OS Support for Improving Data Locality on CC-NUMA Compute Servers

Ben Verghese
Scott Devine
Anoop Gupta
Mendel Rosenblum

Technical Report: CSL-TR-96-688

February 1996

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. Anoop Gupta is partially supported by a National Science Foundation Presidential Young Investigator award.

OS Support for Improving Data Locality on CC-NUMA Compute Servers

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum

Technical Report: CSL-TR-96-688

February 1996

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-2140
{pubs}@cs.Stanford.EDU

Abstract

The dominant architecture for the next generation of cache-coherent 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 - 5 times the latency to local memory. Given the large remote access latencies, data locality is potentially the most important performance issue. In compute-server workloads, when moving processes between nodes for load balancing, to maintain data locality the OS needs to do page-migration and page-replication. Through trace-analysis and actual runs of realistic workloads, we study the potential improvements in performance provided by OS supported dynamic migration and replication. Analyzing our kernel-based implementation of the policy, we provide a detailed breakdown of the costs and point out the functions using the most time. We study alternatives to using full-cache miss information to drive the policy, and show that sampling of cache misses can be used to reduce cost without compromising performance, and that TLB misses are inconsistent as an approximation for cache misses. Finally, our workload runs show that OS supported dynamic page-migration and page-replication can substantially increase performance, as much as 29%, in some workloads.

Key Words and Phrases: data locality, CC-NUMA, virtual memory, page migration, page replication

Copyright © 1996
Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum

1.0 Introduction

Shared-memory multiprocessors today are primarily bus-based machines with uniform memory access time (UMA machines). However, as multiprocessors get larger and the individual processors get faster, the memory bus quickly becomes a bottleneck. With superscalar processors, like the DEC Alpha 21164 that run at 300MHz, the bus is becoming a bottleneck even for small systems. The architectural solution to this problem is to distribute the main memory with individual processors, use directory techniques to provide cache coherence, and to use scalable interconnect technology. Cache-coherent shared-memory systems come in two variations. The first is the more traditional CC-NUMA machine (cache-coherent non-uniform-memory-access), e.g. Stanford DASH [LLG+90], MIT Alewife [ACD+91], and Convex Exemplar, which use specialized custom interconnect. The second is the CC-NOW machine (cache-coherent networks of workstations), that use more general-purpose interconnect technology with the nodes being users' workstations, e.g. the Stanford Distributed FLASH proposal [Kus+94] and the SUN s3.mp architecture.

Both CC-NUMA and CC-NOW machines will provide transparent access to local and remote memory, making them attractive as compute servers. The access to local memory is optimized to provide workstation-like latencies. The remote access latency is expected to be about 3-5 times the local access latency for CC-NUMA and about 10-20 times the local case for CC-NOW. The impact on application performance could be quite drastic if a large fraction of cache misses have to be serviced from remote memory, consequently data locality is potentially the most important performance issue. Traditional techniques to enhance locality, e.g. locking processes to specific processors and allocating memory locally, conflict with the need to move processes around for load balancing in compute-server workloads (affinity scheduling [VaZ91] is a compromise and is important). Page migration can move data to the local memory of the processor running the process. Replication of data in different memories, can also improve memory locality, potentially even in parallel applications where careful static placement of data is usually done. The focus of this report is to study OS supported migration and replication of pages, to enhance memory locality in such systems.

The rest of this report is organized as follows. First we briefly discuss related work. In Section 3, we present a framework, to help reason about various design issues and policy choices for page migration and replication. We describe our experimental environment in Section 4. We base this study on several classes of workloads that are important for multiprocessor compute servers. A multiprogrammed engineering workload, a program-development workload, dedicated and multiprogrammed parallel scientific applications, and a parallel Sybase database server running decision support queries. In Section 5, we provide a description and a base characterization of these workloads. In Section 6, using traces of complete cache miss information, we study the implications of the myriad of policy choices available, and show that there are significant potential gains from migration and replication. Section 7 explores the impact of making page movement decisions based on approximate information, such as TLB misses and sampling. Section 8 discusses the issues in our kernel implementation of the policy and gives a detail breakdown of the costs. We show that the performance of the workloads improves significantly when run on our kernel. Finally, we present our conclusions in Section 9.

2.0 Related Work

Related work on this subject has focussed on three areas. The first is implementing shared memory on message-passing machines and networks of workstations (involving replication and migration of pages/objects). The second focuses on migration/replication on shared-memory machines that are *non-cache-coherent*. The target machine architecture of our work, CC-NUMA, is significantly different from that assumed in these two areas of related work. The third area includes limited work done on CC-NUMA machines, the majority of which has focussed only on migration; replication has not been explored.

Work in first category includes that by Li on IVY [Li88], by Zwaenopoel et al. on Munin and Treadmarks [BCZ90], by Bershad et al. on Midway [BZS93], and by Lam et al. on Jade and SAM [ScL94]. These systems provide a shared-memory programming abstraction on multicomputers or networks of workstations where the underlying hardware does not support direct execution of remote load/store instructions. Consequently, the runtime system or the operating system is used to migrate and/or replicate pages or shared-objects in the local memory of the individual nodes. Given the underlying architecture of these systems, data replication or migration will be *required* when a processor references a remote datum. Consequently, the focus has been to minimize overheads (which could be from tens of microseconds to milliseconds) through techniques to couple data and synchronization and by exploiting task knowledge to do prefetching.

Work in the second category includes that by LaRowe and Ellis [LaE91], Fowler and Cox [CoF89], Holliday [Hol89], and Bolosky et al. [BSF+91]. These groups focused on migration and replication on *non-cache-coherent* shared-memory NUMA machines like the BBN-Butterfly and the IBM ACE machine. The policies to migrate/replicate were triggered by page-faults. Appropriate mechanisms (e.g., freezing and defrosting of pages) were developed to prevent excessive migration/replication and to keep overhead low. Although many of the mechanisms that were developed are directly relevant to us for this study, their policy and performance results are less applicable because of our different hardware assumptions.

Our environment differs from the previous two in the following respects. The ability of CC-NUMA machines to cache remote shared data substantially changes the potential benefits of migrating/replicating pages and requires us to be much less aggressive (the page movement overheads we can tolerate are much lower). In our environment, a remote access latency is on the order of a microsecond, but with shared memory and cache-coherence, subsequent accesses hit in the local processor cache and so take only a few nanoseconds. This changes the trade-offs substantially; for example, the performance can be quite good even without any migration/replication if the cache locality of the workload is good. The workloads that we evaluate are also substantially different from those used in these earlier studies.

Work in the third category includes work by Black et al [BGW89], and Chandra et al [CDV+94]. Black et al. proposed a competitive strategy to migrate and replicate pages in a NUMA multiprocessor using special hardware support (a counter for each page-frame and processor combination). The workloads evaluated included a few small scientific applications from the SPLASH suite [SWG92] (run one at a time), and only user-mode data references were considered. Our workloads are very different and include complex multiprogrammed

applications, and include both kernel and user references. The policy space explored is also quite different. Finally, Chandra et al, reported results for scheduling and page migration studies based on the Stanford DASH multiprocessor. This work while being closely related is different in many important respects: (i) focus of Chandra et al's work was process scheduling and migration, while here we study memory locality through both migration *and* replication; (ii) the workloads in this study are more varied, including Sybase and large engineering applications; and (iii) this study is based on the SimOS simulation environment [HWR+95], while Chandra et al's work was directly implemented on DASH. While using DASH certainly has the advantage of being more real, the SimOS environment used here is realistic (see Section 4) and allows us to vary/choose architecture parameters more freely (e.g., we can choose processor speeds and cache sizes more relevant to today's systems, than when DASH was designed 5-6 years ago).

3.0 The Problem Statement and Solution Framework

In this section, we begin with a more concrete description of the problem we are trying to address. We then present an abstract migration/replication algorithm in the form of a decision tree. Finally, based on the decision tree, we present the key migration/replication policy parameters that we explore in this report.

3.1 The Problem Statement

Our goal, as that for most other optimization techniques, is to minimize the runtime for the user's workload. More concretely, this translates into the statement: *perform page migration and replication so as to maximize the following quantity — (cost savings from converting remote misses to local misses less the cost overheads due to migration and replication).* As is evident from the above statement, we need to be concerned about both components of the above statement.

The first component involves finding the pages suffering the most remote-misses and converting these remote-misses to local-misses using migration or replication. Whether migration should be used, or replication should be used, or whether neither will help is primarily dependent on the access patterns to the page. Based on their access patterns, pages can be broadly classified in three groups:

1. *Pages that are primarily accessed by a single process*: This group includes data pages of sequential applications, and data pages of parallel applications where the accesses from the multiple processes are to disjoint sections of the data. This group also includes the code of sequential applications, if there is only one instance of the application running on the machine. These pages are prime candidates for migration when the process accessing them migrates to another processor, usually for load balancing reasons.
2. *Pages accessed by multiple processes, but with mostly read accesses:* This group includes code pages of parallel applications, and of sequential applications with multiple copies running. It also includes read-only or read-mostly data pages of parallel applications. These pages are prime candidates for replication.

3. *Pages accessed by multiple processes, but with both read and write accesses:* This group includes the data pages of parallel applications where there is fine grain sharing with updates from multiple processors. Replication is never a good idea for such pages; migration does not help either, except in special cases where cache misses from one processor dominate over misses from the other processors.

The second component we need to consider is the cost of supporting migration and replication. We can classify costs into four categories:

1. *Information gathering cost*s to help determine when and what pages to migrate or replicate. We will discuss this topic in detail in Section 7; the options we consider are to keep track of all cache misses or to use time sampling of cache misses (supported by Stanford FLASH). Many processors have software handling of TLB misses, so we also consider using TLB misses or time sampled version of TLB misses.
2. *Data movement cost*s to physically migrate or replicate the page. Machines that we are considering here will have interconnection network bandwidths greater than 100 Mbytes/sec, so a memory-to-memory transfer of a 4KB page will likely take less than 40 microseconds.
3. *Kernel overhead costs* for migration and replication. These include time for allocating a new page, removing all mappings to the old physical page, establishing the new mappings, flushing TLBs to maintain coherence and eliminating/updating other replicas on a store to a replicated page, and so on. These costs will vary with the organization of the VM subsystem; most current kernels are not very supportive of changing mappings cheaply (we will expand on these costs in Section 8).
4. *Memory-pressure related costs* are indirect costs that occur due to the increased memory use with page replication.

Given the above set of costs and potential for benefits, the challenge is to design an algorithm that will maximize the net benefits. The potential for benefit depends on three factors:

1. What fraction of the execution time is spent stalled on cache misses that go to memory and are these misses to pages with access types 1 and 2 described above? The larger the fraction of execution time spent stalled, the greater the potential benefit from improving memory locality.
2. How effective is the policy in finding pages with remote-misses and migrating or replicating these pages to make the misses local.?
3. How large is the overhead in moving these pages? The larger the overhead, the more selective the policy has to be, only finding and moving pages with a large number of misses.

