Resource Management Issues for Shared-Memory Multiprocessors

Ben Verghese

Technical Report: CSL-TR-98-753

March 1998

This work was supported in part by ARPA contract DABT63-94-C-0054.

Resource Management Issues For Shared-Memory Multiprocessors

Ben Verghese

Technical Report: CSL-TR-98-753
March 1998

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

Abstract

Shared-memory multiprocessors (SMPs) are attractive as general-purpose compute servers. On the software side, they present the same programming paradigm as uniprocessors, and they can run unmodified uniprocessor binaries. On the hardware side, the tight coupling of multiple processors, memory, and I/O provides enormous computing power in a single system, and enables the efficient sharing of these resources. As a compute server, this power can be exploited both by a collection of uniprocessor programs and by explicitly or automatically parallelized applications. This thesis addresses two important performance-related issues encountered in such systems, performance isolation and data locality. The solutions presented in this dissertation address these issues through careful resource management in the operating system.

Current shared-memory multiprocessor operating systems provide very few controls for sharing the resources of the system among the active tasks or users. This is a serious limitation for a compute server that is to be used for multiple tasks or by multiple users. The current unconstrained sharing scheme allows the load placed by one user or task to adversely affect the performance seen by another. We show that this lack of isolation is caused by the resource allocation scheme (or lack thereof) carried over from single-user workstations. Multi-user multiprocessor systems require more sophisticated resource management, and we propose “performance isolation”, a new resource management scheme for such systems.

Performance isolation has the following desirable features. It allows the resources on the system to be shared by different tasks or users. The resources can be flexibly partitioned to reflect different criteria, e.g. the share of the cost of the system paid by different groups or the importance of different tasks or users. Performance isolation guarantees the task or user its share of the machine regardless of the load placed on the system by other users, leading to isolation of tasks in a heavily-loaded system. Performance isolation is also flexible enough to allow a task or user to utilize any extra resources that may be idle in the system, leading to better performance in a lightly-loaded system. We implement performance isolation in the operating system for three important system resources: CPU time, memory, and disk bandwidth. Running a number of workloads we show that it is successful at providing workstation-like isolation and latency under heavy load, SMP-like latency under light load, and SMP-like throughput under all conditions.

Most currently available shared-memory systems have a bus-based architecture, with the processors using a common bus to access main memory. This architecture provides an uniform access time to memory, independent of the processor and memory location accessed. For performance and scalability reasons the architecture of shared-memory machines is evolving to one with multiple nodes, each with one or more processors and a portion of the global shared memory (local memory). These nodes are interconnected with a fast and scalable network. In this architecture access to memory in another node (remote memory) is significantly slower than access to memory in the local node. Such systems are called CC-NUMA systems — cache-coherent non-uniform memory-access architecture.

In CC-NUMA systems good data locality is an important performance issue. Ideally, the data accessed by a process should be allocated in memory local to the processor running the process. However, such static allocation is not possible with the dynamic workloads found on compute servers. For fairness and efficiency, a compute server needs to move processes between processors to maintain load-balance. If a process is moved away from its data it will have to make accesses to remote memory, resulting in poor data locality and a large memory stall time.

In this dissertation, we carefully study the problem of automatically providing good data locality to applications. Our solution to this problem is to provide the kernel functionality to efficiently migrate and replicate pages in response to changes in the memory-access patterns of applications. We use cache-miss counting to detect the changing memory-access patterns of dynamic workloads, and use this information to drive the decision to migrate and replicate pages in the virtual memory system. We show that our OS-based solution is very effective in maintaining good data locality in CC-NUMA machines, and is able to greatly improve application performance.

Key Words and Phrases: SMP, resource allocation, performance isolation, fairness, resource sharing, data locality, CC-NUMA, page migration, page replication.

Copyright © 1998

Ben Verghese

Acknowledgments

My thesis work would not have been possible without the help, support, and encouragement of a number of people.

