Operating System Support for Improving Data Locality on CC-NUMA Compute Servers

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum
Computer Systems Laboratory, Stanford University, CA 94305
http://www-flash.stanford.edu

Abstract

The dominant architecture for the next generation of shared-memory multiprocessors is CC-NUMA (cache-coherent non-uniform memory architecture). These machines are attractive as compute servers because they provide transparent access to local and remote memory. However, the access latency to remote memory is 3 to 5 times the latency to local memory. CC-NOW machines provide the benefits of cache coherence to networks of workstations, at the cost of even higher remote access latency. Given the large remote access latencies of these architectures, data locality is potentially the most important performance issue. Using realistic workloads, we study the performance improvements provided by OS supported dynamic page migration and replication. Analyzing our kernel-based implementation, we provide a detailed breakdown of the costs. We show that sampling of cache misses can be used to reduce cost without compromising performance, and that TLB misses may not be a consistent approximation for cache misses. Finally, our experiments show that dynamic page migration and replication can substantially increase application performance, as much as 30%, and reduce contention for resources in the NUMA memory system.

1. Introduction

Shared-memory multiprocessors are attractive as compute servers because they provide tight coupling of processor and memory resources. However, the memory bus on current bus-based multiprocessors is quickly becoming a bottleneck as processors get faster and as these systems scale to larger number of processors. The architectural solution to this problem is to divide the machine into a number of nodes, each node consisting of one or more processors and a part of main memory. These nodes are connected using scalable interconnect technology, and cache-coherence is provided using directory techniques. There are two variations to cache-coherent shared-memory systems. The first is the more traditional CC-NUMA machine (cache-coherent non-uniform memory-access) that uses custom interconnect technology — Stanford DASH [LLG+90], MIT Alewife [ACD+91], Sequent STiNG[LoC96], and Convex Exemplar. The other is the CC-NOW machine (cache-coherent networks of workstations) that uses general-purpose interconnect technology with the nodes being users' workstations — the Stanford Distributed FLASH proposal [Kus+94] and the SUN s3.mp architecture [NAB+95]. On both these architectures, the access to local memory is optimized to provide workstation-like latencies. The remote access latency is expected to be 3 to 5 times the local access latency for CC-NUMA machines and 10 to 20 times for CC-NOW.

The high remote latency makes data locality potentially the most important performance issue because a large number of cache misses to remote memory could severely impact application performance, and increase the utilization of memory system components. This is compounded in compute-server workloads where processes will need to be moved from one processor to another for load balancing. Page migration can be used to keep data local to the process when it is moved. Replication of pages is desirable where data is heavily read-shared between many processors.

Using realistic workloads on a real implementation, we show that the operating system can substantially improve data locality by migrating and replicating pages. On CC-NUMA architectures, our implementation improves the workload execution time as much as 29% over a first touch policy. Improving data locality also has a system-wide benefit, reducing the contention for resources in the NUMA memory system. Our results also show that both page migration and page replication are necessary, and that neither on its own suffices. For most of the workloads, migration and replication provide substantially better performance than the optimal static page placement.

CC-NUMA architectures have the ability to cache remote data, therefore pages must be moved selectively and efficiently. Pages that are write-shared cannot benefit from migration or replication, and we show that our policy is robust in the face of workloads which exhibit this type of sharing. Our studies show that TLB miss information is not adequate to consistently select the appropriate pages to move. Information about cache misses is needed, and we identify the architectural support necessary. Finally, current SMP operating systems are not designed to support migration and replication, and large kernel overheads can potentially negate any performance improvement. We describe optimizations we made to reduce the kernel overheads, and point out other remaining kernel bottlenecks.

The rest of this paper is organized as follows. Section 3 presents a framework within which various design issues and policy choices are developed. Section 4 describes the kernel mechanisms needed. In Sections 5 and 6 we describe our experimental environment and workloads. In Section 7, we present the costs and benefits of page migration and replication from experimental runs. In Section 8, we explore a number of interesting issues related to migration and replication.

2. Related Work

Significant work related to page migration and replication has been done on architectures other than CC-NUMA. Much of this has focused on implementing shared memory in software, including IVY [Li88], Munin and Treadmarks [BCZ90], Midway [BZS93], Jade [RSL92], and SAM [ScL94]. In these systems, data

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS VII 10/96 MA, USA
© 1996 ACM 0-89791-767-7/96/0010...$3.50

replication or migration, at a page or object level, is required for correctness when a processor references a remote datum. Our work differs from these systems because the hardware on CC-NUMA machines provides the shared memory abstraction; page replication and migration is purely an optimization. Other work has focused on migration and replication on shared-memory machines that are not cache-coherent, such as the BBN-Butterfly and the IBM ACE [BSF+91, LEK91, CoF89, Hol89]. In these systems, migration and replication is triggered by page faults not cache misses, and although some mechanisms developed are relevant to our study, such as freezing and defrosting of pages, the policies used and the performance results are not, because of the lack of coherent caches.

The ability of CC-NUMA machines to cache remote data substantially changes the potential benefits of migrating and replicating pages. In this environment, although the remote access latency is on the order of a microsecond, with cache-coherence, subsequent accesses hit in the local processor cache and take only a few nanoseconds. If the workload exhibits good cache locality, cache-coherent systems will derive less benefits from migration and replication of pages. Therefore coherent caches force us to be more selective when moving pages, because the page movement overheads we can tolerate are much lower.

More directly within the CC-NUMA model, Black et al. [BGW89] proposed a competitive strategy to migrate and replicate pages using special hardware support (a counter for each page-frame per processor). The workloads evaluated were a few small scientific applications from the SPLASH suite [SWG92], run one at a time, and only user-mode data references were considered. In contrast, we evaluate workloads of relevant, complex applications with multiprogramming, and both kernel and user references are studied. The policy space explored is also quite different. The work by Chandra et al [CDV+94] on the Stanford DASH multiprocessor, while being closely related, is different in three important respects: (i) focus of their work was process scheduling and migration, while we build on their work and study memory locality through both migration *and* replication; (ii) the workloads in our study are more varied; and (iii) our study is based on the SimOS simulation environment [RHW+95]. The SimOS environment used here is realistic and flexible (see Section 5), and allows us to choose architectural parameters — processor speeds, cache sizes, and latencies — more relevant to today's systems than when DASH was designed six years ago.

3. Policy Framework

We begin with a detailed analysis of the problem we are trying to solve, in terms of the benefits and the costs of page migration and replication. We then present the policy for page migration and replication that will serve as the framework for the rest of the study.