3.2 Solution Framework and Policy Parameters

We now outline, in the form of a decision tree (see Figure 3.1), the sequence of decisions that need to be made to decide when and what pages to migrate or replicate. For now, we assume that action may be triggered on any cache miss. Although, it may be possible to approximate cache misses by UTLB misses to reduce overhead (see Section 7), the nature of the decision tree remains the same.

Given that there is substantial cost associated with page migration/replication, our first task is to determine the hot pages, that is the pages to which a large number of misses are occurring. If the

number of misses to a page is small, then end performance benefits will be small, and we need not concern ourselves with such pages (especially given that remote reference in CC-NUMA are not too expensive and shared data may be cached). If the page is hot, our second task (node 2 in Figure 3.1) is to determine the type of sharing that applies to the page in question. This is important as it will help decide if the page should be migrated (if referenced primarily by one process) or replicated (if referenced by many processes). Based on the sharing pattern, we take the replication (high sharing) or the migration (low sharing) branch. The final two nodes (numbered 3 and 4) help reduce the overhead cost under various circumstances. Replication is allowed only if the write frequency is low *and* there is no memory pressure. Migration is done only if the page has not been pinging back and forth between different nodes. We note that this decision tree is very similar to that used in earlier studies on non-cache-coherent NUMA machines, the key difference being that we are driven by cache-misses instead of all memory references.

The abstract decision tree can now be translated into concrete parameters than can be observed in the system and thresholds that can be used to make policy decisions. The framework we develop here will be used later in Sections 6, 7, and 8 for analyzing performance benefits. We also note that for now we remain with cache-miss based parameters; later we will discuss sampled-cache-miss based and UTLB-based parameters. Finally, since it is difficult to track rates, all rates are computed using counters with a periodic zeroing of counters.

To approximate miss rate, we keep a miss counter per page per processor, and define a *trigger threshold* and a *reset interval*. A counter for a page hitting the trigger threshold makes the page hot (i.e., migration/replication action is considered for that page) if the page is not on that processor. With a periodicity of the reset interval all the counters for a page are reset. To determine sharing behavior, we define a *hold threshold*. It is used as follows. When a page is triggered (i.e., some processor exceeds its trigger threshold), we examine the miss counters for

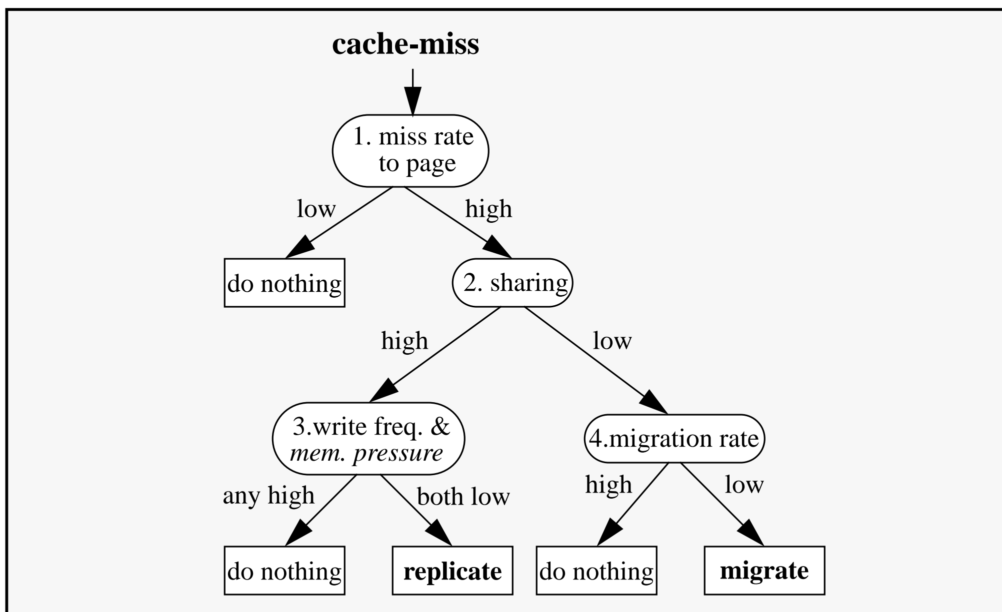

FIGURE 3.1. Replication/Migration Decision Tree. The flowchart shows the steps involved in making a decision about a page when it takes a cache miss. The possibilities are replicate, migrate or do nothing.

other processors that have a copy of that page (replica or original). If any of them is above the hold threshold, we consider the page multiply shared and consider it a candidate for replication; if hold threshold is not exceeded by any other processor, the page is considered only for migration. (The hold threshold is always less than the trigger threshold.) Finally, the write frequency and migration rate are tracked by keeping a counter per page for the number of writes and number of migrates. These counters are also reset periodically, every reset interval. To determine if too many writes and/or migrates are happening to a page, we define a *write threshold* and a *migrate threshold*. If number of writes to a page within a reset interval exceeds write threshold, the page is not considered for replication until the end of the reset interval, and if number of migrates exceeds its threshold, the page is not considered for migration until the end of the reset interval. These parameters are summarized in Table 1. We now discuss the experimental environment and workloads we use for our evaluation studies.

Table 1 Key parameters used by OS policies to decide migration and replication. The counters used by the policies include a per-page per-processor miss counter, and a single per-page write counter and migrate counter.

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.
Hold Threshold	Number of misses from another processor with a local copy, making a page a candidate for replication.
Write Threshold	Number of writes after which a page is stopped from consideration for replication
Migrate Threshold	Number of allowable migrates before page is stopped from consideration for migration.

4.0 Experimental Environment

The studies in this report are based on running actual applications on a simulated machine called SimOS[RHW+95]. The analysis is done through tracing of cache and TLB misses and statistics collection during the run. To do the above non-intrusively and since our target machine has characteristics not currently available we use SimOS, a machine simulator. SimOS is a complete and accurate machine simulator capable of booting a commercial Operating System (Silicon

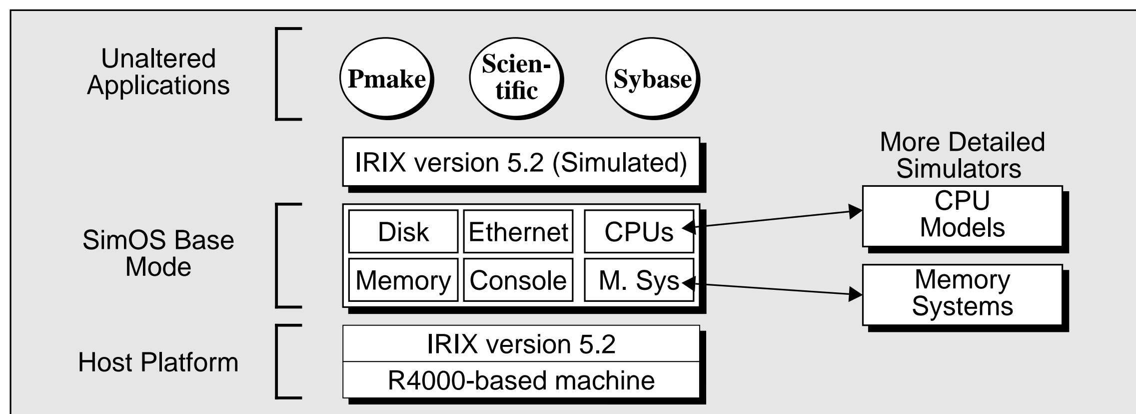

FIGURE 4.1. The SimOS environment runs as a layer between the simulation host platform and unaltered workload applications. Additionally, SimOS provides the flexibility to include a variety of more detailed CPU and memory system simulators.

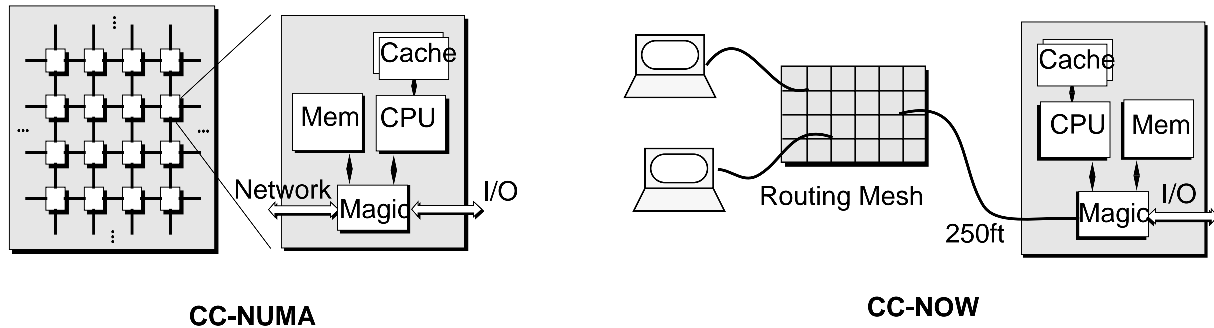

FIGURE 4.2. CC-NUMA and CC-NOW multiprocessors explored in this report. For CC-NUMA, we assume local memory access latency of 300ns and remote memory latency of 1200ns (ratio of 1:4). For CC-NOW we assume local memory access latency of 300ns and remote memory latency of 3000ns (ratio of 1:10).

Graphics' IRIX 5.2 in our case) and executing unmodified applications (i.e., *unmodified* binaries for original SGI systems), including all of the system activity that they require. SimOS models various processors, I/O devices (disks, ethernet, etc.), processor caches, and memory systems.

Because the notion of simulated machine time is completely internal to the simulator, any number of statistics can be recorded in essentially zero time without altering the behavior of the workload. This non-intrusiveness is one of the key advantages of simulation over software-based instrumentation techniques on actual hardware. The simulator not only collects data and statistics on the hardware components of the simulated machine, but also provides non-intrusive tracing of machine activity. This allows the collection of cache and TLB information without perturbing the system. In our studies we use SimOS in two modes:

- Generating traces of cache and TLB misses for each workload. These traces are used to do a wide range of policy explorations in Sections 6 and 7 of the report.
- Actually running the various workloads for a limited set of policies with a kernel that implements page migration and replication. These are studied in Section 8.

The target CC-NUMA and CC-NOW machines that we model using SimOS are based roughly on the Stanford FLASH and Distributed FLASH architectures (see Figure 4.2). The actual hardware parameters that we use in our simulations are as follows. We assume an 8-node machine for all our explorations. Although 8-nodes are quite few for CC-NUMA organization, our reasons were: (i) some of our workloads (e.g., the database) do not yet scale to large numbers of processors; (ii) larger number of processors would have meant larger sized workloads, and the simulation times would have become unacceptable (some of them already take over 24 hours to run on a 133MHz SGI Indy); and finally (iii) even with 8 nodes, the probability that a process would by accident find a randomly allocated page local is already quite small (12.5%), so most of the benefits of migration/replication can be seen with these few nodes. We assume that each node is a 300MHz processor with TLB size of 64 entries with 8 reserved for the kernel and wired entries. We assume separate level-1 I and D caches, each 32KB 2-way set associative with 1 cycle hit time. We assume a unified I and D level-2 cache that is 512KB 2-way set associative and with a hit time of 50ns. We assume local memory access time of 300ns, and remote memory access time of 1200ns (ratio 1:4) for CC-NUMA configuration and 3000ns (ratio 1:10) for CC-NOW configuration (we

assume the 1000 ft. of fiber traversed in CC-NOW causes approximately 2000ns of latency). Finally, the OS we use is Silicon Graphics' IRIX 5.2.

