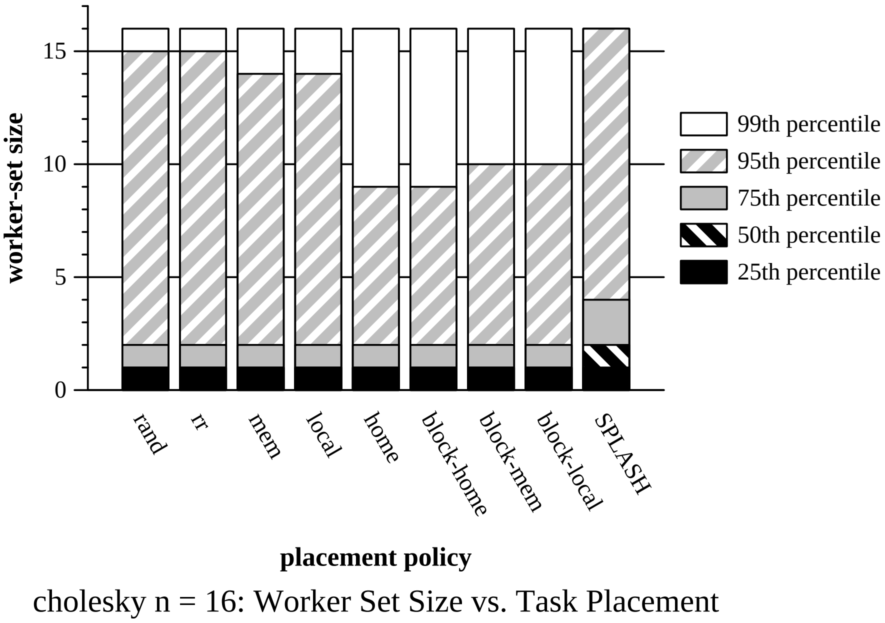

Figure 3: Cholesky-tq: worker set size distribution versus task placement policy.

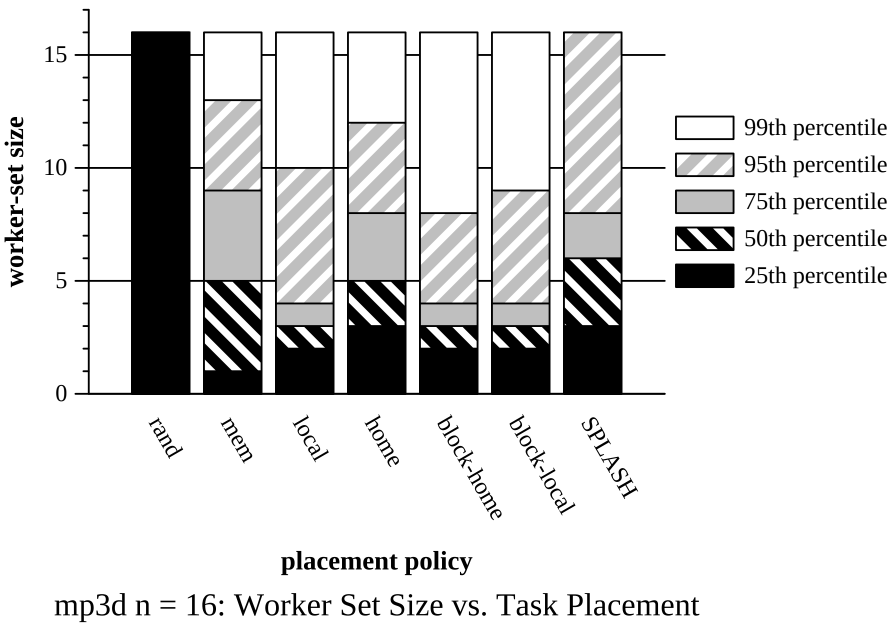

Figure 4: Mp3d-tq: worker set size distribution versus task placement policy.