First, I would like to thank my advisors, Anoop Gupta and Mendel Rosenblum. Anoop's unflagging enthusiasm and demands for perfection greatly improved the quality of this thesis. Mendel's amazing system-building skills and his development of SimOS were crucial to the work done in this thesis. This thesis owes a lot to their inspiration and guidance.

I would also like to thank John Gill for graciously agreeing to be the chairman of my orals committee and for reading my thesis carefully, and John Hennessy for making time to be on my orals committee.

I am extremely grateful to my whole research group — Robert Bosch, Ed Bugnion, John Chapin, Scott Devine, Kinshuk Govil, Steve Herrod, Dan Teodosiu, Emmett Witchel — for all their encouragement and help, especially through those endless paper-deadline nights. This thesis would have been considerably harder if the whole group had not developed a superb simulator (SimOS) and had not indulged in many hours of crazy soccer, both indoors and outdoors.

I would particularly like to acknowledge Scott Devine, my co-conspirator in all the “Data Locality” work.

A number of people on the FLASH hardware team were extremely helpful, especially Joel Baxter, Mark Heinrich, Dave Ofelt, and Dave Nakahira who patiently answered my many questions about the MAGIC chip and other hardware-related issues.

I would like to thank my parents for their love and support. Their faith in my ability to succeed gave me the confidence to embark on this adventure. Finally my wife Preeti, who having been through this experience herself, was always there for me, celebrating my successes and helping me to overcome my failures. I thank her for her love.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The last thirty years of mainstream computing have seen significant changes in the basic computing paradigm in response to the evolution of computing technology and the concurrent changes in the requirements of users and applications. In the early days, the computing paradigm was essentially a centralized one, mostly mainframe systems that were shared by a large number of users. A centralized model was appropriate at that time because computing resources were scarce and expensive, and efficient utilization of these expensive resources was the primary concern.

In response to a variety of factors, this centralized computing paradigm gradually changed to a more distributed computing paradigm; personal workstations connected over a network. The main reasons for this change were:

- The cost of computing resources reduced dramatically. The first cracks appeared with minicomputers that were cheaper to build, resulting in some degree of decentralization. The advent of the microprocessor speeded up the process of change. The microprocessor enabled manufacturers to build relatively inexpensive personal workstations that rivalled the minicomputer in processing speed.
- The graphical display directly connected to the personal workstation replaced the clunky terminal. This change gave users a more sophisticated user interface, enabling more complex interactions between the user and the computer.

- The personal workstation gave the user significantly better control over the performance of applications running on it. In the centralized model in some cases, system overload conditions caused considerable user frustration.

More recently the computing paradigm is gradually shifting back to a more centralized model. The widespread adoption of computers in the workplace has exposed inefficiencies in the highly distributed model of computing. Two important factors driving this shift are:

- The cost of computing resources is no longer a large fraction of total cost of ownership of a desktop computer. A recent Gartner Group study [Gar94] showed that only about 27% of the costs of a network of computers were technology costs, i.e. for the hardware and software. The rest were personnel costs for maintaining this distributed computing environment, i.e. administration, troubleshooting, installation, support, and training.
- Second, a completely distributed environment results in the duplication of data and the fragmentation of resources. Every user must have enough resources on their system to accommodate the largest possible application they may want to run. For example, a single task requiring more memory than is available on a workstation cannot be run efficiently even though there is plenty of idle memory on other workstations in the network.

The client-server computing paradigm has emerged to address these concerns. The clients are simple personal computers with a good display capability and only limited computing and storage resources; moving more in the direction of “thin clients” in the future. These clients maintain very little permanent state, and so do not require much by way of system administration. The bulk of the actual computing resources is centralized in powerful and efficient servers. Only these servers, much fewer in number than the clients, have to be administered carefully. Examples of these servers are file servers, web servers, and database servers for centralizing information retrieval and storage, and also general-purpose compute servers that are able to efficiently run large tasks for users; ones that they would not be able to run on the clients or on personal workstations.