3.1 Problem Statement

Our goal is to minimize the runtime for the user's workload by reducing the component due to memory stall. On CC-NUMA machines, memory stall can be reduced by converting remote misses to local misses through the migration and replication of pages. To maximize the overall reduction in execution time, it is also important to keep the costs of migration and replication to a minimum. Therefore, we need to be concerned about both the savings from data locality and the costs of migration and replication.

The first aspect, improving data locality, involves finding the pages that suffer the most remote-misses and converting them to local-

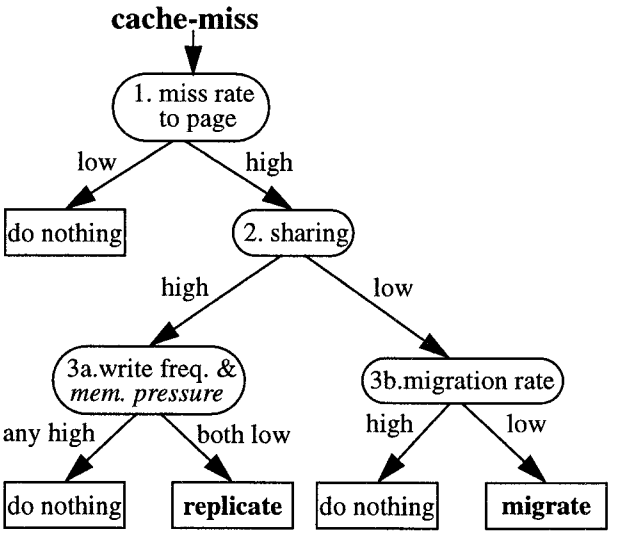


FIGURE 1: Replication/Migration decision tree.
The flowchart shows the decision process for a page to which a cache miss is taken. The possibilities are to replicate, migrate, or do nothing.

misses if possible. The access patterns to the page can be broadly classified into three groups that determine the policy action to be taken. The first group consists of pages that are *primarily accessed by a single process*. These pages are candidates for migration when the process accessing them migrates to another processor, and they include: data pages of sequential applications; data pages of parallel applications where the accesses from the processes are to disjoint sections of the data; and the code of sequential applications, when only one instance of the application is running on the machine. The second group consists of pages *accessed by multiple processes, but with mostly read accesses*. These pages are candidates for replication, and they include: code pages of parallel applications; code pages of concurrently executing copies of a sequential application; and read-mostly data pages of parallel applications. The third group consists of pages *accessed by multiple processes, but with both read and write accesses*. These pages are not candidates for either replication or migration, and they include the data pages of parallel applications where there is fine grain sharing with updates from multiple processors.

The second aspect of the problem concerns the costs of page migration and replication. We classify these costs into four categories. The first cost is that of *gathering information* to help determine when and what pages to migrate or replicate. The options we consider are to keep counts of all cache misses (supported by Stanford FLASH) or all TLB misses (many processors have software handling of TLB misses), or to use time sampling of either cache misses or TLB misses. The second cost is that of the *kernel overhead* for migration and replication. This cost includes the overheads for allocating a new page, removing all mappings to the old physical page, establishing the new mappings, flushing TLBs to maintain coherence and eliminating replicas on a store to a replicated page, to name a few. The third is the cost of *data movement* to physically copy a page from one location to another. The fourth is the indirect cost of *increased memory use* (memory pressure) resulting from page replication.

3.2 Solution Framework

The next step is to design a policy that uses available cache-miss information to decide if and when to migrate or replicate a page.

Parameter	Semantics
Reset Interval	Number of clock cycles after which all counters are reset.
Trigger Threshold	Number of misses after which page is considered hot and a migration/replication decision is triggered.
Sharing Threshold	Number of misses from another processor, making a page a candidate for replication instead of migration.
Write Threshold	Number of writes after which a page is not considered for replication
Migrate Threshold	Number of migrates after which a page is not considered for migration.

TABLE 1: Key parameters used by the policy. These parameters are used along with the counters to approximate rates. The counters used by the policy include a per-page per-processor miss counter, a per-page write counter, and a per-page migrate counter.

Figure 1 shows a decision tree for our policy. We assume that action may be triggered on any cache miss, but the decision tree is independent of the metric used.

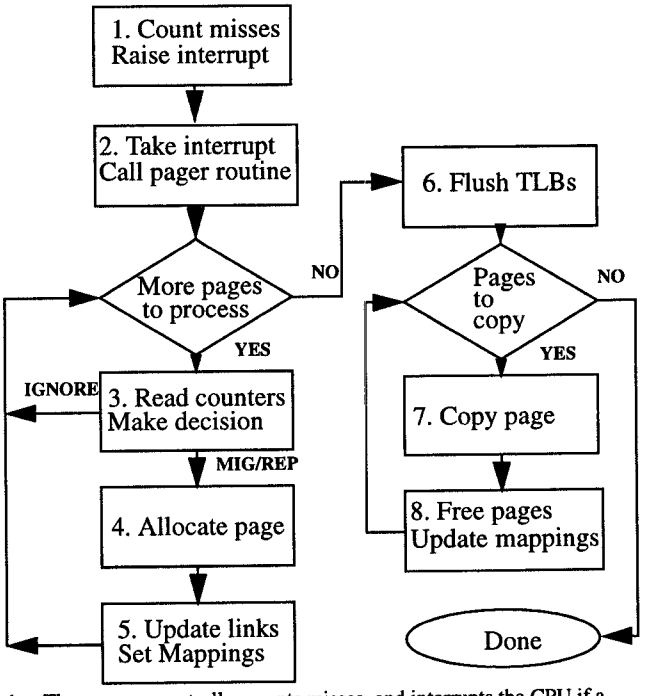
The first step (node 1) is to determine the pages to which a large number of misses are occurring, the "hot" pages. Because there is a page-movement cost, moving only hot pages maximizes performance. The second step (node 2) is to determine the type of sharing that applies to the page in question; the page should be migrated if referenced primarily by one process or replicated if referenced by many processes. On the basis of the sharing pattern, we take the replication (high sharing) or the migration (low sharing) branch. The third step (nodes 3a and 3b) is to help control the overhead cost. Replication is allowed only if the write frequency is low and there is no memory pressure. Migration is allowed only if the page has not been migrated too often in the past. This decision tree is similar to that used in earlier studies on non-cache-coherent NUMA machines, but ours is driven by all cache-misses, not all memory references.

4. Kernel Mechanisms

We modified the SGI IRIX 5.2 operating system to implement the page migration/replication policies. IRIX is an SMP OS that runs on the SGI CHALLENGE, a bus-based multi-processor system. Without going into the kernel modifications in detail, we point out some of the important issues in the implementation.