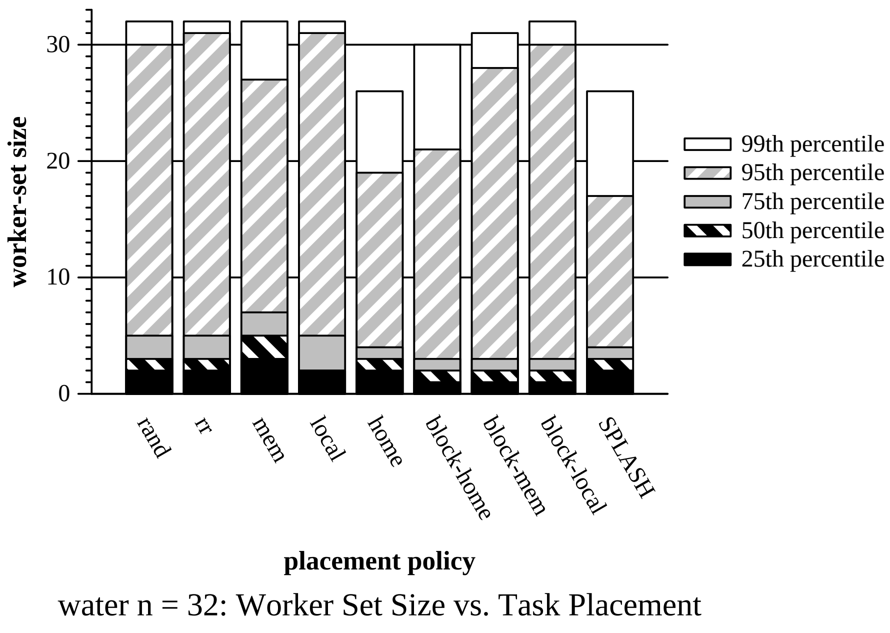

Figure 5: Water-tq: worker set size distribution versus task placement policy.

5.1 Worker-set size and task placement

Barnes. Results are summarized in Figure 2. As expected, local and blocked-local placement policies work relatively well, due to the static assignment of bodies to processes. Somewhat surprisingly, round-robin and random placement policies perform better (in terms of worker-set sizes) than memory affinity placements. RR placement tends to inflate worker-set sizes by placing adjacent tasks on adjacent processors rather than on the same processor. However, because the number of tasks generated by each top-level task is divisible by the number of processors, RR also places tasks on the same processor on successive time steps.[3] Random placement results in slightly smaller worker-set sizes than RR because tasks in the same coherency block are more often mapped to the same processor. MEM placement results in large worker-set sizes because it attempts to balance computational load by distributing tasks among all nodes with valid copies of the affinity page, rather than attempting to localize all tasks with affinity for a certain page on a single node. In addition, MEM uses a higher page invalidation threshold, which tends to inflate worker-set sizes. (We'll say more about this in the Section 5.2.) In contrast, local and home placement policies attempt to allow tasks to accumulate state on a particular node by scheduling tasks on the same node over several successive time steps. LOCAL placement causes tasks to be enqueued on the generating node. In the case of barnes-tq, each top-level task resides on a fixed processor, time-steps a fixed set of contiguous (in memory space) bodies, and generates force-computation tasks for each of its bodies. Thus, LOCAL placement results in tasks being enqueued with other tasks from the same coherency block, on a node that is likely to have a copy of the block. Likewise, HOME placement attempts to schedule all tasks associated with a given coherency block on the same node, though not necessarily on the originating node, and to distribute these blocks of tasks evenly throughout the multiprocessor. These two policies and their blocked versions result in large worker-set size decreases relative to RR, MEM, and RAND placement policies, and also relative to barnes-splash.

Cholesky. Unlike barnes-tq, cholesky-tq shows very little variability in worker-set size as a function of task placement policy. All placement policies result in roughly the same worker-set size distribution. As previously mentioned, blocked policies have little effect due to the dynamic nature of task generation. This also affects unblocked policies. Because of the small number of ready tasks at any time, tasks tend to be executed by the first idle server process rather than by the owner of the queue on which they are placed. Our statistics show that under HOME task placement, for instance, of 564 tasks generated, 395 are executed by remote server processes.