Client-server computing has generated an interest in servers. We will first discuss why shared-memory multiprocessors are well suited to assume the function of servers (Section 1.1). Next, we will describe the current and evolving architecture of shared-memory multiprocessors (Section 1.2). In Section 1.3, we examine the new demands being placed on the operating systems for these machines when they are used as servers. Specifically, we identify two important resource management issues that we explore in this thesis — “Performance Isolation” (Section 1.4) and “Data Locality” (Section 1.5). In Section 1.4 and Section 1.5, we will also present a summary of the results and the contributions of this thesis for each of these two operating systems issues. Finally we describe the organization of the remainder of this thesis in Section 1.6.

1.1 Shared-Memory Multiprocessors as Compute Servers

Shared-memory multiprocessors are now being widely used as large servers. There are two important reasons why these machines are being selected for this purpose. First, shared-memory systems aggregate a large collection of computing resource — multiple processors, large amounts of memory, and I/O — in a tightly-coupled system. Second, they provide a familiar view of memory, a single cache-coherent physical address space independent of the processor making the access and of the memory location being accessed. This hardware cache-coherence provides transparent low latency access to memory from the processors.

These two features are attractive for a number of reasons:

- The entire set of legacy applications available for the particular architecture can run on this server without any modifications, including sequential programs that run on an equivalent uniprocessor system.
- The single physical address space paradigm is also familiar to programmers, the one that they are used to with uniprocessor systems. This makes it easier to speed up large programs by parallelizing them and using the multiple processors available on these systems.
- These systems can be exploited in different ways by different types of applications. Sequential applications can now process larger data sets that could not fit in the memory of any single workstation [VeR97]. Parallel applications can use the multiple processors to achieve significant speedups. I/O bandwidth limited applications can exploit parallel I/O to a common buffer pool.
- Efficient utilization of the total computing resources is possible through fine-grain sharing. The tight coupling gives the OS great flexibility to schedule multiple applications.

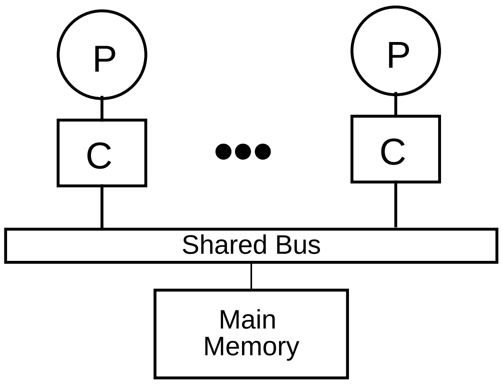

FIGURE 1. Architecture of bus-based shared-memory multiprocessors. Most current shared-memory multiprocessors have a bus-based architecture, with multiple processors (P), with their caches (C), connected to main memory through a shared bus. This type of architecture is called UMA (uniform memory architecture) because any processors can access any memory location with the same latency.

1.2 The Architecture of Shared-Memory Multiprocessors

Most current shared-memory multiprocessors use hardware-support to provide a cache-coherent single physical address space across the whole system. This architecture guarantees two important properties for all memory accesses. First, the single physical address space property guarantees that any memory location can be directly addressed by any processor, and can be read or modified through load and store instructions. Second, the cache-coherence property guarantees that the value read or modified is the most current value corresponding to the memory location addressed; the cache-coherence hardware finds the most current value, either from memory if it is clean, or from one of the caches in the system if it is dirty.

Most current cache-coherent shared-memory multiprocessors are small (4 - 8 PEs), and are designed around a shared bus as shown in Figure 1. This shared bus connects all the CPUs to the main memory of the system. These bus-based systems are called *uniform memory access* (UMA) systems because any processor can access any memory location with the same latency. Cache coherence is implemented by processors snooping memory requests on the shared bus, and responding appropriately if they have a copy of the requested line in their cache [PaH96]. However, this bus-based architecture does not scale to larger systems because contention for the bus increases as processors are added to the system. The shared memory bus becomes the bottleneck in the system, and the latency of all memory accesses increases.