5.0 Workloads

The choice of workloads used to evaluate the benefits of migration/replication is a critical factor affecting the value of the study. In this report we use five "real-life" workloads capturing some of the major uses of compute servers. These include: (i) a multiprogrammed collection of large sequential engineering applications (verilog and Flash simulations); (ii) a single optimized parallel graphics application from the SPLASH suite (Raytrace); (iii) a multiprogrammed collection of parallel scientific applications using a space-partitioning scheduling policy [TuG89] again from the SPLASH suite (Raytrace + Vol.Rendering + Ocean); (iv) a Sybase system with multiple SQL servers executing decision support queries; and finally (v) a multiprogrammed code-development environment (parallel compilations). For all workloads we observe both kernel and user references and both instruction and data references. We believe this is a more diverse and realistic collection of workloads than used in any of the previous studies. A summary of the workloads is given in Table 2.

Table 2 Description of the workloads. For each workload, we list the applications in the workload, the number of processors (PEs) used, and a short description of the workload.

Name	Contents	PEs	Notes
Engineering	6 Flashlite, 6 Verilog	8	multiprogrammed, compute intensive, serial applications
Raytrace	Raytrace	8	parallel graphics application, rendering a scene
Parallel	Raytrace, Volrendering, Ocean	8	multiprogrammed, compute intensive, parallel applications, space multiplexed scheduling
Database	Sybase	4	commercial database, with decision support queries
Pmake	4 four-way Make	8	software development, compile of gnuchess

Before describe the applications we note that each of the workloads is very long running, so it was not possible for us to observe the workloads from start to finish. Instead, we do the following. SimOS allows for fast and slow modes of execution. In the fast mode, instructions and memory references are executed directly by the host processor, and the target hardware system's caches are not simulated. We used the fast mode to get each workload deep into its execution ("steady state", past all initialization), and then take a checkpoint of the complete system state. We then used the slower detailed mode of SimOS, which executes the workloads for several seconds after that checkpoint, modeling all memory hierarchy details. The detailed numbers we report below in Table 3 and Table 4 correspond to this interval beyond the checkpoint. It is obvious from these numbers that we are able to observe a fair amount of the application execution, which corresponds to a large number of instructions and memory references (in the billions). We now describe the salient features of each of the workloads in greater detail.

Multiprogrammed Engineering Workload (Engineering): Our first workload consists of large sequential computation- and memory-intensive applications (six large verilog simulations and six large Flash machine simulations). This represents one of the most common uses of multiprocessors today. The verilog simulator we use is VCS (commonly used in large industrial projects); it compiles the simulated circuit into code and so has a very large code segment, and

Table 3 Execution time and memory usage of the workloads. We show the cumulative execution time of the workload, the percentage of the execution time spent in Idle, Kernel and User modes, the percentage of time spent stalled on the secondary cache for instructions and data, and the number of instruction and data pages touched during the workload. The numbers for stall time and the pages touched are separated into Kernel and User.

Work load	Cum CPU Time (s)	CPU Time Breakdown (%)			Stall Time (%)				Pages Touched			
					Kernel		User		Kernel		User	
		Idle	Kern	User	Instr	Data	Instr	Data	Instr	Data	Instr	Data
Engr.	52.84	0	4	96	0.5	1.0	26.5	28.7	138	1155	1077	5955
Raytrace	30.78	5	6	89	0.1	3.0	5.8	32.5	165	1098	128	7257
Splash	53.49	8	7	85	0.9	2.9	2.8	37.5	164	3860	184	14563
Database	10.85	36	11	53	1.3	4.8	3.3	28.1	197	3292	581	4761
Pmake	35.27	22	44	34	4.0	29.3	3.6	9.1	168	28661	518	18350

Table 4 Memory behavior of the workloads. We show the number of memory references, the number of cache misses, and the number of UTLB misses for each workload. The memory references are divided into kernel, user, read and write. The cache misses are similarly divided and they separate out the instruction misses too. The UTLB misses are separated into read and write misses.

Work load	Mem References (millions)				Cache Misses (millions)						UTLB Misses (millions)	
	Kernel		User		Kernel			User				
	Read	Wr.	Read	Wr.	Read	Wr.	Instr	Read	Wr.	Instr	Read	Wr.
Engr.	40.73	5.71	1047	689	0.55	0.20	0.42	6.25	5.99	11.4	27.78	2.21
Raytrace	25.11	8.35	1480	494	0.61	0.27	0.46	7.21	0.88	1.41	7.46	0.19
Splash	37.10	16.26	2107	687	1.45	0.89	0.76	12.81	3.43	1.27	9.53	0.25
Database	14.99	8.06	158	69	0.39	0.39	0.21	1.34	1.27	0.30	3.23	0.30
Pmake	94.93	88.95	547	260	3.38	5.06	1.15	1.38	1.26	1.04	4.12	0.57

generates a high I-cache miss rate (see Table). With multiple instances of the same application running, this generates interesting opportunities for benefits from replication of code pages. For this workload, we use regular UNIX priority scheduling with affinity. As the tables show, this workload spends very little time in the kernel (4%), so we will focus only on migration/replication for user code and data.

Dedicated Single Application (Raytrace): One common workload expected to run on compute servers is single compute-intensive parallel applications. As representative of this class, we have chosen Raytrace, a widely used graphics algorithm for rendering images. This particular Raytrace program come from the SPLASH2 suite [SWG92], and we use the car.env (a car model) as the input dataset. This application forms an interesting workload for the replication study, since it has widely shared data segments. The processes of the application were locked down to specific processors, as would be the case in such a dedicated-use workload. As the tables show, this workload spends very little time in the kernel (6%), so we will focus only on migration/replication for user code and data.

Multiprogrammed Scientific Workload (Splash): The second workload consists of Raytrace, Volume rendering, and Ocean applications from SPLASH in a multiprogrammed setting. For

this workload we use space-partitioning *scheduler-activations-like* scheduling as described in [ABL+91]. The applications effectively enter and leave the system at different times, so the number and specific set of processors assigned to an individual application changes over time. For example, in this workload, Raytrace starts with 8 processes, then reduces to 4, then 3, then 6 and finally ends with 8, because of arrival and departure of Volume rendering and Ocean. This is an interesting test case for automatic page movement, because static placement does not do well with such dynamically changing processor allocations**.** Both Raytrace and Volume rendering have substantial read-mostly structures that may be replicatable. The access to these structures from processors is fairly unstructured and efficient static placement is difficult as we show later. As the tables show, this workload spends very little time in the kernel (7%), so we will focus only on migration/replication for user code and data.

Decision Support Database (Database): Our fourth workload represents use of CC-NUMA/CC-NOW machines for database applications. Our workload models decision support type queries on an off-line main-memory database.We also considered a workload based on the TPC-B benchmark. However this would have entailed updates to somewhat random entries in a large table, and so there would have been little potential for benefits from migration/replication. Our database engine is Sybase. It 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. We use the decision support queries from the Metaphor benchmark for this workload. As we show later, there is a lot of synchronization and locking behavior in this workload that adversely affects the possibilities for migration and replication.

Multiprogrammed Software Development (Pmake): Our final workload represents a software development environment typically consisting of many small, short-lived jobs such as compilers and linkers. Our workload consists of four Pmake jobs each compiling the gnuchess program with 4-way parallelism. The workload is I/O intensive with a lot of system activity. Again, UNIX priority scheduling with affinity is used for this workload. In this workload, the majority of the memory stall time arises due to kernel instruction and data references (33%) rather than from user references (13%). For this reason, with this workload we focus on migration and replication potential in the kernel.

In assembling the workloads we have picked realistic and important workloads that would be of interest to users of compute servers. The execution window that we studied is long enough to capture the essential characteristics of the applications. The workloads themselves show significant memory stall times with a reasonable potential for benefitting from improvements in memory locality. These benefits are only likely to increase further with future faster processor making the choice of workloads more significant.

6.0 Exploring the Policy Parameter Space

In Section 3 we had described the various parameters (reset interval, trigger threshold, hold threshold, write threshold, migrate threshold) which determine the effectiveness of the migration/replication policy. In this section we explore the impact of various choices that we make for these policy parameters. As mentioned in Section 4, we use traces for this part of the analysis because they provide perfect information about cache misses, and since the runs are

cheap, we can perform a more extensive exploration of the policy space to study the sensitivity of the results to variations in the parameters. The traces also offer a determinism in the runs across policy variations; this obviously has its benefits and drawbacks, but we believe that for improving our basic understanding, the benefits are substantial. Our analysis in the section is based on full cache-miss traces; in Section 7 we will consider the effect of using less than perfect information on the results. In Section 8 we will discuss actual implementation results.

6.1 Tracing Methodology and Policy Parameters

We use SimOS to run the workloads and non-intrusively generate a detail trace. The trace contains all secondary cache and TLB misses, with information about the processor taking the miss, a timestamp, and whether it was user or kernel, code or data, and a read or a write. The trace is then post-processed by an analysis program that mimics a policy based on the parameters given and provides the number of misses that would be local and remote, the number of pages that will be migrated or replicated, and other relevant information for this policy. For the results in this section we ignore the cost of collecting the information required to make policy decisions. The cost to do page movement is modelled as a fixed cost per migrate, replicate, or collapse. From actual runs of the kernel implementation of the policy on a contention-less NUMA memory model, we see that the average cost per page movement is about 350µs for CC-NUMA. We estimate that the cost for CC-NOW will be about 1ms. We also do not model contention effects or the change in execution interleaving of the workload when replication and migration make a greater number of misses local.

For the trace analysis we establish a base case for the policy parameters and compare the results of this base case against static placement policies, replication-only and migration-only policies. Subsequently we examine the sensitivity of the results to each of the parameters. The base case we picked was the one that on average did the best across all the workloads. The parameters for the base case are trigger threshold at 128 (which is twice the number of cache lines in a page), hold threshold at 32 (a quarter of the trigger threshold), reset interval 32 million cycles (approximately 100 ms), write threshold 1 and migrate threshold 1. The reset interval is set to be 32 million cycles (100ms), large enough so that the resetting of counters in a practical implementation will not be an appreciable overhead.

6.2 Workload Results

We first compare the base policy with other page allocation strategies, exploring the potential performance gains. Next we vary the parameters in the base policy to examine their effect on workload performance.