We note that the SPLASH version of cholesky uses a shared task-queue

[3] Placing tasks in the same coherency block on different processors is inherent to the round-robin policy while placing tasks on the same processor over multiple time steps is a coincidence of this particular problem size. RR placement does not do this for mp3d-tq, for instance.

implemented as part of the application module. Results for cholesky-splash presented in Figure 3 include task-queue pages in the computation of worker-set size distribution, while data for cholesky-tq does not. Including the task-queue pages in results for cholesky-tq does not change median or lower-quartile worker-set sizes for any of the placements examined, although it increases upper-quartile size for four placements (MEM and all blocked placements), and 95th percentile size for all placements except RAND. For all placements, worker-set sizes are still smaller for cholesky-tq than for cholesky-splash.

Mp3d. Mp3d-tq shows the highest sensitivity to placement of the four applications. In part this is due to the fine granularity of tasks. Random placement causes nearly every page to be shared by every processor, while local placements result in much smaller worker-sets (median size of three on a sixteen processor partition). Unlike the other applications, memory-affinity placement performs significantly better than random, although random has particularly large worker-sets. In terms of execution time, RAND in fact outperforms MEM. RR and BLOCK-MEM performed so poorly that they caused the simulation time counter to overflow. Both policies would tend to spread related tasks over a large number of nodes. No results are presented for these two policies.

Water. Water-tq, unlike barnes-tq and cholesky-tq, uses a master process to generate all application tasks, so we would expect local placement policies to lead to large worker sets. However, Figure 5 shows that LOCAL and BLOCK-LOCAL perform quite similarly to other policies. In effect, distributing tasks on the master's ready queues by coherency block (as in BLOCK-LOCAL) seems to mitigate the effect of placing all tasks on a single node by allowing blocks of related tasks to be fetched by remote server processes, and preventing unnecessary replication of data. And because tasks which are adjacent in memory space are enqueued consecutively, a sort of *de facto* task blocking exists under the LOCAL policy as well: remote server processes are likely to dequeue a group of tasks with affinity for the same coherency block.

Surprisingly, the median worker-set size for the MEM policy is larger than for LOCAL. As with barnes-tq, this seems to result from distributing load among all the nodes in the target worker-set, as well as from higher invalidation thresholds. In addition, the difference may be reinforced by the master-slave implementation of water-tq: consecutive tasks will correspond to adjacent (in memory space) data structures, and a chunk of tasks on the local (master's) queue will tend to operate on a small number of coherency blocks.[4] In contrast, the unblocked memory affinity policy will tend to spread tasks from a single block among all nodes with a copy of the block, and because the tasks are spread around, they will also tend to be more finely interleaved with tasks from other coherency blocks. This can lead to the replication of several coherency blocks when chunks of remote tasks are fetched. Worker-set sizes under BLOCK-MEM are comparable to other blocked policies, presumably because this task inter-

[4]The precise number will depend on chunk size and alignment.

leaving is avoided by blocking tasks, and because invalidation thresholds are lower.

For all applications we were able to improve worker-set size distribution over the original SPLASH versions. Most of the SPLASH applications already showed good worker-data locality, as evidenced by their generally small median worker-set sizes. Still, in most cases, median worker-set size was reduced just by moving to a task-queue environment, and in many cases it was reduced even further by the appropriate task placement policies. For instance, the median worker-set size for water-splash with 32 processors is three. Median worker-set size for water-tq is three or less for all but the worst placement policies. Likewise, median worker-set size for cholesky-splash is two for 16 processors, while median worker-set size for cholesky-tq is one for all placements. Much more variation is seen in the results for barnes-tq, which has the largest worker-set sizes of any of the SPLASH applications. Several placement policies result in significantly larger median worker-set sizes than the original SPLASH version, but several policies show significant improvement over barnes-splash. In particular, the local and block-local policies perform well.

5.2 Task placement and memory coherence overhead

Memory coherence traffic consists of data sent between nodes in the form of updates, page transfers, and remove-node-from-sharing-ring messages in order to maintain a consistent view of system state.[5] In this section we will examine the effect of task placement on the number of pages transfered and updates delivered.