To implement the policy, we first need to translate the abstract decision tree developed in the previous section into concrete parameters that can be observed in the system. It is difficult to track rates, therefore we approximate them using counters with a periodic reset. For each page there is a miss counter per processor, a migrate counter, and a write counter. Table 1 defines a *reset interval* parameter and the *trigger*, *sharing*, *write*, and *migrate* thresholds that together with the counters approximate the rates in the decision tree.

A hot page is one whose counter for a processor reaches the *trigger threshold* within the reset interval. If that page is in a remote memory, further action is considered. Next, if the miss counter for this page on any other processor has exceeded the *sharing threshold*, we consider the page shared, and it is a candidate for replication; otherwise the page is not shared and is a candidate for migration. Finally, a candidate page is only replicated if the write counter has not exceeded the *write threshold*, and a candidate page is only migrated if the migration counter has not exceeded the *migrate threshold*.



1. The memory controller counts misses, and interrupts the CPU if a counter reaches the trigger threshold.
2. The processor takes the interrupt, and calls the pager routine that implements the page movement code.
3. The pager routine reads the counters for the page and its replicas, and decides to migrate, replicate, or collapse a page on the basis of the counter values and the sharing, migrate, and write thresholds. The necessary page-level locks are acquired.The decision taken can also be to do nothing, in which case the remaining steps are skipped.
4. A new page is allocated from the appropriate memory.
5. The new page is linked into all the appropriate data structures maintained by the OS. The page table entries are changed to reflect the transient nature of the page. The page itself is locked, so that the entries cannot be changed by a faulting processor.
6. The TLBs on all the processors are flushed. Flushing the TLB removes old TLB mappings that may not be valid after a migrate or a collapse. It also prevents writes to the page while the data is being copied.
7. The data is now copied to the new page.
8. Old pages are freed if necessary, and page table mappings are updated to point to the nearest page.

FIGURE 2: Flow chart for migration or replication of a page in the kernel implementation. A page collapse only goes through a subset of these steps and is not shown.

In our implementation, the directory controller maintains the miss counters, and generates an interrupt when a page crosses the trigger threshold. The flow-chart in Figure 2 describes the low-priority interrupt handler, which services this pager interrupt by replicating or migrating pages if necessary. To reduce the per-page overhead for taking the interrupt and flushing TLBs, the directory controller attempts to collect multiple pages before generating an interrupt. The interrupt handler code iterates over steps 3 to 5 for each page. Then a single TLB flush operation is done, followed by steps 7 and 8. There is one other path to the pager routine, the page collapse (not shown in Figure 2); it handles writes to replicated pages. The page table entries for replicated pages are marked read-

only, and a write to a replicated page causes a trap that vectors the processor to the protection fault handler (pfault). This handler collapses the replicas to a single page before letting the write proceed.

Three significant changes had to be made to the IRIX VM system to enable our implementation. The performance overhead of these changes, when not migrating or replicating pages, was insignificant (less than 0.5% of non-idle execution time).

Replication support: In IRIX, a hash table is used to translate logical pages (vnode, offset) to physical pages. Physical page frame descriptors (pfds) are linked into this open hash table. Support was added for replicas of a physical page. The replicas are linked together, and one of them is linked into the hash table.

Finer grain locking: The version of IRIX we used had fairly coarse locking for VM related structures. There is one lock (memlock), which protects the global hash table of active physical pages and the global free page list. There is also one lock per memory region (region lock), which is acquired when changing page table entries. Both memlock and the region locks for shared text or data regions were potential performance bottlenecks. To reduce the contention for memlock, we added page level locking for manipulating replica chains. A lock was also added to page table entries, to avoid having to acquire the region lock when changing the mapping. These changes reduced the synchronization cost in our implementation.

Page table back mappings: Page table entries (ptes) for virtual addresses point to pfds, but there is no link from the pfd directly back to the pte. To facilitate easy mapping changes, links were added to the pfd pointing back to all the ptes mapping this page, similar to an inverted page table.

5. Experimental Environment

In this study we model CC-NUMA and CC-NOW machines based on the Stanford FLASH and Distributed FLASH architectures, respectively. As these machines are not yet available, we use a machine simulator called SimOS [RHW+95]. SimOS is a complete and accurate simulator of the FLASH machine. It is capable of booting a commercial operating system, Silicon Graphics' IRIX 5.2 in this case, and executing any application that is binary compatible with IRIX. SimOS accurately models the processors, caches, memory system, and I/O devices (disks, ethernet, etc.) of the system, and includes a cycle accurate simulator of the MAGIC chip, the directory controller on FLASH. SimOS is an invaluable tool because it allows us to collect any number of statistics about the hardware or system activity, without altering the behavior of the workload. This non-intrusiveness, along with the complete and accurate modelling of all system activity, is the key advantage of SimOS over software-based instrumentation techniques on actual hardware.

In this study we model an 8 processor FLASH machine. The benefits of migration and replication should be seen even with this small configuration because the probability that a process would randomly find a page in local memory is already quite small (0.125). The following are the other machine characteristics assumed: 300MHz processors with a TLB size of 64 entries; separate 32KB two-way set-associative first-level I and D caches with a one cycle hit time; a unified 512KB two-way set-associative second-level cache with a 50ns hit time; a minimum local memory access time of 300ns and a minimum remote memory access time of 1200ns for the CC-NUMA configuration, and 3000ns for the CC-NOW configuration (we assume the 1000 ft. of fiber traversed in CC-NOW causes approximately 2000ns of latency).

Name	Contents	Notes
Engr.	6 Flashlite 6 Verilog	multiprogrammed, compute-intensive serial applications
Raytrace	Raytrace	parallel graphics application (rendering a scene)
Splash	Raytrace Volrendering Ocean	multiprogrammed, compute-intensive parallel applications
Database	Sybase	commercial database (decision support queries)
Pmake	4 four-way parallel Makes	software development (compilation of gnuchess)

TABLE 2: Description of the workloads. For each workload, the table lists the applications in the workload and a short description of the workload. All the workloads are run on an eight processor configuration, except the database workload that uses four.

6. Workload Characterization

The value of a study such as this depends critically on the workloads used. We use five diverse and realistic workloads to capture some of the major uses of compute servers. These workloads are summarized in Table 2 and detailed CPU and memory system statistics are given in Table 3.

Multiprogrammed Engineering Workload (Engineering): Our first workload consists of large sequential computation and memory intensive applications. This is a multiprogrammed workload, scheduled by UNIX priority scheduling with affinity [VaZ91]. The workload consists of copies of two applications. One is the commercial verilog simulator VCS, simulating a large VLSI circuit. VCS compiles the simulated circuit into code, and the resulting large code segment causes a high user instruction stall time. The other application is Flashlite, a functional simulator of the FLASH machine.