6.2.1 Base Policy Results

As shown in Table 3, the engineering, splash, raytrace and database workloads spend most of their execution and memory stall time in user mode. Because they have very little kernel activity we will first use these four workloads to study user performance with page migration and replication. We then use the pmake workload, which has a significant level of kernel activity, to study how migration and replication affect kernel performance.

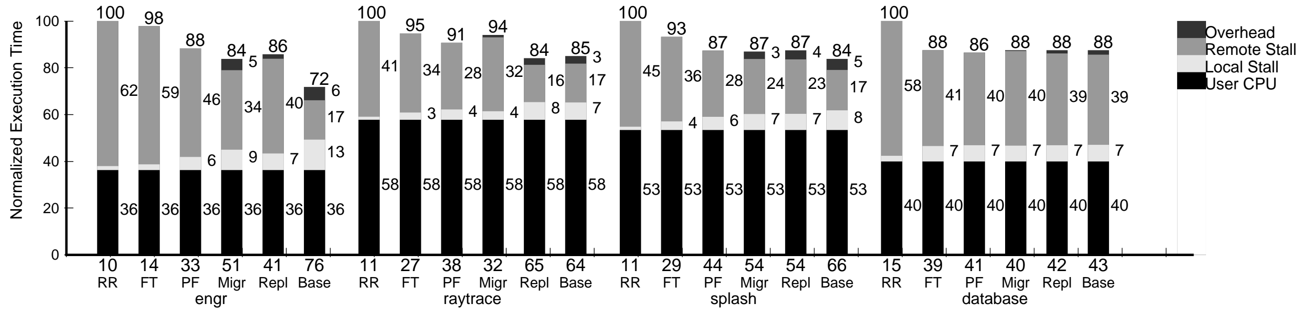

FIGURE 6.1. Breakdown of user execution time for various policies for CC-NUMA architecture with 1200ns remote latencies. There are six runs for each workload, three static allocation policies Roundrobin (RR), First touch (FT), and Post-facto (PF), and three dynamic policies Migration-only (Migr), Replication-only (Repl), and the combined migration/replication policy (Base). Each bar shows the execution time for that policy normalized w.r.t. the RR policy (which is 100). Additionally each bar separately shows User CPU time, cache-miss stall to local memory, cache-miss stall to remote memory and the overhead to migrate and replicate pages. The percentage of misses that were to local memory is shown at the bottom of each bar.

Workload performance with the CC-NUMA model is shown in Figure 6.1. For each workload six page allocation strategies are shown; three static allocation policies, roundrobin (**RR**) which is equivalent to random allocation, first touch (**FT**) where the page is allocated to the processor that first touches it, post facto (**PF**) which is the best static allocation case and assumes future knowledge, and three dynamic policies migration only (**Migr**) with trigger 128, replication only (**Repl**) with trigger 128, and the base case combined policy outlined above (**Base**).

Overall we see that for 3 of the 4 workloads shown, policies doing migration or replication or both generally outperform the static policies, even considering that the PF policy that assumes perfect future knowledge. By improving memory locality, they reduce the remote stall time component. This reduction is more than the increase in local stall time and overhead and so there is a net reduction in execution time. We also see that doing both replication and migration can do better than either by itself. The extent of the improvement relative to round robin varies, from 28% in the case of the engineering workload to 12% in the case of the database workload.

In addition to the results shown in Figure 6.1, we did a more detail analysis for each workload for code and data separately, to better understand the benefits seen with each policy. We now consider the performance of the policies for each workload in detail, comparing benefits from replication and migration, and from code and data misses.

Engineering workload: From Table 3 and Table 4, the engineering workload shows a large user stall time, about 55% of non-idle time, and the misses are about equally divided between code and data. For the code misses, there are multiple copies of each application (VCS and Flashlite) in the workload, which leads to read-only shared access that can be made local through replication. However, for the data misses, this workload consists of sequential applications and so the data accesses are unshared. These misses can be made local through page migration (coupled with affinity scheduling). As expected, the “Repl” and “Migr” policies are successful at improving the execution time of this workload through replication of code and migration of data respectively. This workload clearly brings out the point that you require a policy that does both migration and replication to fully exploit memory locality. The “Base” policy by doing both is able to make 76% of the misses local which translates to an improvement in total execution time of 28% over the

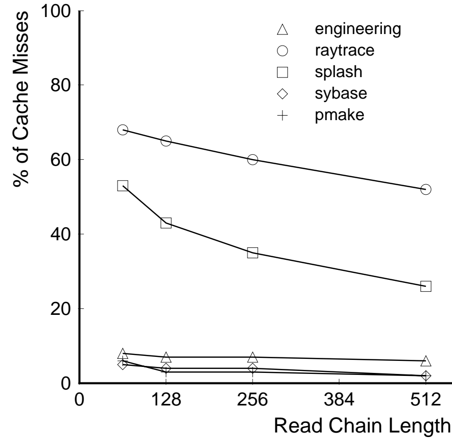

FIGURE 6.2. Percentage of data cache misses in read chains. A read chain represents a string of reads to a page from a processor which is terminated by a write from any processor to that page. A long read chain indicates a page that could benefit from replication. The X-axis shows read chain lengths and the Y-axis shows the percentage of the total data misses that are part of read chains of that length or more.

"RR" policy. The memory cost of doing replication is increased memory usage. There are 1077 code pages that are touched in this workload. If code replication was done on first touch as has been suggested, this could potentially increase usage by 7539 pages. Table 5 shows that by selecting hot pages, the "Base" policy increases usage by only 2307 pages representing an increase of 32% in memory usage for the workload.

Table 5 Comparison of the Base and RR policies. We show Pages Replicated and Migrated, the Average and Maximum pages utilized, Percentage of misses to local memory, and the Stall, Overhead and Run time.

Work load	Policy	Pages moved		Pages utilized		Percent Local	Time (seconds)		
		Repl	Migr	Avg.	Max.		Stall	Ovhd	Run
Engr	RR	0	0	5028	7032	10	29.2	0	48.06
	Base	2394	5071	6731	9339	76	13.7	2.6	35.16
Raytrace	RR	0	0	5279	7379	11	11.6	0	29.1
	Base	1727	780	6031	8798	64	6.6	0.9	25.0
Splash	RR	0	0	10792	14739	11	21.4	0	49.4
	Base	2013	4131	11341	16355	66	11.8	2.2	42.0
Database	RR	0	0	4418	5332	14	3.5	0	7.0
	Base	209	103	4595	5533	43	2.6	0.1	6.2

Raytrace: Raytrace is an interesting parallel application for this study, since a large portion of its data, representing the scene to be rendered, is shared and read-mostly. It has about 38% user stall time, and unlike the engineering workload most of the stall is due to data misses. We characterize the read-only nature of data in Raytrace and this is graphically illustrated in Figure 6.2,. Raytrace has almost 60% of its data misses in read chains that are longer than 512, and since this is shared data, it indicates good replication potential. Since both the data and code are shared, migration will not provide much benefit. From Figure 6.1 we see that both the "Repl" and the "Base" policy perform well reducing the execution time by 15% over the "RR" policy. This workload touches

7257 pages of code and data. From Table 5, the additional page usage through replication is only 1419 page, a 20% increase. Static data placement or replication would be difficult in the raytrace workload because of its unstructured data access. This then is a clear example of dynamic data replication improving execution time and reducing the memory overhead of replication.

Splash workload: This workload of parallel applications includes raytrace, volume rendering and ocean. It has about 40% stall time, which is almost entirely due to data misses. Raytrace and volume rendering, both have unstructured accesses to read only data sets and so can benefit from replication. Figure 6.2 shows about 30% of the data misses in this workload in read chains longer than 512, so confirming the replication benefit from data. Ocean has only nearest neighbor communication and so mostly unshared access to data. Ocean's cache misses can be made local with a first touch type policy. However, in a multiprogrammed workload, processes will have to be moved across processors to achieve load balancing. In this scenario, migration of code and data to the processor running the process will be needed to improve the data locality [CDV+94]. Given the type of data accesses just described, the "Migr" and the "Repl" policies are each only able to capture a part of the potential benefit. The "Base" policy, by doing both performs better, with a reduction in execution time of 16%. Table 5 shows that the base policy increased memory usage just 11% through replication.

Database workload: For this workload we chose a set of decision support queries expecting that the data accesses would be mostly reads and so the workload would benefit from replication. Since we locked the servers to the processors we do not expect much benefit from migration. Figure 6.1 shows that first-touch reduces remote stall time, improving performance by 12% over round robin. However, there is very little additional benefit for any other policy over first touch. Table 4 shows that almost half of the user data cache misses are writes. 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 "read-write" pages seem to be mainly for synchronization purposes with fine grain sharing by all the processors. This sharing pattern of type 3 outlined in section 3 cannot benefit from replication or migration. Consequently no policy gives much additional benefit beyond first touch.

Figure 6.3 shows the performance of the various policies on CC-NOW type systems. The results are similar to the CC-NUMA case. The larger remote latency of CC-NOW systems results in greater performance improvements for the dynamic policies. The improvement for the Base policy over RR ranges from 44% for engineering to 20% in the case of the database workload.

In summary, for the workloads where we analyzed user misses, three of the four show potential for substantial performance improvements using dynamic policies. Both migration and replication are needed to exploit the full potential of memory locality. Dynamic replication of data is effective and essential for improving performance in the raytrace and splash workloads. Also dynamically selecting hot code pages to replicate can substantially reduce memory usage compared to a possible replicate on first miss policy.

Pmake workload: We use the pmake workload to study the potential for replication and migration in the kernel, since it has significant kernel activity and much less user stall time (see Table 3). The kernel can be viewed as a large parallel program with shared code and data. Most

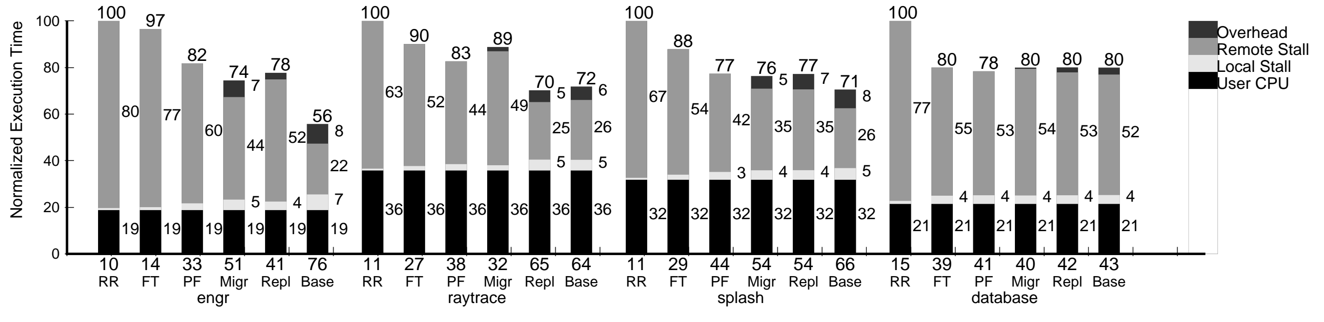

FIGURE 6.3. Breakdown of user execution time for various policies for CC-NOW architecture with 3000ns remote latencies. There are six runs for each workload, three static allocation policies Roundrobin (RR), First touch (FT), and Post-facto (PF), and three dynamic policies Migration-only (Migr), Replication-only (Repl), and the combined migration/replication policy (Base). Each bar shows the execution time for that policy normalized w.r.t. the RR policy (which is 100). Additionally each bar separately shows User CPU time, cache-miss stall to local memory, cache-miss stall to remote memory and the overhead to migrate and replicate pages. The percentage of misses that were to local memory is shown at the bottom of each bar.