Figure 6 shows total coherency traffic for the workload applications under the eight placement policies. We assume that each page transfer sends 4096 data bytes plus 8 address bytes, and each update message sends 8 data bytes and 8 address bytes. Update traffic counts the number of update bytes *received* (an update on a four-node sharing ring results in 16 bytes being received by four nodes—the sender receives a copy of the update as an acknowledgment—for a total of 64 update bytes). We note that with the exception of barnes-tq, placement policy seems to have only a small effect on application page transfers. In particular, policies designed to allow tasks to build up state on a given node perform comparably to random or round-robin policies. This would seem to indicate that tasks are unable to accumulate any significant amount of state on a node over successive iterations of an algorithm. Perhaps pages are shared widely enough that even when most tasks from a given page are scheduled on the same node, the affinity page will become stale on its home node and be invalidated over the course of most iterations. Alternatively, it could indicate that page transfers due to initial references to data structures by the "owning"

[5] Page invalidation signals arise from a local update counter, and so are not counted as part of the coherency traffic; however, when a page is invalidated, a message must be sent around the virtual sharing ring to prevent updates from being sent to the removed node.

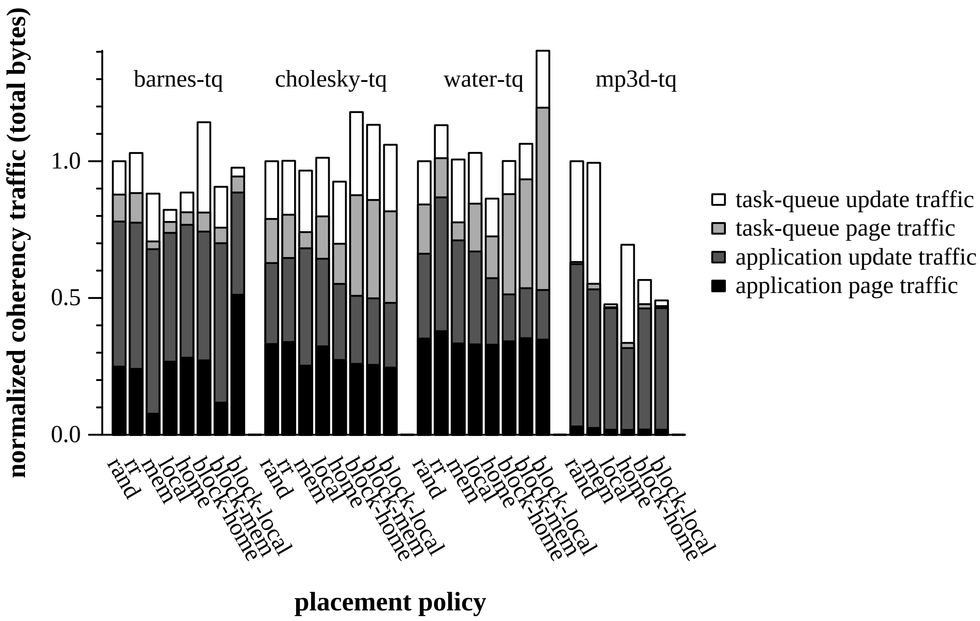

Figure 6: Memory coherence traffic as a function of task placement policy. Traffic consists of update messages and page transfers for both application and task-queue data. Data is normalized with respect to each application's total traffic under random task placement.

task (the task currently in charge of time stepping the data structure) comprise a small fraction of the total page transfers for the page containing the object. Either case would indicate high worker-set volatility as a possible performance bottleneck.

For barnes-tq MEM and BLOCK-MEM result in significantly fewer page transfers than other policies (roughly 69% and 55% fewer transfers, respectively, than RAND). In fact, the difference is caused primarily by differences in page invalidation threshold and only indirectly by task placement policy. Recall that page invalidation threshold is determined empirically such that application execution time is minimized (Section 4). From Table 1 we see that under MEM applications tend to run faster with higher invalidation thresholds than under other unblocked policies. In this case, the higher threshold makes it less likely that a page for which an unexecuted task has affinity will be invalidated before the task is executed. Application coherency traffic under BLOCK-MEM and BLOCK-HOME is comparable to unblocked policies. We note that task blocking is designed to minimize task fragmentation (spreading tasks operating on the same coherency block over several nodes); with median worker-set size in excess of 20 nodes for all policies, task fragmentation accounts for only a small fraction of coherency traffic. In general, we note that barnes-tq shows little sensitivity (in terms of coherency traffic) to placement policy. We attribute this to the wide sharing of application pages: reducing worker-set size by one or two processors simply does not have much of an impact on total traffic.