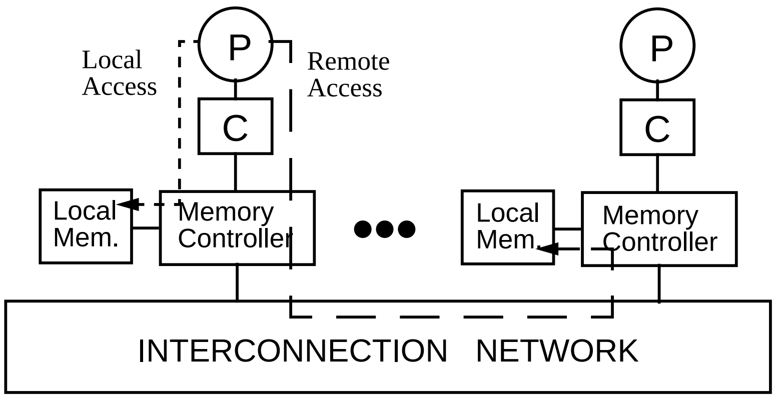

FIGURE 2. Architecture of CC-NUMA shared-memory multiprocessors. The figure shows the CC-NUMA (cache-coherent non-uniform access) architecture for scalable shared-memory multiprocessors. This architecture allows fast access to data in local memory and slower access to data in a remote memory. The dashed lines show sample paths taken for local and remote memory access. Each node has a specialized memory controller, and the memory controllers in all the nodes co-operate using directory techniques to maintain cache-coherence across the system.

The dominant architectural solution to this scalability problem is the distributed shared-memory architecture also known as the *cache-coherent non-uniform memory-access architecture* (CC-NUMA) [PaH96]. In this architecture the machine is divided into a number of nodes as shown in Figure 1. Each node consists of one or more processors and a part of main memory. These nodes are connected using scalable interconnect technology, and cache coherence is provided by a special memory controller using directory techniques. As processors get faster and their architecture gets more sophisticated, the shared bus will be a bottleneck even for smaller multiprocessors. Therefore, the CC-NUMA architecture is likely to be adopted even for smaller systems. These CC-NUMA machines were first designed in academia, Stanford DASH [LLG+90] and MIT Alewife [ABC+91]. A number of these machines have recently become available commercially, such as the Sequent STiNG [LoC96], Convex Exemplar [BrA97], Data General NumaLiine and SGI ORIGIN 2000 [LaL97].

CC-NUMA systems have different memory-access latencies depending on the location of memory. They provide fast access to data that is in memory local to the node. The latency for access to data in remote memory is considerably larger. The local access can be satisfied by the local memory controller without communicating with any other part of the system. The remote access takes longer because the local memory controller has to communicate with other nodes to satisfy the request as shown in Figure 1. The ratio of local to remote access latency is a function of the memory-controller architecture, the directory protocol used, and the interconnection network. Published numbers for the ratio of local to remote access latency for some systems include 1:3 for the Stanford DASH machine [LLG+90], 1:4 expected for the Stanford FLASH machine [KOH+94], 1:2 for small configurations and 1:3 for large configurations for the SGI ORIGIN system [LaL97], 1:8 for the Sequent STiNG system. These numbers are for latencies without any contention on an idle system. These latencies can increase significantly with contention. The disparity in the access time for local and remote memory makes data locality an important issue on these systems.

1.3 Challenges for the Operating System

Typical compute server workloads are quite different from large stand-alone parallel applications, which were an important initial motivation for the design of scalable shared-memory multiprocessor systems. Today, compute-server workloads consist of multiple applications with potentially widely differing characteristics. They may be sequential or parallel, long-lived or short-lived, memory-intensive, CPU-intensive or I/O intensive. These workloads are multiprogrammed and very dynamic, with the applications constantly entering and leaving. These multiprocessors also represent a large aggregation of computing resources that will potentially be used for multiple tasks and shared by multiple users or projects. Some examples of workloads running on multiprocessor compute servers include:

- A database server (with multiple threads) replying to queries from many clients.
- A web server and a video-streaming server providing information service to a whole building.
- Many long running jobs from multiple users on a machine, where each user owns a fraction of the machine.
- Multiple parallel scientific applications in a multiprogrammed workload.