UNIX kernels today are loaded in memory at boot time and their code and data are not mapped through the TLB. Therefore dynamic replication and migration of kernel code and data is not possible in these implementations. However, using our traces we can investigate the potential benefit from migration and replication. Figure 6.4 shows the effect of applying the various policies to 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. However this accounts for only about 12% of the misses (see Table 4). Kernel data shows no real benefit beyond that of the static "FT" policy. The "FT" policy gives some benefit over "RR" because there are some kernel structures that have a natural affinity to a particular processor, e.g. the Private Data Area (PDA), a per processor structure, and the Page Frame Descriptors (PFD) for memory local to a processor. There could be a small potential benefit from page migration, for structures that are per process, e.g. user page tables, but the realization of this migration benefit is

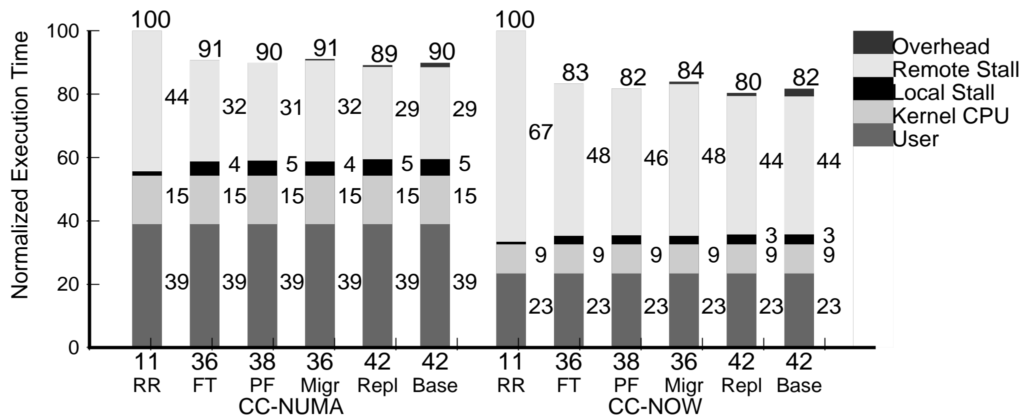

FIGURE 6.4. Execution time breakdown for the pmake workload for both CC-NUMA and CC-NOW. Only Kernel misses are considered for the different policies. There are six runs for each configuration, three static allocation policies Roundrobin (RR), First touch (FT), and Post-facto (PF), and three dynamic policies Migration-only (Migr), Replication-only (Repl), and the combined migration/replication policy (Base). Each bar shows the execution time for that policy normalized w.r.t. the RR policy (which is 100). Additionally each bar separately shows User time, Kernel CPU time, cache-miss stall to local memory, cache-miss stall to remote memory and the overhead to migrate and replicate pages. The percentage of misses that were to local memory is shown at the bottom of each bar. There is no significant improvement beyond FT.

similar to that of unshared user data and is dependent on the scheduling of the associated user process.

Table 6 Breakdown of Kernel D-cache Misses by Kernel Segment for the Pmake workload. We show total misses to each segment and misses to read-mostly blocks, We consider blocks to be pages (4KB) or Cache lines (64Bytes) or Words (4Bytes). We use read-mostly (< 1% writes) instead of read-only, since we want to consider pages that may be written very infrequently. Other than text, only a small fraction of the pages are read-mostly.

Kernel Segment	Total Misses 1000's	Misses to Read-Mostly Blocks (<1% writes)		
		Page (4KB)	Line (64Byte)	Word (4Byte)
User data copyin/copyout	4,022 (43%)	234 (2%)	257 (3%)	513 (5%)
Kernel initialized data	1,970 (21%)	24 (0%)	48 (1%)	238 (3%)
Kernel allocated tables	1,491 (16%)	1 (0%)	43 (0%)	351 (4%)
Kernel heap	173 (2%)	1 (0%)	10 (0%)	26 (0%)
U area	243 (3%)	0 (0%)	1 (0%)	50 (1%)
Kernel stack	238 (3%)	0 (0%)	0 (0%)	5 (0%)
Page tables	72 (1%)	72 (1%)	72 (1%)	72 (1%)
Private data area (PDA)	33 (0%)	0 (0%)	0 (0%)	2 (0%)
Kernel text	1,157 (12%)	1,157 (12%)	1,157 (12%)	1,157 (12%)
Total	9,400 (100%)	1,489 (16%)	1,588 (17%)	2,414 (26%)

We classified the kernel misses by the structures accessed as shown in Table 6, to see if the lack of replication benefit was a basic characteristic of the kernel or a result of the kernel data not being structured to take advantage of page replication, i.e., read-only structures are not grouped together on pages. We expand the scope beyond read-only structures, to also include structures that are written infrequently (we consider structures where writes are less than 1% of the misses as "read-mostly"). We analyzed the kernel miss traces at the granularity of pages, cache lines, and words and the results are shown in Table 5. Of the 9.4 million total kernel misses, only 1.25 million misses to kernel data are potentially "read-mostly". The largest part is when reading data for copyin which represent the kernel copying pages in from user space. These may not really be all "read-mostly" since we do not see the writes which may happen in user space. The kernel stall time for this workload was 42% of non-idle time, so even if all the 1.25 million misses which could be to "read-mostly" structures are made local through replication, the improvement in execution time would be only about 5% (without accounting for the overhead). This leads us to conclude that even at a word granularity there is not enough "read-mostly" kernel data to give any substantial benefit from dynamic replication.

6.2.2 Sensitivity to Policy Parameters

Our above analysis for the workloads shows that there are substantial performance gains from replicating shared code and "read-mostly data and from migrating unshared code and data. Having considered the performance of a few basic policies, we now examine the sensitivity of the results to the different parameters in the policy. The parameters of interest to us are the trigger threshold, the hold threshold and the reset interval. We also study the effect of restricting the additional memory used for replication.

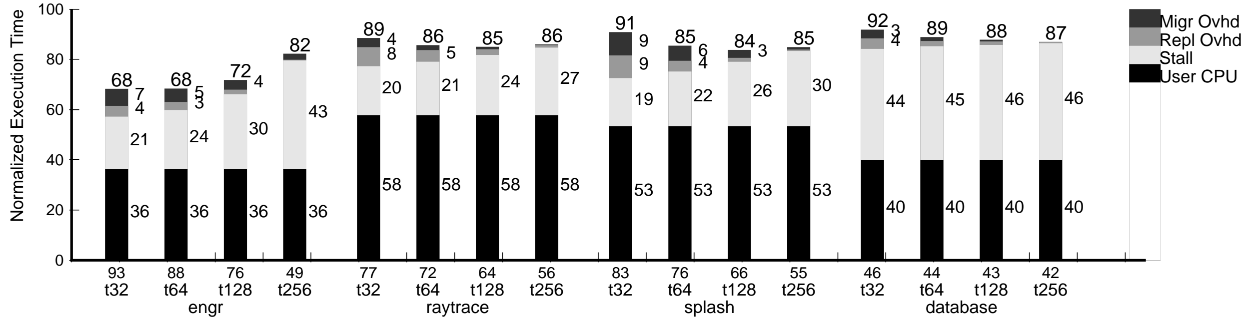

FIGURE 6.5. Variation in performance with the trigger threshold. Each workload is run with four trigger thresholds, 32, 64, 128, and 256. The hold threshold is a quarter of the trigger threshold and trigger 128 is that base case. Each bar shows the run time for a configuration normalized to the run time 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.

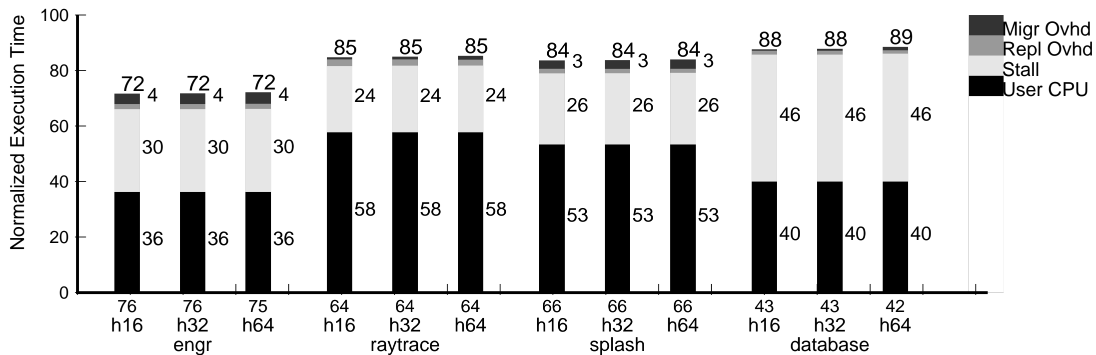

FIGURE 6.6. Variation in performance with changes in the hold threshold. Each application is run with three different hold thresholds, 16, 32, and 64. Trigger threshold is 128 for all cases and hold threshold 32 is the base case. Each bar shows the run time for a configuration normalized to the run time for the Roundrobin allocation case. 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. Variations in the hold threshold have minimal impact on performance.

Trigger threshold: The trigger threshold controls the aggressiveness of the policy and provides a trade-off between increased memory locality resulting in reduced stall time and the total overhead to migrate and replicate pages. Figure 6.5 illustrates this trade-off clearly for the different workloads. A smaller trigger makes more misses local and reduces stall time, but increases the overhead. Given the increase in both the costs and the benefit, 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 for the workload. The trigger threshold is clearly a critical parameter and a more sophisticated policy implementation could be considered that varies it automatically at runtime (through feedback) to control the rate of migrations and replications to respond to different situations, e.g., excessive memory pressure or lock contention.

Hold threshold: The hold threshold is used to differentiate between shared and unshared pages and so decides whether a hot page is potentially migrated or replicated. A higher hold threshold potentially favors migration over replication. The results in Figure 6.6 show that the performance

seems to be quite insensitive to the value of the hold threshold within a reasonable range. This result holds across all the workloads. The performance being insensitive to the hold threshold points to the fact 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 ambiguous sharing behavior.