For cholesky-tq and water-tq, application coherency traffic is minimized by the three blocked placement policies (though total coherency traffic is relatively high for these policies). Because these applications have good worker-data locality, as indicated by their small median worker-set sizes, task blocking is able to reduce coherency traffic by keeping tasks from the same coherency block together in the presence of task stealing. Although the dynamic nature of task creation in cholesky limits the effectiveness of all placement policies, blocked policies are able to take advantage of the initial surplus of ready tasks to fetch chunks of related tasks, while unblocked policies cannot. Note that because the initial third of cholesky-tq tasks are created by processor 0, and have affinity for processor 0, almost all of these tasks are enqueued on node 0 under BLOCK-LOCAL and BLOCK-MEM.

Mp3d-tq and water-tq show somewhat more sensitivity to task placement than barnes-tq or cholesky-tq. Both have small median worker-set sizes (though mp3d-tq's vary considerably), so increases in worker-set size often results in a noticeable increase in coherency traffic. For instance, RR tends to distribute tasks from the same coherency block on a large number of processors, and results in a significant increase in coherency traffic for water-tq (31% more traffic than RAND and 69% more than HOME). Results are equally striking for mp3d-tq: under RAND the application requires 35% more coherency traffic than under LOCAL.

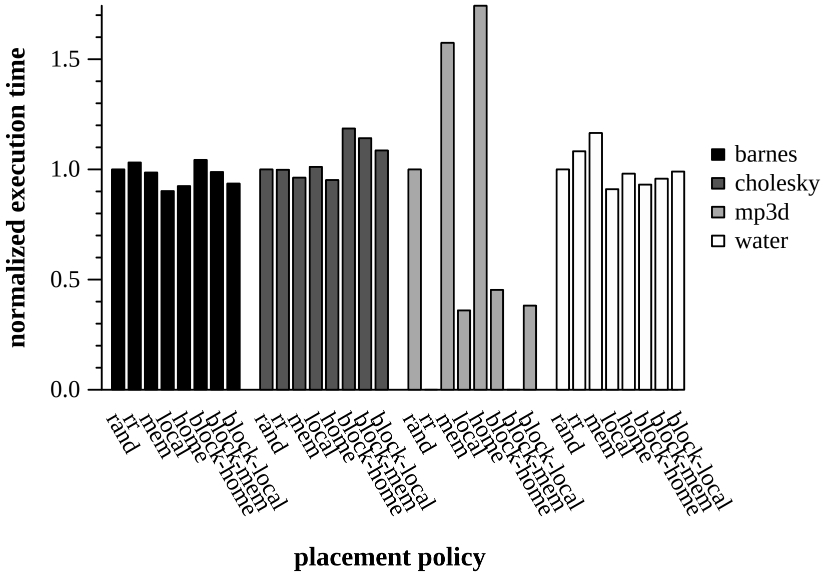

Figure 7: Execution times for workload applications versus task placement policy. Times are normalized with respect to each application's execution time under RAND.

5.3 Task placement and execution time

RAND, RR, MEM, and **BLOCK-MEM.** Execution times for the workload applications under the various placement policies are summarized in Figure 7. For all applications except cholesky-tq, random task placement results in slightly better performance than round-robin placement, indicating that load balancing is not the determining factor in application performance. (This is in contrast to [19].) However, as we have seen, load-imbalance is probably partially responsible for the poor performance of the blocked placement policies in cholesky-tq. Likewise, affinity based placement policies improve performance only marginally or not at all (this is in contrast to [6] and [14]). Both results are due to the nature of the DSM target architecture we simulate, which allows pages of memory to be cached automatically when referenced.

LOCAL. As anticipated, LOCAL performs relatively well, in terms of execution time, for barnes-tq (11% faster than RAND) and mp3d-tq (178% faster). Surprisingly, it also performs well for water-tq (10% faster than RAND), but is slightly slower than RAND for cholesky-tq. We note that the in the initial phase of cholesky-tq, LOCAL results in all tasks being enqueued on node 0, causing queue contention and task fragmentation, while the large amount of task stealing in the latter portion of cholesky-tq makes task placement virtually useless.