Single Parallel Application (Raytrace): This workload consists of Raytrace[SWG92], a single compute-intensive parallel graphics algorithm widely used for rendering images. The processes of the application are locked to individual processors, a common practice for dedicated-use workloads.

Multiprogrammed Scientific Workload (Splash): The third workload consists of parallel invocations of Raytrace, Volume rendering, and Ocean [SWG92]. The applications enter and leave the system at different times, and a space-partitioning approach, similar to scheduler-activations [ABL+91, TuG89], is used for scheduling the jobs.

Decision Support Database (Database): The fourth workload is a database system running a decision-support benchmark on an off-line main-memory database. We use the Sybase database that supports multiple engines, but has not been optimized for NUMA architectures. This workload is run on a four-processor system, with the database engines locked to processors.

Multiprogrammed Software Development (Pmake): Our final workload consists of four Pmake jobs, each compiling the gnuchess program with four-way parallelism. The workload is I/O intensive, with a lot of system activity from many small short-lived processes, such as compilers and linkers. UNIX priority scheduling with affinity is used. Kernel instruction and data references, rather than user references, account for the bulk of the

Workload	Cumulative CPU Time (sec)	Total Memory (Megs)	CPU Time Breakdown (%)			Stall Time (% Non-Idle)			
						Kernel		User	
			User	Kern	Idle	Instr.	Data	Instr.	Data
Engineering	61.76	27.5	74	6	20	1.6	3.8	34.4	37.4
Raytrace	74.08	28.8	69	25	6	3.6	15.1	4.8	36.1
Splash	87.52	57.6	65	17	18	4.4	11.8	3.1	36.3
Database	30.40	20.8	55	7	38	1.4	6.0	2.5	50.3
Pmake	35.27	73.7	34	44	22	4.0	29.3	3.6	9.1

TABLE 3: Execution time and memory usage of the workloads. The table shows the execution time of the workload, the total memory used, the percentage of the execution time spent in Idle, Kernel and User modes, the percentage of non-idle time spent stalled on the secondary cache for instructions and data.

memory stall time. Therefore, we use this workload in Section 8.2 to focus on the migration and replication potential in the kernel.

7. Workload Execution Results

This section explores the overall performance improvement, and then analyzes the different overheads of migrating and replicating user level pages. We study the engineering, raytrace, splash and database workloads in this section because of their large user stall times.

We varied each of the policy parameters for the migration and replication policy, and chose the ones that gave the best results; trigger threshold of 128 for the raytrace, splash, and database workloads and 96 for the engineering workload, sharing threshold is one quarter of the trigger threshold, the write and migrate thresholds are set at one, and the reset interval is 100 milliseconds. This policy is referred to as the *base policy*. Section 8.4 discusses the variation of policy parameters. We compare the results using the base policy against that of first-touch page allocation, the default policy on CC-NUMA machines.

7.1 Performance Improvement

We first examine in detail the workload execution time improvements from using the base policy. Next we show how improving data locality reduces contention for resources in the NUMA memory system, and so improves system-wide performance. Finally we examine the effects of longer network latency as in the CC-NOW configuration.

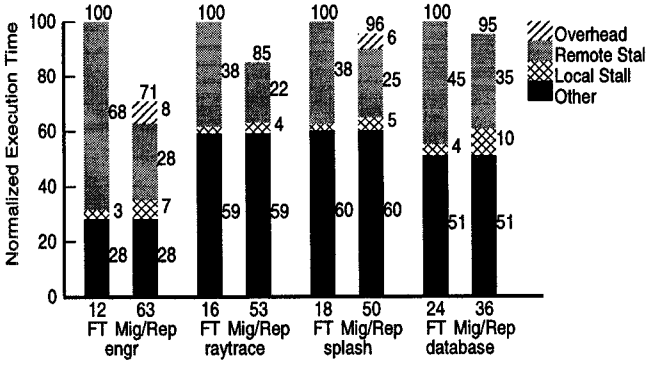


FIGURE 3: Performance improvement of the base policy (Mig/Rep) over first touch (FT). The execution time is divided into the kernel overhead for migration and replication, the remote and local stall times, and all other time. The figure shows the percentage of misses to local memory at the bottom of each bar.

7.1.1 Workload Performance

Figure 3 shows the results of our experimental runs. The improvement in total execution time is dependent on three factors: the contribution of user stall time to the total execution time, the fraction of misses satisfied from local memory, and the kernel overhead required to improve the memory locality. The base policy improved memory locality substantially, and this resulted in reductions in memory stall time — engineering 52%, raytrace 36%, splash 24%, and database 10%[1]. The total execution time improved in all cases, even after considering the additional kernel overhead — engineering 29%, raytrace 15%, splash 4%, and database 5%. We now discuss each workload.

Engineering: The engineering workload that has a large user stall time (72% of non-idle time), shows the largest performance improvement. For this workload, page migration improves the locality of the data segments that are not shared in sequential applications. Replication of code pages also has a large impact because of the large percentage of instruction cache stall time (34% of non-idle time). This workload demonstrates that both migration and replication are necessary to fully exploit memory locality.

Raytrace: In the raytrace workload, the processes make unstructured read-only accesses to a large shared data structure, which represents the scene to be rendered. Figure 4 illustrates the read-only nature of the data misses. This graph indicates the fraction of cache misses that could be potentially satisfied locally through replication. The fact that raytrace has 60% of its data misses in read-chains that are 512 misses or longer indicates most of the benefit comes from page replication.

Splash: In the splash workload there are three parallel applications, Raytrace, Volume Rendering, and Ocean. As the applications enter and leave the system, the jobs are redistributed across the processors to best utilize the CPUs. Therefore static placement of data is difficult. Ocean has only nearest-neighbor communication that results in mostly unshared access to data, therefore migrating pages to the processor currently running the process will improve locality. Raytrace and Volume Rendering

1. The FT database time has been compensated for user spin cycles.

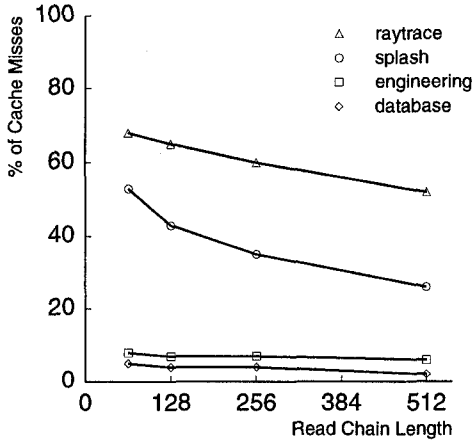


FIGURE 4: Percentage of data cache misses in read chains. A read chain represents a string of reads to a page from a processor, which is terminated by a write from any processor to that page. A long read chain indicates a page that could benefit from replication. The X-axis shows read chain lengths and the Y-axis shows the percentage of the total data misses that are part of read chains of that length or more.

have substantial read-mostly structures that could be replicated; 30% of the data misses in this workload are in read chains longer than 512. Therefore we would expect to see gains from replication also.