Reset interval: The reset interval converts a count of cache misses to a cache miss rate. Figure 6.7 shows that the larger reset interval, 400 ms instead of 100ms seems to not affect the results significantly. The local/total cache miss ratio does not change significantly and the small resulting reduction in stall time is compensated by the small increase in overhead. This indicates that pages are either hot (many misses) or cold (few misses). There are not too many that are borderline w.r.t the trigger threshold, that would get hot with a larger reset interval. Resetting the counters is an overhead in the implementation of the policy. If this overhead turns out to be significant, based on the above result, we can reduce this overhead by using a longer reset interval without impacting the performance in any significant way.

Memory Pressure: All the previous runs were done without any memory pressure. We studied the performance of the base policy, in the presence of memory pressure. The policy responds to memory pressure, by stopping replication and reclaiming replica pages that have not been recently referenced till the memory pressure drops. We specify memory pressure as a maximum number of pages that are available. Table 7, compares the performance of the base policy with and without memory pressure. When memory is restricted, achieved memory locality is still very good and the percentage of additional pages used in this case is quite small, 0.3% for Splash to 11% for Engineering. The additional cost is an increase in the number of replications and to a lesser extent migrations. This is due to replicas being reclaimed under memory pressure and then later being replicated again (or migrated). One could also consider reclaiming infrequently accessed non-replica pages under memory pressure, however this must be done carefully else it could increase the number of disk accesses.

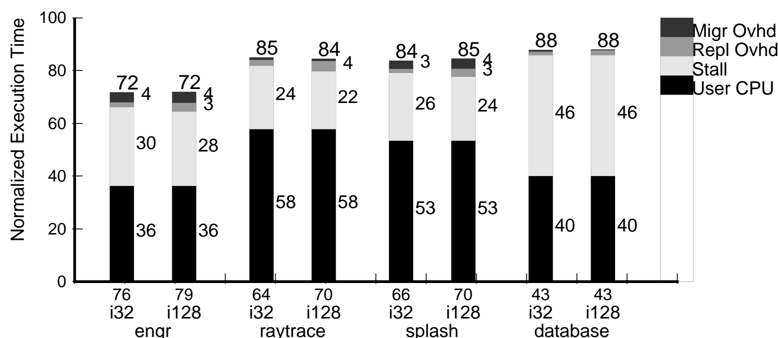

FIGURE 6.7. Variation in performance with changes in the reset interval. Each application is run with two different reset intervals, 100ms (i32) and 400ms (i128). Trigger threshold is 128 and hold threshold is 32 in both cases, 100ms reset interval is the base case. Each bar shows the run time for a configuration normalized to the run time for the Roundrobin allocation case. 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. Variations in the reset interval have minimal impact on performance

Table 7 Effect of Memory Pressure on policy performance. Each workload is run in two configurations. First with unlimited additional pages available for replication. Second with total memory restricted such that only half the additional pages used in the unlimited case are available. We show pages moved through replication and migration, Average and Maximum pages utilized, Additional pages used as a percentage of the pages used in the RR case and the percentage of misses that were to local memory. We see a gradual reduction in%Local as the total available pages is reduced.

		Pages moved		Pages Utilized		Addl Pages Used (%)	Percent Local
	Avail. Mem.	Replicated	Migrated	Average	Maximum		
Eng	Unlimited	2394	5071	6731	9339	33	76
	8186 pages	5905	5214	5858	7778	11	72
Ray.	Unlimited	1727	780	6031	8798	19	64
	8089 pages	2557	987	5597	7685	4	61
Spls	Unlimited	2013	4131	11341	16355	11	66
	15547pages	2569	4291	10929	14790	0.3	64

7.0 Quality of Information

In the previous section we showed that the memory stall time of some of the workloads can be reduced significantly through page replication and migration. These policies were based on perfect information, counting all cache misses. Cache miss information to this level of detail is not available in the OS. In this section we examine other forms of information that are available in current or planned systems and discuss the costs involved in collecting this information. We will study their accuracy in approximating perfect cache miss information and their effect on the performance of the policies.

7.1 Alternate Sources of Information

To maintain cache-coherence, CC-NUMA systems have a directory controller implemented in hardware or software. The directory controller on a node has access to every cache miss that originates from the node or has to be satisfied by memory on the node. These controllers are highly optimized to do the necessary processing per cache miss very efficiently, so counting and bucketing every cache miss by processor would add an unacceptable overhead. It is however feasible to sample this information and provide feedback to the page migration and replication policies. The sampling frequency can be used to trade-off the overhead of collection with the accuracy of the information. A scheme similar to this is to be implemented in the MAGIC directory controller chip for the FLASH machine being designed at Stanford [Kus+94]**.** This sampled cache miss information will be evaluated as a potential substitute for full cache miss information.

Another source of information about memory references is the TLB. Every user access to data or instructions also needs a TLB access to translate the virtual page number into a physical page number, Therefore TLB misses could be a possible approximation for cache misses. The TLB can be viewed as a cache with the line size being a page. Therefore the validity of the approximation would depend on the access patterns of the application and the size and architecture of both the cache and the TLB. Since the TLB misses are handled in software by the OS, the miss handler could be modified to count the TLB misses. On IRIX the user TLB miss handler is highly tuned and takes about 15 cycles, so adding counting to this could increase the overhead of this fairly frequent operation. To reduce the counting overhead, sampling of TLB

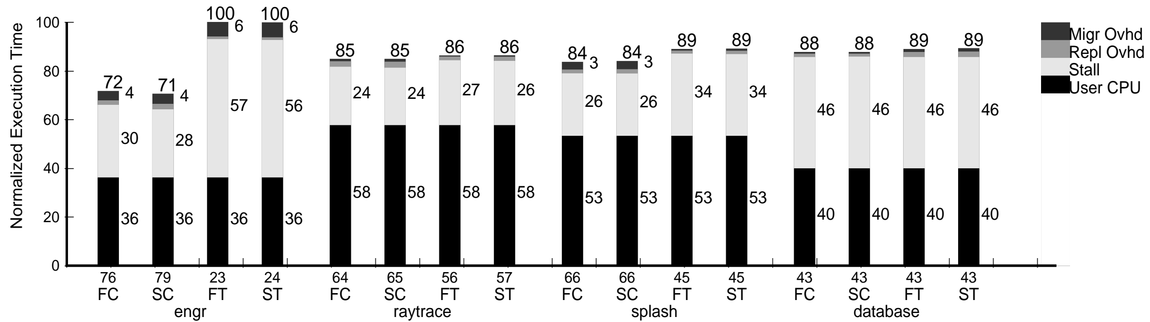

FIGURE 7.1. Performance impact of approximate information. There are 4 bars for each workload, Full cache (FC), Sampled cache (SC), Full TLB (FT), and Sampled TLB (ST). All sampled cases are sampled with ratio 1:10. Each bar shows the run time normalized to the run time for the Round robin case. The run time is broken down as User CPU, User stall (local and remote), Overhead for replication and migration separately. The percentage of misses made local is shown at the bottom of each bar. Sampled cache does as well as Full cache in all cases.

misses can be done in software, with an action being performed only on every Nth TLB miss. We will evaluate both TLB misses and sampled TLB misses as potential sources of information.

7.2 Effectiveness of 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. To study the effectiveness of using the above forms of approximate information we rerun selected policies with the appropriate source of information. The trigger and hold threshold counts are based on sampled cache misses, TLB misses, or sampled TLB misses. A sampling rate of 1 in 10 is used, as this should be sufficient to reduce the collection overhead. The thresholds are scaled based on the sampling rate, so a trigger threshold of 128 with full misses will be equivalent to a threshold of 13 with 1 in 10 sampling.

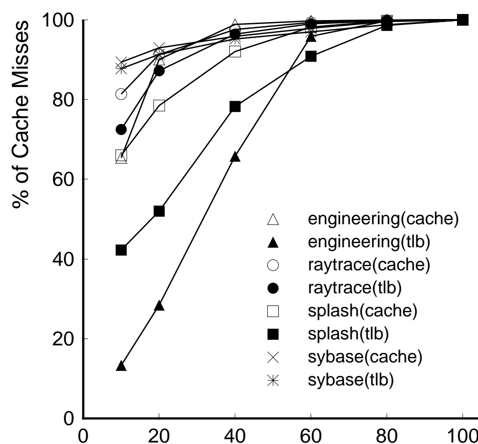

FIGURE 7.2. Effectiveness of TLB misses and cache misses in finding hot pages. The X-axis is percentage of pages considered. The pages are sorted in descending order based on the number of misses taken, cache misses for one set of graphs and TLB misses for the other. The Y-axis shows the percentage of the total cache misses that are to the pages considered. A close correspondence between the cache and TLB lines for a workload indicates that hot cache miss pages are also hot TLB miss pages. This does not seem to be the case for engineering and to a lesser extent for splash and raytrace.

The results for the policies with approximate information for each of the workloads is shown in Figure 7.1.

Sampled cache: Comparing the sampled cache based policies with the policies that use full cache information, the striking point is that the performance is almost identical across all the workloads. This is true for both the local/total ratio and the overhead for replication and migration. The Flashpoint study [MOH96] implements functionality which includes the counting of cache misses in the MAGIC directory controller. This functionality is at least as complex as what we need and, without using sampling, they report an increase of about 2 - 5% in application execution time. Therefore, the 1 in 10 sampled cache case evaluated here is very attractive, both for accuracy and for performance.

TLB: Our results in Figure 7.1 show that TLB misses are a somewhat inconsistent metric to use for the policies. The performance is worst in the engineering workload with a local/total ratio of only 23% as opposed to 76% with perfect information. On the other hand, the TLB metric performs quite well for the database workload achieving the same local/total ratio. The performance of the other two workloads, Splash and Raytrace, fall in between these extremes. To explain this difference in behavior we use Figure 7.2, which shows the effectiveness of using TLB misses in finding the hot user pages. The graph partly explains the difference in performance of the workloads with the TLB metric. In Figure 7.2, the difference in behavior between using TLB misses and cache misses (the separation of the corresponding graphs) is largest for the engineering workload and least for the database workload. Other factors that contribute to the reduction in performance with TLB misses are the difference in the absolute number of TLB and cache misses for a workload and the difference in the read/write ratio of each metric (see Table 4).

Sampled TLB: Figure 7.1 also shows that sampled TLB performs about the same as TLB. Sampled TLB misses would be an adequate metric for the policy, for workloads with a good correlation between TLB misses and cache misses.

8.0 Performance Results

In sections 6 and 7 we used cache and TLB miss traces to explore the policy space and study the effect of approximate information. The results showed potential performance improvements of 28% for CC-NUMA and 44% for CC-NOW type systems. The trace based analysis used a fixed per page overhead and was useful for exploring potential benefits. However it does not completely represent all the costs in the system and all the benefits from doing the replication and migration of pages. To explore the realistic costs and benefits and validate the trace results, we implemented the dynamic policy, used in the trace-based studies, in the IRIX 5.2 kernel. As described in section 4, we use SimOS [HWR+95], a realistic hardware simulator for our runs allowing us to simulate a CC-NUMA machine based on the FLASH multiprocessor [Kus+94]. SimOS also enables us to non-intrusively collect detailed information about cache misses and the kernel overhead associated with migrating and replicating pages.