To understand why LOCAL performs better for water-tq than HOME, which has 15% less memory traffic (TQ and application combined), consider the manner in which tasks are created and enqueued by the master process. Under LOCAL all tasks are enqueued on the master's ready queue. A copy of this queue is likely to reside on the master's node. Enqueuing a large number of tasks will cause other instances of the queue to be invalidated. When the slave processes begin to look for tasks, each will fault *in parallel* on the master's task queue. After an initial period of contention for the single queue, the slave processes become staggered, and contention is reduced. Since slave page transfers are serviced in parallel, most of this transfer time is not spent on the critical path.[6] In fact, the master process is likely to be working during this time since it should not fault on its own queue. In contrast, under HOME, the master process must enqueue most tasks on remote ready queues which are not likely to be resident on the master node since they have presumably been heavily utilized by their owners during the previous computational phase. Thus, the master process will likely stall repeatedly while each queue is transfered in. Each of these transfers will be on the critical path since the master generates all tasks before starting the slaves. In addition, enqueuing tasks will result in the original copy of the queue becoming stale on the remote node, and another page transfer will be necessary when each slave process attempts to access its local list[7]. Assuming 31 (remote queue) page transfers per phase, four phases per time step, and 4 time steps over the course of the application, we estimate that HOME will incur roughly 500 more critical path page transfers than LOCAL, resulting in a 2,500,000 cycle performance hit.

HOME. Both HOME and LOCAL address the goals of load distribution and establishing task state over several time steps. HOME addresses load distribution explicitly, while LOCAL addresses it implicitly. A side effect of the HOME strategy is that processes often attempt to enqueue tasks on remote nodes and must wait for the remote queue object to be transfered in. While this may not increase total traffic, it does tend to move page transfers onto the critical task create and dequeue paths and thus increase execution time. This can be seen in barnes-tq, mp3d-tq, and water-tq. The lone exception is cholesky-tq, in which HOME results in much better load distribution during the initial phase of computation (recall that a single process creates the initial third of all tasks). In this case, reductions in worker-set fragmentation seem to compensate for critical path page transferring.

[6] Whether interconnection bandwidth is high enough to service a large number of page transfers in parallel is another matter. We note that in this case, all the slave processes are requesting copies of the same page, so an intelligent operating system should be able to handle the traffic.

[7] Although these transfers may be serviced in parallel, getting copies from the master process may be a bottleneck.

BLOCK-HOME and **BLOCK-LOCAL.** Our observations concerning LOCAL and HOME also hold for BLOCK-LOCAL and BLOCK-HOME. We note that cholesky-tq runs slightly faster under BLOCK-LOCAL than BLOCK-HOME (while the opposite was true under LOCAL and HOME) because task blocking prevents the worker-set fragmentation that resulted in poor performance for LOCAL. (Note however that due to the overhead of searching additional queues for a small number of tasks, unblocked policies outperform blocked policies for cholesky-tq.) Likewise, water-tq performs slightly better under BLOCK-HOME than under BLOCK-LOCAL. This results from the need to search a large number of empty queues on other nodes in order to find ready tasks on node 0. Figure 6 shows that there is a tremendous increase in task-queue page transfers under BLOCK-LOCAL; however, the performance impact is much smaller since most queue searching is done in parallel.

6 Conclusions

We used simulations of real programs to examine the effect of task placement on memory coherence overhead and application performance. Our results show that the appropriate task placement policies can result in significant reductions in application-level coherency traffic and moderate improvements in run time. We were able to observe improvements of up to 11% in execution time and 25% in application memory coherence traffic. For mp3d-tq, even larger improvements were observed. In addition we conclude:

- Decoupling logical tasks from the processes that execute them can reduce data sharing. In most cases, just porting the workload to a task-queue environment resulted in reductions in worker-set size. Using appropriate task placement policies can sometimes further reduce sharing, depending on an application's data access patterns.