There are a couple of reasons the splash workload does not show more than a 4% improvement. First, as shown in Table 4, 24% of the time a hot page is identified, the kernel fails to take action because it cannot allocate a physical page frame on the local node. Although the total workload fits comfortably in the machine's memory, the memory on some of the nodes is exhausted. Second, when a process switches processors, it continues to use the page from the old node, even if there is a replica on the new node. Due to the current page table design, the process will not pick up the new replica until the page is identified as a hot page and processed by the kernel.

Database: We expected the database workload to benefit from replication because the data accesses would be mostly reads. We did not expect much benefit from migration because we locked the servers to the processors. However, we see very little additional benefit for replication over FT. Classifying the pages based on the type of access reveals that, of the 2.6 million user data misses, only about 10% are to read-mostly pages that could benefit from replication. The remaining 90% of the misses are concentrated in about 5% of the pages that have more writes than reads. These pages are apparently used for synchronization purposes, with fine

Work load	Hot Pages	% Migrate	% Replicate	% No Action	% No Page
Engr.	7,728	55	27	12	6
Ray.	2,934	34	31	35	0
Splash	6,328	36	22	18	24
DB	2,003	13	2	85	0

TABLE 4: Breakdown of actions taken on hot pages. For each workload the hot pages are broken down into the percentage of migrations, replications, instances no action was taken, and failures because no physical page was available on the local node.

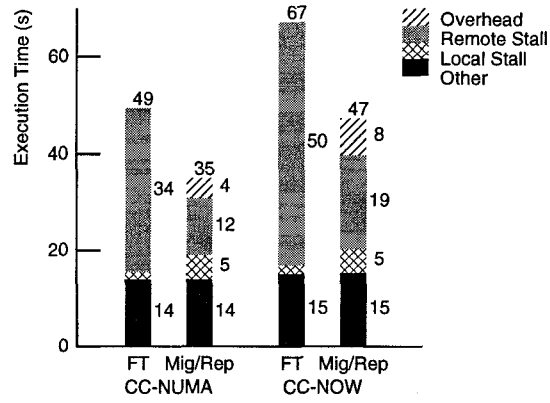


FIGURE 5: Performance comparison of the CC-NUMA and CC-NOW configurations for the engineering workload. For each of the two configurations, the non-idle execution time is shown for the first touch (FT) and base (Mig/Rep) policies.

grain sharing by all the processors. This sharing pattern was outlined in Section 3, and cannot benefit from replication or migration. Our policy is robust and correctly identifies this type of sharing; no action is taken on 85% of the pages, as shown in Table 4.

7.1.2 System-wide Benefit

In addition to reducing stall time for individual applications, the improvement of data locality has an additional system-wide benefit. A remote miss consumes resources on multiple nodes, and an excess of these remote misses can cause congestion in the interconnection network, increase occupancy in the directory processors, and increase queuing delays.

For example in the engineering workload, the base policy, by improving data locality, reduced the invocations of the remote memory request handler by 40%, the average network queue length for remote requests by 38%, and the maximum directory processor occupancy by 32%. Consequently, the average latency of a local read miss was reduced by 34%. To reduce hotspots in the NUMA memory system, we are considering modifying our policy to migrate even write-shared pages.

Contention for resources can contribute significantly to the latency of cache misses, both local and remote. Therefore with high contention, the actual latency of cache misses is much higher than the expected minimum. To demonstrate how improving data locality can reduce contention and improve performance, we ran the engineering workload on a setup with zero interconnection network delay. For this configuration, the base policy reduced memory-stall time by 38%, and improved overall execution time by 21%. This result also shows that improving data locality is important for CC-NUMA systems, even when the ratio of remote to local latency is smaller than the 4:1 assumed in this study.

7.1.3 Effect of Longer Network Latency

To study the effect of longer network latency on workload performance, we increased the network delay such that the minimum remote cache-miss latency was 3000µs. This represents a CC-NOW-like system. The engineering workload was run in this configuration, with and without migration and replication as in the CC-NUMA case. Figure 5 compares the performance of the CC-

Workload		Intr. Proc	Policy Decision	Page Alloc	Links & Mapping	TLB Flush	Page Copying	Policy End	Total Latency
Engineering	Repl.	13.0	12.6	184.3	28.6	35.9	87.0	80.5	441.9
	Migr.				75.8			63.4	472.0
Raytrace	Repl.	24.4	16.0	74.4	34.3	61.5	106.7	77.4	394.7
	Migr.				100.5			64.9	448.4
Splash	Repl.	22.2	12.8	170.6	40.2	51.3	97.1	91.9	486.1
	Migr.				99.7			62.4	516.1

TABLE 5: Latency of various functions in the policy implementation. The columns correspond to the implementation steps in section 4 Figure 2. The numbers are shown separately for Replication and Migration, where applicable. The latencies are in µs and are averaged across the entire run.

Workload	Kernel Ovhd (secs)	Percentage of Kernel Overhead							
		TLB Flush	Page Alloc	Page Copy	Page Fault	Links & Mapping	Policy End	Policy Decision	Intr. Proc.
Engineering	4.54	34.5	25.5	11.1	8.9	8.3	8.8	2.1	1.7
Raytrace	1.80	54.4	7.6	10.8	5.4	7.4	7.4	2.6	2.6
Splash	4.00	44.1	20.7	8.1	7.3	6.5	6.3	2.0	1.9

TABLE 6: Breakdown of total kernel overhead by function. We show the percentage of the kernel overhead for migration and replication that can be attributed to the various functions. In addition to the steps from Figure 2, we added a category for additional page faults, due to changes in mappings. TLB flushing and page allocation comprise a large part of the overhead.

NUMA and CC-NOW configurations. In the CC-NOW case, migration and replication reduces user memory-stall time by 53%, and improves overall performance for the workload by 30%.

Though the absolute reduction in stall time and improvement in performance is good, intuitively we expected larger gains based on the increase in remote latency. This did not happen for two reasons. First, the observed increase in remote cache-miss latency is only 60% (2279µs for CC-NUMA and 3680µs for CC-NOW), instead of the expected 150% (1200µs for CC-NUMA and 3000µs for CC-NOW). This difference is due to controller occupancy that adds significantly to the minimum remote latency. Second, the cost of a migrate or replicate increases to about 600µs, and the total kernel overhead to achieve the same data locality almost doubles. This reduces the overall performance improvement.