We first describe the important aspects of the implementation of the policy. We then analyze the latency of page-migration or page-replication and give a detail breakdown by function of the kernel overhead. Finally, we present performance numbers for the actual implementation.

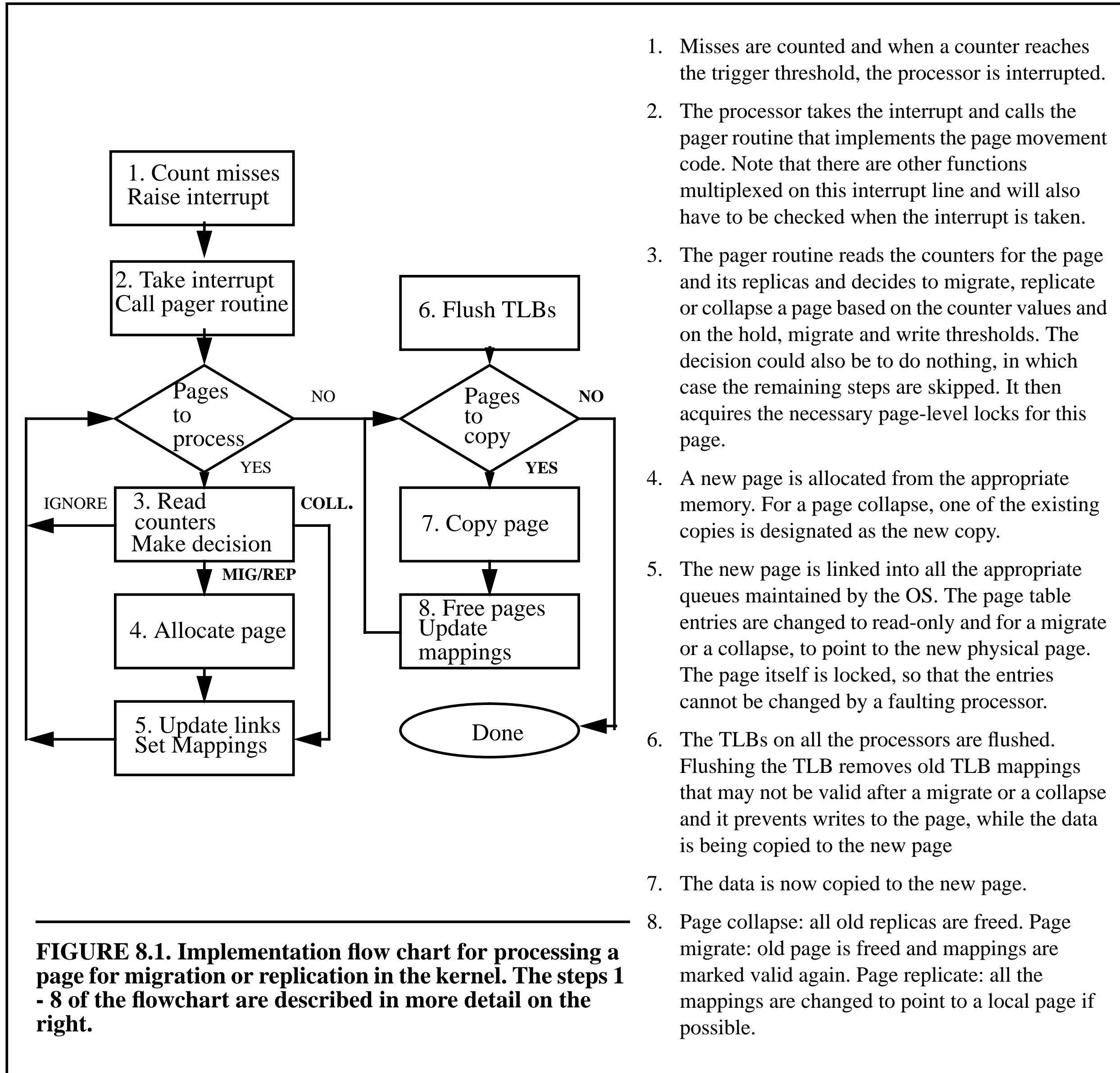

FIGURE 8.1. Implementation flow chart for processing a page for migration or replication in the kernel. The steps 1 - 8 of the flowchart are described in more detail on the right.

8.1 Implementation Details

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

The directory controller counts misses, and generates an interrupt when a page crosses the trigger threshold. The policy is implemented in the kernel as a low priority interrupt handler. Figure 8.1 describes the sequence of events in our kernel implementation of the page migration and replication policy. The flow chart shows the steps in generating and servicing a pager interrupt. There is one other path to the pager routine. The page table entries for replicated pages are marked read-only. Should a write occur to any replica, the processor traps to the protection fault handler (pfault). The pfault code directly calls a routine to collapse the replicas to a single page.

In our initial implementation, every hot page generated an interrupt. However measurements showed that the kernel overhead in taking an interrupt and flushing TLBs for each page is quite significant. To reduce this cost, the directory controller attempts to collect a few pages that have crossed the threshold before interrupting the local processor. The waiting criterion is collecting five pages or 500µs whichever comes first. This is a trade-off between reducing the per page overhead and taking more remote misses due to the delay in moving the page. Therefore on each interrupt, the code iterates over steps 3-5 for each page. A single TLB flush operation is done and then steps 7 and 8 are done for all the pages that require it. We observed that on average there are approximately two and a half pages for each interrupt.

Step 1 represents the collection of miss information. It is possible to implement collection of TLB miss information by instrumenting the TLB handler in the OS. However our results from section 7 show that using TLB misses as an approximation for cache misses does not produce consistent results. However, counting cache misses needs hardware support. The FLASH design will include cache miss counting in the directory controller using sampling of cache misses to reduce the overhead. Therefore in our implementation, we include the collection of cache miss information within the directory controller simulator.

Three important changes had to be made to the IRIX VM system to enable our implementation of replication and migration. Figure 8.1 gives an overview of these changes.

Replication support: Support was added to link together the replicas of a physical page. One replica is designated as the master page and is linked in the hash chain of physical pages. The other replicas are linked to the master page.

Finer grain locking: The version of IRIX that we used had fairly coarse locking for VM related structures. There is one lock (memlock) protecting the global hash table of active physical pages and the global free page list. There is also one lock (region lock) per region (e.g. text, data, etc.) that has to be acquired when changing page table entries. It was evident that both memlock and the region locks for shared text or data would be significant bottlenecks in the implementation of the policy. To reduce the contention on memlock, we added page level locking for manipulating replica chains. A lock was added to each page table entry to avoid having to grab the region lock when changing a mapping. However the lower-level functions that access the hash table of active physical pages and do the allocation and deallocation of pages still use memlock since changing this would require a significant redesign of the VM system. As a result of these changes, we reduce synchronization cost significantly in our implementation, however it still accounts for a significant fraction of the kernel overhead for some workloads (as high as 33% for engineering).

Page table back mappings: Support was added for back mappings from the physical page to the page table entries pointing to it. This functionality is similar to an inverted page table. This was necessary to enable changing the mappings to a page when it was replicated or migrated.

8.2 Analysis of Kernel Overhead

We ran selected workloads, with the migration/replication policy implemented in the kernel as described above. We used the Engr., Raytrace and Splash workloads for these runs as they showed significant improvements in the trace-based analysis. Our SIMOS execution environment provides detailed kernel and hardware statistics for each run. Using these statistics we analyze the

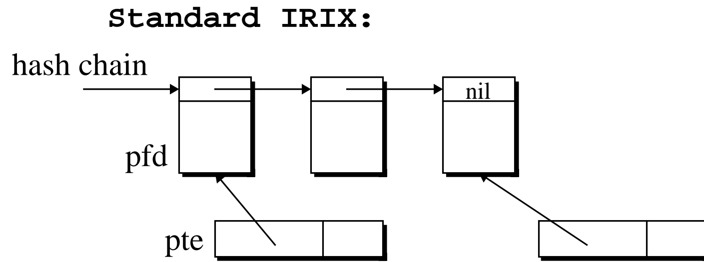

FIGURE 8.2. a In IRIX a hash table is used to translate logical pages (vnode, offset) to physical pages. Physical page frame descriptors (pfds) are linked into this open hash table through the hash chain. Also, virtual page table entries (ptes) point at the pfds, but there is no link directly from the pfd back to the pte. There is one global lock (memlock) which protects all the physical pages and multiple region locks, each of which locks all the ptes in a region.

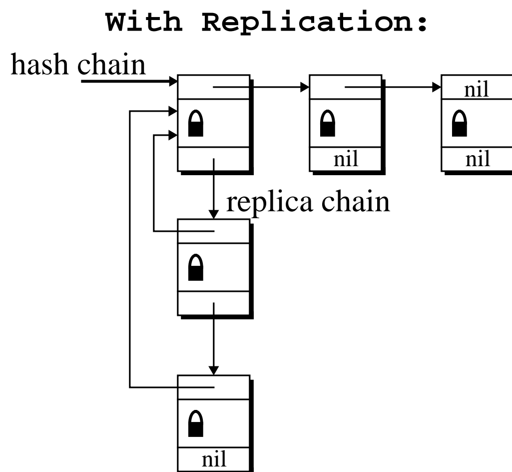

FIGURE 8.2. b All replicas of a page are linked on a replica chain. Only one of the replicas is linked into the hash table. Individual page level locking was added to lock the replica chain. Memlock is now only used for changes to the hash chain and for allocation and deallocation of page frames.

With Back Mappings:

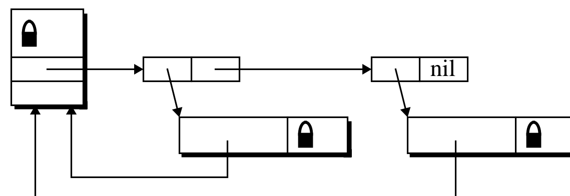

FIGURE 8.2. c To facilitate easy mapping changes, links were added to the pfd pointing back to all the ptes mapping this page. pte level locks were added so that the region lock did not have to be grabbed during a migration or replication operation. (Although the region still exists to protect other resources.)

kernel overhead to understand where the time is being spent while migrating and replicating pages.

Table 8 Latency of various functions in the policy implementation. The columns correspond to the implementation steps in section 8.1. The numbers are shown separately for Replication and Migration, where applicable. The latencies shown are in µs and is the average value for invocations of the function across the entire run.

Workload		Intr. Proc	Policy Decision	Page Alloc	Links & Mapping	TLB Flush	Page Copying	Policy End	Total Latency
Engr.	Repl.	13.0	12.6	184.3	28.6	35.9	87.0	80.5	441.9
	Migr.				75.8			63.4	472.0
Ray	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 8 shows the average latency for each of the steps 2-8 from Figure 8.1 while migrating or replicating a page. The latency for the interrupt-handling and the TLB flush are amortized over a number of pages as described in the previous section. Effective values for a single page are shown (this accounts for the variations in those latencies across workloads). The component for step 5 (Links & Mapping) is higher for migration, since during migration, the old page has to be taken off the physical page hash queue and the new page put back on. This requires acquiring the memlock. For replication, the replicas are queued on the master page and only requires acquiring a page-level lock. In step 8 (Policy End), replication takes longer, since all current mappings to the page have to be examined and set to point to the most appropriate replica.