- Very simple policies such as LOCAL are surprisingly robust. Our results show that even when this policy causes all tasks to be enqueued on a single node, the application is still competitive with other policies.

- Task blocking seems to offer promise as a way to reduce application coherency traffic, but only if it can be implemented with low overhead. In two of our applications, cholesky-tq and water-tq, task blocking resulted in significant reductions in application coherency traffic. However, task-queue layer coherency traffic was high enough in these cases that total traffic was no lower than for unblocked policies. Separately tuning task-queue and application parameters should allow task-queue traffic to be reduced while keeping application traffic low. In addition, applications whose task creation behavior evolves over the course of application lifetime (e.g. cholesky) may benefit from self-tuning placement policies: a blocked policy may be appropriate for the initial phase of task generation, while an unblocked policy may be more appropriate later.

- The interaction of task generation policies and memory coherence policies can have a large and sometimes counter-intuitive impact on performance. For instance, attempting to distribute load evenly by enqueuing tasks on remote nodes can force page transfers onto the critical path and degrade performance.
- Worker-set volatility either limits a task's ability to accumulate state on a particular node, or makes such state accumulation into a small fraction of total data traffic. This may indicate significant migratory sharing, or it may indicate page bouncing.

Finally, we believe that our results are conservative in several important respects. First, better tuning of the task-queue layer will make the modest gains we observed more significant as overhead is reduced. Second, moving to a task-queue model facilitates the use of techniques such as First-Class threads [1, 16] which can further reduce execution time. Third, given the reductions in memory traffic we achieved, we speculate that as internode communication continues to become more expensive relative to processor speed, our policies will achieve better performance.

References

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

[2] BBN. The Uniform System approach to programming the butterfly parallel processor. Technical Report Number 6149, Bolt Beranek and Newman Adv. Computers Inc., October 1985.

[3] B. Bershad, E. Lazowska, and H. Levy. PRESTO: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988. TR 87-09-01.

[4] W. Bolosky, M. Scott, and R. Fitzgerald. Simple but effective techniques for NUMA memory management. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[5] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings, Sixth International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 12–24, 1994.

[6] Rohit Chandra, Anoop Gupta, and John L. Hennessey. Integrating concurrency and data abstraction in the COOL programming language. *IEEE Computer*, 27(2), February 1994.

[7] Christopher Connelly and Carla S. Ellis. Workload characterization and locality management for coarse grain multiprocessors. Technical Report CS-1994-30, Duke University, September 1994.

[8] Christopher Connelly and Carla S. Ellis. A workload characterization for coarse grain multiprocessors. In *International Parallel Processing Symposium*, April 1995. To appear.

[9] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 99–107, 1991.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[11] Rick LaRowe and Carla Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[12] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The robustness of NUMA memory management. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 137–151, October 1991.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[14] E. Markatos and T. LeBlanc. Load balancing versus locality management in shared memory multiprocessors. Technical Report 399, University of Rochester, October 1991.

[15] Evangelos Markatos. Scheduling for locality in shared-memory multiprocessors. Technical Report 457, University of Rochester, May 1993.

[16] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 110–21. Association for Computing Machinery SIGOPS, October 1991.

[17] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):1–44, 1992.

[18] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared memory multiprocessor scheduling. Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, 1889. To Appear IEEE Transactions on Parallel and Distributed Systems.

[19] Radhika Thekkath and Susan J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 176–186, April 1994.

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

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

[22] Andrew Wilson, Marc Teller, Thomas Probert, Dyung Le, and Richard LaRowe. Lynx/Galactica Net: A distributed, cache coherent multiprocessing system. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 416–426, 1992.

[23] Andrew W. Wilson and Richard P. LaRowe. Hiding shared memory reference latency on the galactica net distributed shared memory arcitecture. *Journal of Parallel and Distributed Computing*, to appear.

[24] Andrew W. Wilson Jr., Richard P. LaRowe Jr., Robert J. Ionta, Ralph P. Valentineo, Beeching Hu, Peter R. Breton, and Pocheong Lau. Update propagation in the Galactica Net distributed shared memory architecture. Technical Report CHPC TR 93-007, Center for High Performance Computing, Worcester Polytechnic Institute, 1993.