7.2 Overhead Analysis

As discussed in Section 3.1, it is important to reduce the overheads involved with migration and replication. These include the cost of gathering cache-miss information, the kernel overheads, the cost of data movement, and the indirect cost of increased memory use resulting from page replication. This section includes results for the engineering, raytrace, and splash workloads (the database workload moves relatively few pages).

7.2.1 Information Gathering Overhead

Cache miss collection involves both a time and a space overhead. The directory controller in MAGIC is a dedicated processor that runs software handlers to service cache misses. We added code to the relevant handlers to count the misses from each processor to each page of memory. The counting code is added to the end of the handler, and is only run after all necessary processing is done. The latency of the handler is unchanged, and only the run-time of the handler is increased. Further we use sampling, and count only one in ten invocations of the handler. In Section 8.3 we show that using sampled cache-misses is as effective as full cache-miss information. Our measurements show that the inserted cache-miss counting code has no measurable impact on the latency of cache misses or the application execution time.

Counting cache misses also incurs a space overhead because the policy requires one counter for each processor for each page. On the eight node system we evaluate, with 1 byte counters and 4K pages, this is a 0.2% overhead. On larger systems, for example with 128 nodes, this would be a 3.1% overhead. Sampling can also reduce this space overhead to 1.6% of memory by using half size counters. The overhead can be further reduced by logically grouping processors, and keeping a shared counter for the group. This overhead is to be contrasted with a 7% memory overhead for the per cache-line directory information that the controller already keeps to maintain cache-coherence in FLASH.

7.2.2 Kernel and Data Movement Overhead

We analyze the kernel overhead in two ways. First, we look at the latency of a page migration or replication operation (Table 5). The latency is the end-to-end time as seen by the interrupt handler when performing the operation. Second, we look at a more global view of the costs – the increase in system activity (kernel time) because of migration and replication (Table 6).

The total latency per operation ranges between 400 - 500 µs. The interrupt processing and TLB flushing costs are amortized across multiple pages. Effective values for a single page are shown. The engineering workload had on average more pages per interrupt, so its effective latency in these categories is lower. During migration, memlock is acquired to remove the old page from the physical page hash table, and to insert the new page. In contrast, for replication only a page-level lock is acquired because the replicas

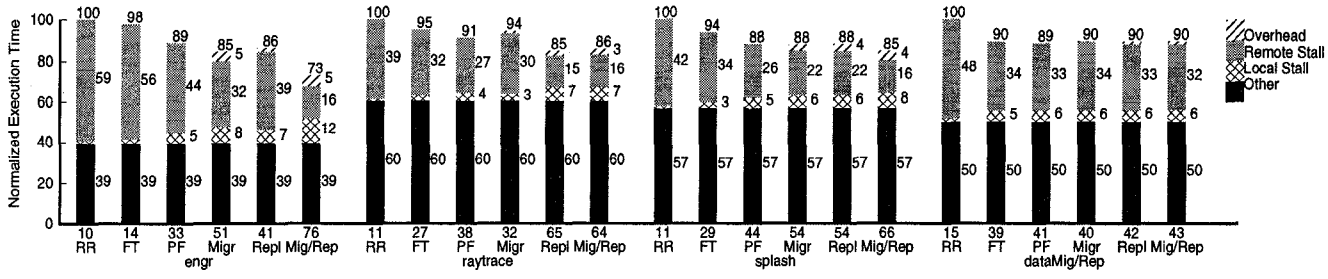


FIGURE 6: Breakdown of user execution time for various policies. There are six runs for each workload, Roundrobin (RR), First touch (FT), and Post-facto (PF), Migration-only (Migr), Replication-only (Repl), and the combined migration/replication policy (Mig/Rep). Each bar represents the execution time for a policy normalized w.r.t. the RR policy. Each bar shows cache-miss stall to local memory, cache-miss stall to remote memory, the overhead to migrate and replicate pages, and all other time. The percentage of misses to local memory is shown at the bottom of each bar.

are queued only on the master page. Consequently, the latency for step 5 (Links & Mapping) is larger for migration. In step 8 (Policy End), replication takes longer than migration because all current mappings to the page have to be set to point to the closest replica. Engineering and splash allocate more pages than raytrace, and the greater resulting contention for memlock leads to larger latencies.

We reduced the latency for the TLB flush by streamlining the communication between the requesting CPU and the others, and by collecting multiple pages before flushing the TLBs. Surprisingly however, Table 6 shows that TLB flushing is still the leading kernel overhead. This is because all processors on the system must flush their TLBs, as there is no information in the kernel about which processors currently hold mappings for the pages in question. The time spent flushing TLBs can be reduced by tracking the processors that have mappings for a page, and flushing only those TLBs. We simulated this capability, and found that the total kernel overhead decreased by approximately 25%; on average, each TLB flush only required two TLBs to be flushed instead of the current eight. However, this functionality is not supported by the current OS, and would require significant changes to the VM system.

Page allocation is the second leading cause of kernel overhead, primarily because of the time spent in contention for memlock. We reduced the contention for memlock in our implementation through page-level locks. Redesigning the VM system to remove the memlock bottleneck entirely should reduce the time spent in page allocation and some of the other routines, leading to significantly lower latency and overhead.

We had expected the actual copying of bytes to be the leading cause of overhead, but we find that it represents only about 10% of the observed kernel overhead. An unoptimized bcopy done by the processor takes approximately 100µs. A pipelined memory to memory copy done by the directory controller in FLASH takes about 35µs, and using this feature could reduce the copying overhead further.

7.2.3 Replication Space Overhead

There is a memory space overhead associated with page replication because of the additional copies. Our migration/replication policy attempts to keep this space overhead down by selecting only hot pages for replication. Another possible policy is to replicate code pages on first-touch. In the engineering workload, replication on first-touch could potentially result in a 500% increase in memory usage for code pages because there are six instances of each application. By selecting only hot pages, the base policy increased memory usage by only 32%. Similarly in raytrace, for code and data, the increase in memory usage was only 20%. Additionally, the kernel implementation was designed to respond to memory pressure by stopping replication and preferentially reclaiming replicated pages.

Summarizing this section, we have shown that page migration and replication can reduce the overall execution time of a workload. We achieve large reductions in memory stall time by increasing the percentage of total cache misses serviced from local memory. However the kernel overheads, such as TLB flushing and lock contention, can be rather large. A more efficient implementation would allow for a more aggressive policy, and increase the benefits seen.

8. Exploration of Policies and Parameters

Section 7 analyzed the costs and benefits of page replication and migration on a set of workloads. In this section we will explore a few interesting issues related to alternative policies and parameters: effectiveness of alternative policies; migration and replication for the kernel; effectiveness of alternative metrics; sensitivity of the base policy to changes in the trigger and sharing thresholds.