The individual latencies are about the same for the different workloads. An interesting point to note is the increase in time for the page allocation routine for the Splash and engineering workloads. This increase is primarily due to contention for memlock (which still protects the free list and the page hash table). The engineering and Splash workloads allocate more pages than the Raytrace workload and so increase the contention for memlock. The average latency per operation is in the 400 - 500 µs range.

Table 9 Breakdown of total overhead by function. For each workload, we show the kernel overhead and the percentage of the overhead that can be attributed to the various functions. Page allocation, TLB flushing and Page copying comprise a large part of the overhead.

Workload	Kernel Ovhd(s)	Percentage of Kernel Overhead							
		TLB Flush	Page Alloc	Page Copy	Page Fault	Links & Mapping	Policy End	Policy Decision	Intr. Proc.
Engr.	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

The previous table showed the latencies of each of the individual operations involved in migrating or replicating pages. There are two additional sources of kernel overhead not accounted for in these numbers. The time spent in the kernel when it receives a request from another processor to

flush its TLB and the additional protection faults taken because pages are marked read-only during migration or replication. Table 9 shows the kernel overhead due to all replication and migration costs and its breakdown by function. Overall, the functions that cause the most overhead are the allocation of pages, the flushing of TLBs and the copying of pages.

A big factor in doing page allocation is the time spent in acquiring memlock. Detail measurements of synchronization time showed that as a percentage of total kernel overhead, it was 33% for engineering and 17% for Splash, primarily due to contention for memlock. Our implementation reduced the contention for memlock, but redesigning the VM to remove the memlock bottleneck should yield significantly better performance. This would reduce the time spent in page allocation and also in some of the other functions that acquire memlock. Due to the high synchronization overhead in the current implementation, our experiments indicate that adaptively changing the trigger to control the rate at which pages are moved can improve performance. The trigger threshold could be changed through feedback from the kernel, based on the contention for kernel resources.

The time spent flushing TLBs can also potentially be reduced. This could be done by tracking the processors that have mappings for a page and flushing only those TLBs. This functionality is not supported by the current OS and would require significant changes to the VM system.

The actual copying of bytes is not as significant an overhead as initially expected. An unoptimized bcopy done by the processor takes approximately 100µs. A pipelined memory to memory copy that is done by the directory controller in FLASH takes about 35µs. Runs with the FLASH supported bcopy show that performance could be improved with this technique.

8.3 Execution Time Improvement

As stated earlier, the workloads were run both with our implemented replication and migration policy and, for comparison, with “first touch” (FT) page allocation implemented in the kernel[1]. Table 10 shows the results of the two sets of runs for each of the workloads. We confirm that the kernel implementation of the migration and replication policy is able to improve memory locality substantially. The improvement in execution time is dependent on three factors. First, the achieved fraction of misses satisfied from local memory. Second, the contribution of user stall time to the total execution time, as this is the part that can be reduced through improved page locality. Third, the kernel overhead required to improve the memory locality. The total execution time is improved in all three cases even after considering the additional overhead in the kernel for migration and replication.

In Table 10, for raytrace, we see that the total increase in kernel time is actually less than the kernel overhead for replication and migration as shown in Table 9. To a lesser extent this is true for engineering too. This is partly due to the reduced wall clock time for the execution of the workload with page migration and replication. As a result, routines that run at regular intervals (on a clock tick) run fewer times and so reduce the kernel time used. The more important cause for the reduction in kernel time is the reduction in the average latency per cache miss, both local

1. We used a trigger value of 128 for the Raytrace and Splash workload and 96 for the engineering workload. These values gave the best execution times for the actual implementation.

Table 10 Comparison of the performance of the workload runs with "First Touch"(FT) and with the implemented policy (Rep/Mig). We show the percentage of misses to local memory, and the resulting second-level cache stall time in user mode. This reflects the improvement in memory locality through migration and replication. We also show User mode execution time which reduces due to better memory locality, Kernel mode execution time which increases due to the kernel overhead, and total execution time which shows the net improvement.

Workload	Memory Alloc. Policy	Percent misses to local mem.	User Stall time (s)	User Exec. time (s)	Kernel Exec. time (s)	Tot. Exec. time (s)
Engr	FT	12	35.4	45.7	3.6	49.3
	Rep/Mig	63	17.0	27.4	7.7	35.1
Raytrace	FT	16	28.4	51.3	18.2	69.4
	Rep/Mig	53	18.2	40.9	18.3	59.2
Splash	FT	18	28.2	56.5	15.0	71.5
	Rep/Mig	50	21.5	49.3	19.1	68.4

Table 11 Effect of improving memory locality on latency of cache misses. We show the average latency for local and remote cache misses with two different page allocation policies, first touch (FT) and our implemented migration/replication policy (Rep/Mig). We also show the improvement in latency for Rep/Mig over FT. The minimum local latency is 0.27µs and remote latency is 1.33µs. There is significant reduction of local latency by improving the memory locality.

Work-load	Policy	Average Latency of Cache Miss (µs)		Percent improvement over FT	
		Local	Remote	Local	Remote
Engr	FT	0.78	2.26	34.6	11.1
	Rep/Mig	0.51	2.01		
Raytrace	FT	1.15	2.93	40.0	5.1
	Rep/Mig	0.69	2.78		
Splash	FT	0.74	2.20	18.9	- 3.6
	Rep/Mig	0.60	2.28		

and remote, through better page locality. The FLASH memory model is more detailed and models contention at buses, in the directory controller processor and at the memory modules. In this model, a miss to local or remote memory could be dirty in another processor's cache and so resulting in 2 network hops for the local miss or 3 network hops for the remote miss. Consequently, the latency of a local or remote miss is not fixed and varies based on queuing delays. The minimum local latency is 0.27µs and remote latency is 1.33µs when the memory line is clean in main memory.

In this configuration improving memory locality has two benefits. First, the misses can be satisfied from local memory and so with a smaller latency. Second, they do not occupy a remote directory controller in addition to the local one and so actually reduce the contention in the directory controller and the network. Consequently, the latency of all misses, local and remote, is reduced. Table 11 shows the average latency for misses for the two sets of runs. Improving the memory locality reduced the average local memory access latency for each of the runs shown above, Engineering (34.6%), Raytrace (40%), and Splash (18.9%). The effect of latency reduction

through page locality was large enough to offset some or most of the kernel overhead for migration and replication.

Table 12. compares the results from the traces and the actual runs. With all the realistic overheads

Table 12 Performance improvement with page movement over "First Touch". We show results for the trace-based analysis and the FLASH architecture. The achieved memory locality in FLASH is lower than the traces, but because of the contention effects in the real architecture, the actual execution time improvement can be greater.

Workload	Memory model	Percentage of misses to local memory	Percent improvement over First-touch	
			User Stall time	Execution time
Engr.	Trace	76	51.5	27.7
	FLASH	63	52.1	28.9
Raytrace	Trace	64	35.0	9.6
	FLASH	53	35.8	14.7
Splash	Trace	66	35.6	9.4
	FLASH	50	23.7	4.4

of the actual implementation, the performance improvement in execution time is still significant and ranges from 28.9% for the engineering workload to 4.4% for the Splash workload. The benefits with the real run is comparable to that with traces for the "Raytrace" and the "Engineering" workloads despite the achieved local ratio being somewhat less. This is a result of the memory system contention and higher latency for misses in a real system. The trace-based analysis used a simple contention-less NUMA memory model, with fixed remote and local memory access times.

The lower local miss ratio reflects the realities of an actual implementation and is due to a number of factors. There is an interval between when the counter exceeds a trigger and when the page is moved or replicated, resulting in additional remote misses. Since migration and replication do not affect correctness on a CC-NUMA machine, the pager routine will ignore a page if its page-level lock is currently held by another processor. When a process is rescheduled to a different node, which has a local copy of a page, it may take a while before the page-table entry is changed to reflect this. Currently we only change mappings when a page or one of its replicas is processed after crossing the trigger threshold. The above factors affect the Splash workload most, since it has both rescheduling of processes and shared code and data segments.

The workloads were also run using sampled cache misses and TLB misses as amtracs. The results exactly mirrored what we observed with the traces, showing that sampling at 1:10 ratio was quite effective with the performance being similar to that without sampling. We had hoped that the flushing of TLBs, required when moving pages, might substantially change the effectiveness of TLB misses, making their results closer to that of cache misses. However, the observed behavior was very similar to that with the traces, TLB misses continued to be a somewhat inconsistent approximation of cache misses.

9.0 Conclusions and Summary

Cache coherent shared memory (CC-NUMA) multiprocessors are becoming increasingly popular as compute servers and now constitute an important class of machines. The key factor that affects the performance of applications on these systems is memory locality, which is the focus of this report. To study the issues in improving memory locality we assembled a very realistic and important set of workloads, that included a Sybase database, single and multiprogrammed parallel graphics and supercomputer applications, engineering simulators including VCS, and software development tools. Using traces and perfect cache miss information we studied the potential improvements in memory stall time that are achievable in these workloads. We showed that there are some workloads that benefit from the replication of data (in addition to code) and that dynamically replicating only hot pages (as opposed to first touch) can reduce the memory overhead. 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, even a cache-miss sampling rate of 1 in 10 can give results closely matching those of full cache information. Therefore, sampling can be used to reduce the cost of collecting cache-miss information in hardware. We did a detailed analysis of our kernel-based implementation of the policy and discovered that the primary sources of overhead were the synchronization between processors and the flushing of TLBs. Running the workloads on a kernel implementation of the policy, our results showed that improving memory locality reduces the overall stall time and by reducing contention, the latency of misses too. This compounding effect is important, and the performance improvement was as much as 28% for the engineering workload on the CC-NUMA architecture and could be as high as 44% on the CC-NOW architecture.

Bibliography

[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.

[BSF89] W. Bolosky, M. Scott, and R. Fitzgerald. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19-31, December 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.

[CHR+95] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory System Performance of UNIX on CC-NUMA Multiprocessors. In A*CM SIGMETRICS '95*, pages 1-13, May 1995.

[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.

[Hol88] M. Holliday. Page table management in local/remote architectures. In ACM SIGARCH International Conference on Supercomputing, pages 1-8, July 1988.

[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.

[LaE91] R. P. LaRowe Jr. and C. S. Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel Distributed Computing*, 11(2):112-129, February 1991.

[LaE91] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319-363, November 1991.

[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.

[MOH96] Margaret Martonosi, David Ofelt, and Mark Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers (92 kB). To appear in *the proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems,* Philadelphia, 1996.

[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.

[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.

[WSH94] S. Woo, J. P. Singh, J. L. Hennesey. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 219-232, October 1994.