We non-intrusively generated a detailed trace for each workload using SimOS. The trace contains information about all secondary cache misses, both user and kernel, and TLB misses, including the processor taking the miss, and a timestamp. The trace was then used as input to a policy simulator with a simple contentionless memory model. The memory model has a 300ns local-miss latency and a 1200ns remote-miss latency. The cost of a migrate, replicate, or collapse is 350µs. We use the traces for this part of the study because we are only comparing performance, and are not interested in absolute values. Using traces allows us to explore a wide range of issues in a controlled manner.

8.1 Alternative Policies

In this section we show how the combined migration/replication policy performs against other policies. We consider six strategies, three static and three dynamic. The static allocation strategies are: roundrobin (**RR**) that is equivalent to random allocation; first touch (**FT**), where the page is allocated to the processor that first touches it; and post-facto (**PF**) that is the best possible static allocation case and assumes future knowledge. The dynamic strategies are: migration only (**Migr**), replication only (**Repl**), and the combined replication/migration policy (**Mig/Rep**). The parameters for the dynamic policies are similar to the base policy in Section 7; trigger threshold 128, reset interval 100ms, write threshold 1, migrate threshold 1, and sharing threshold 32.

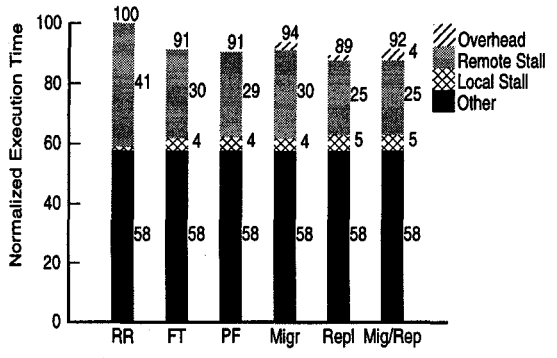


FIGURE 7: Execution time breakdown for the pmake workload. Only Kernel misses are considered for the different policies.

Figure 6 shows the result of these simulations for the four workloads. Overall, for three of the four workloads shown, policies doing migration or replication or both outperform the static policies, even the post-facto (PF) policy that assumes perfect future knowledge. The reduction in execution time is significant for the Mig/Rep policy even after modelling the costs of page movement.

8.2 Migration and Replication in the Kernel

The operating system can be considered as a large parallel program with shared code and data, and for some workloads, like pmake, a significant fraction of CPU and memory stall time is spent in the kernel. IRIX, like most UNIX kernels today, is loaded in memory at boot time, and its code and data are not mapped through the TLB. Therefore, we cannot actually migrate and replicate kernel pages, but we use the traces for the pmake workload to study if the kernel can benefit from migration and replication.

Figure 7 shows the effect of applying the various policies, using our policy simulation, on the traces of kernel activity. There is almost no benefit beyond first touch, and the little that is observed with the Repl and base policies is from the replication of kernel code that accounts for only about 12% of the misses. Kernel data shows no real benefit beyond that of the static FT policy. We analyzed the kernel miss trace at the granularity of cache lines and words, to see if restructuring the data layout might produce more read-mostly pages. We found very little potential for replication. The FT policy gives some benefit over RR because there are some kernel structures that are per-processor or have a natural affinity to a particular processor, e.g. the Private Data Area (PDA), the kernel stacks, and the Page Frame Descriptors (PFD) for memory local to a processor. Page migration could produce a small improvement for structures that are per process, e.g. user page tables. The migration benefit for these structures is similar to that for unshared user data, and is dependent on the scheduling of the associated user process.

8.3 Effectiveness of Alternative Metrics

In the experimental runs in section 7, the policies were based on full cache-miss information from the directory controller. Full cache-miss information is difficult to obtain on many machines. We consider two alternatives to full cache-miss information, sampling and TLB misses. Cache-miss sampling is becoming available on next-generation machines, and is a realistic alternative to full cache-miss information. Also, on systems with software reloaded TLBs, TLB misses can be collected directly by the OS. The miss behavior of the TLB can be modelled as a cache with the line size being a page. Therefore the validity of this approximation would depend on the access patterns of the application and the size and architecture of both the cache and the TLB. To evaluate the effect of approximate information we studied four metrics: full cache (FC), sampled cache (SC), full TLB (FT), and sampled TLB (ST). For the sampled metrics, we use a 1 in 10 sampling rate.

Figure 8 shows the results for the policies with approximate information. The important question when using partial or approximate information is how faithful is the heuristic to the original for the purpose at hand. The performance of the policy when using sampled cache-miss information is identical to that using full cache-miss information for all the workloads. Clearly sampled cache-miss information is an effective approximation for full cache-miss information. Using the sampling of cache-misses in our instrumentation of the software handlers in MAGIC, we are able to almost completely eliminate any information gathering overhead. Using TLB misses is effective in some workloads, but clearly not so in the engineering workload. The use of TLB misses to drive migration and replication policy needs to be studied further.

8.4 Variation in Policy Parameters

Trigger threshold: The trigger threshold controls the aggressiveness of the policy, and introduces a trade-off between increased memory locality resulting in reduced stall time and the kernel overhead to migrate and replicate pages. Figure 9 illustrates this trade-off clearly for the different workloads. The best "operating point" would depend on the actual values of the local and remote miss latencies, the overhead to migrate or replicate a page, and the percentage of execution time spent in memory stall

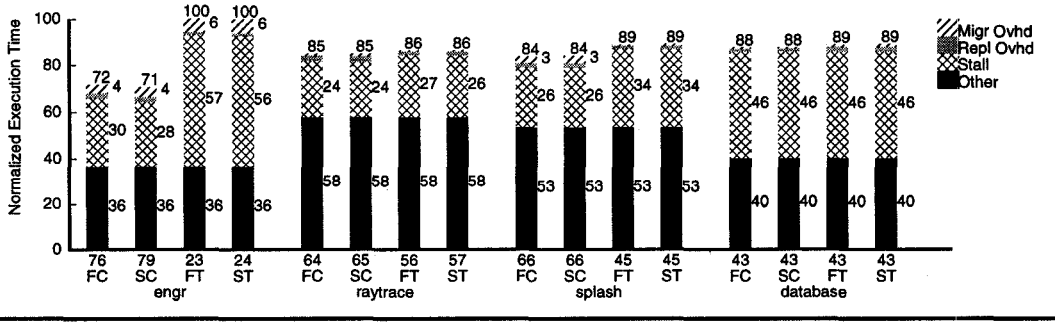


FIGURE 8: Performance impact of approximate information. There are 4 bars for each workload, Full cache (FC), Sampled cache (SC), Full TLB (FT), and Sampled TLB (ST). All sampled cases are sampled with ratio 1:10. Each bar shows the run time normalized to the run time for the Round robin case. The run time is broken down as User stall (local and remote), Overhead for replication and migration separately, and all other time. The percentage of misses made local is shown at the bottom of each bar.

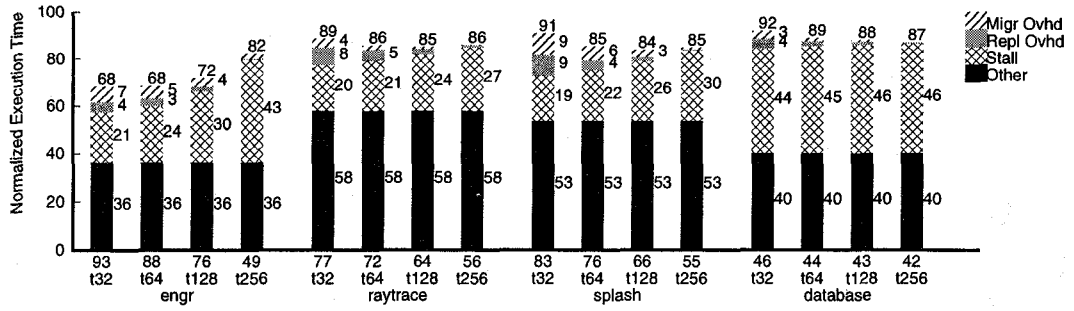


FIGURE 9: Variation in performance with the trigger threshold. Each workload is run with four trigger thresholds, 32, 64, 128, and 256. The sharing threshold is a quarter of the trigger threshold. Each bar shows the run time for a configuration normalized to the run time for the roundrobin case for that application. In each bar, we separately show User CPU, User stall, and overhead cost of replication and migration. The percentage of misses made local is shown at the bottom of each bar.

for the workload. Reducing the kernel overhead, would enable larger improvements in performance through the use of a smaller trigger threshold. The trigger threshold is a critical parameter and selecting the correct trigger value, statically or adaptively, is a topic for further study.

Sharing threshold: The sharing threshold is used to differentiate between shared and unshared pages, and so decides whether a hot page is potentially migrated or replicated. A higher sharing threshold favors migration over replication. Our tests show that the performance is quite insensitive to the value of the sharing threshold within a reasonable range. This fact indicates that most pages are clearly differentiated, being either shared (code and shared data of parallel applications) or unshared (data of sequential applications), and very few pages in the workloads have an ambiguous sharing behavior.

9. Conclusions

Cache coherent distributed shared memory multiprocessors are becoming increasingly popular as compute servers. The CC-NUMA architecture will be used for even moderate-sized machines because of the increasing speed of next generation processors and the scalability of this architecture. The key factor that affects the performance of applications on these systems is memory locality that is the focus of this paper. We assembled a realistic set of workloads that included single and multiprogrammed parallel applications, engineering simulators, software development tools, and a database server. The results from running the workloads on a kernel implementation of our policy showed that migration and replication of pages improved memory locality and reduced the overall stall time by as much as 52%. Additionally, this significantly reduced the contention for resources in the NUMA memory system.

A detailed analysis of our kernel-based implementation of the policy showed that the primary sources of overhead were processor synchronization and TLB flushing. We investigated the use of other forms of information to drive the policy and found that TLB misses were an inconsistent approximation for cache misses. Our studies also showed that using cache-miss information sampled at a rate of 1 in 10 can give results matching those of full cache information without causing any appreciable overhead due to information collection. To support page migration and replication, future machines should include the ability to collect sampled cache-miss information.

Considering both the benefits and the costs involved, page migration and replication reduced workload execution time as much as 29% on CC-NUMA, and could potentially do better on CC-NOW. For applications with access patterns that cannot benefit from replication or migration of pages, our algorithm is robust and does not degrade performance. We believe that a fully optimized migration/replication implementation will allow for a more aggressive policy leading to even larger gains. In workloads where memory stall time is a problem, page migration and replication is an effective solution, and is therefore necessary on CC-NUMA machines to maximize individual application and overall system performance.

Acknowledgments

We are grateful to the SimOS development team for providing our experimental platform. We would also like to thank the FLASH hardware team for their help with Flashlite, and for providing the pieces of the engineering workload.

This work was supported in part by ARPA contract DABT63-94-C-0054. Mendel Rosenblum is partially supported by a National Science Foundation Young Investigator award.

References

[ABL+91] 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. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95-109, October 1991.

[ACD+91] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. *MIT/LCS Memo TM-454, Massachusetts Institute of Technology*, 1991.

[BCZ90] J. K. Bennett, J. B. Carter, W. Zwaeneopoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second Symposium on Principles and Practice of Parallel Programming*, pages 168-175, March 1990.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 1993 IEEE CompCon Conference*, pages 528-537, February 1993.

[BGW89] D. Black, A. Gupta, and W. D. Weber. Competitive management of distributed shared memory. In

Proceedings of COMPCON, pages 184-190, March 1989.

[BSF+91] W. Bolosky, M. Scott, R. Fitzgerald, and A. Cox. NUMA policies and their relationship to memory architecture. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 212-221, April 1991.

[CDV+94] R. Chandra, S Devine, B Verghese, A Gupta, and Mendel Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, 12-24, October 1994.

[CoF89] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32-43, December 1989.

[Hol89] M Holliday. Reference history, page size, and migration daemons in local/remote architectures. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 104-112, April 1989.

[Kus+94] J. Kuskin, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[LEK91] 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 System Principles*, pages 137-151, October 1991.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessey. 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.

[Li88] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 125-132, August 1988.

[LoC96] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308-317, May 1996.

[NAB+95] A. Nowatzyk et al. The S3.mp Scalable Memory Multiprocessor. Proceedings of the 24th International Conference on Parallel Processing, Aug. 1995

[RHW+95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: the SimOS approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[RSL92] M. Rinard, D. Scales, M. Lam. Heterogeneous parallel programming in Jade. In *Proceedings of Supercomputing '92*, pages 245-56.

[ScL94] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings, Operating Systems Design and Implementation*, pages101-114, November 1994.

[SWG92] J.P. Singh, W. Weber, A. Gupta. Splash: Stanford Parallel Applications for Shared Memory. Computer Architecture News, 20(1):5-44, 1992.

[TuG91] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In P*roceedings of the Twelfth ACM Symposium on Operating Systems Principles,* pages 159-166, December 1991.

[VaZ91] R. Vaswani and J Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared-memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 26-40, October 1991.