Such environments require intervention from the operating system to carefully allocate resources to different tasks, making trade-offs where required. These trade-offs are generally between the performance of individual applications or users and system-wide performance qualities such as fairness, efficiency and throughput. The usage of computing resources, such as CPU time, memory, disk bandwidth and I/O bandwidth, by different applications needs to be managed on a continuous basis. For example in the workload above with a web server and a video server, it is important that the OS carefully manage the usage of disk and network bandwidth along with CPU time and memory. Given the dynamic and multi-user nature of these workloads, static allocation and partitioning of resources is either not possible or will lead to poor response time for individuals, and poor throughput for the whole system.

This thesis identifies, analyzes, and proposes solutions for two important performance issues for multiprocessor compute servers. Both involve interesting trade-offs between individual applications and user performance on one hand, and overall system issues such as fairness, efficiency and load-balance on the other. The two issues we will address are performance isolation and data locality.

- **Performance Isolation:** Can the operating system provide performance guarantees to groupings of processes representing individual users or tasks in a multiprogrammed environment, and at the same time maintain the fine-grain sharing of machine resources that is important for good throughput.
- **Data locality:** Can the operating system automatically provide good data locality on CC-NUMA systems (i.e. ensure that most of the cache misses are to local memory), even though applications are being moved around among the processors to maintain load balance in response to the dynamic nature of the workload.

We discuss each of these two issues in greater detail in the next two sub-sections.

1.4 The Performance Isolation Problem

In this section, we will first motivate and describe the performance isolation problem and then present the results and contributions of this thesis.

1.4.1 Problem Description

Current operating systems present the user with two fairly sub-optimal choices for a compute-server solution. As an illustration, let us take the example of end-users accessing information over the web. For simplicity we consider a content provider (CP) providing information resources to a number of high-end customers. The customers needs are diverse, and the CP provides web page service, video service, SQL-based data access, etc.

The CP has basically two possible server solutions available today. The first solution is to provide a separate machine for each service. Each machine must be configured with enough resources to

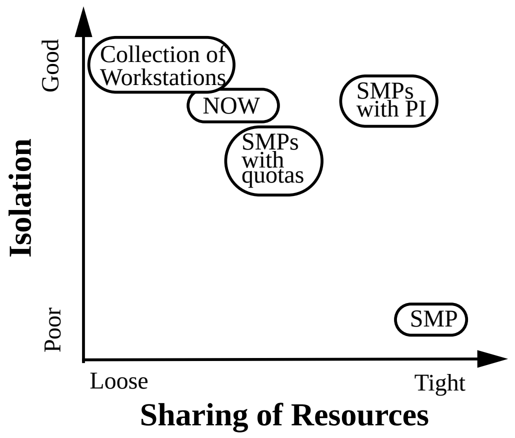

FIGURE 3. Trade-offs when clustering computing resources. We consider the effectiveness of different ways of clustering computing resources, as collections of workstations or as shared-memory multiprocessors with equivalent configurations (processors, memory, and I/O resources).The two axes here are sharing of resources and isolation of performance. A network of workstations (NOW) provides good isolation of performance, but poor sharing of resources. The situation is the opposite for current SMPs. SMPs with fixed quotas fall somewhere in between. With Performance Isolation on shared-memory multiprocessors (SMPs with PI), we get most of the benefit on both axes, workstation response times under heavy load, and SMP response times under light load.

handle the peak requirements of the service. This solution leads to an over-commitment of resources, and is inefficient because a large fraction of the resources are potentially idle at any time. The consequence will be higher hardware and maintenance costs for the CP (and the end-user).

The second solution is to have the different services share a multiprocessor system. The multiprocessor solution provides more efficient utilization of resources, but could lead to interference between the services. For example, a flurry of queries to the web server could cause enough disk activity and force the video server to miss deadlines because of delayed disk accesses. Similarly, web accesses may interfere with video data at the network interface. In a different scenario, a complex query on the SQL server may be executed in parallel and starve the web server and video server of CPU time. There can be no assurance of predictable service and this choice could lead to a decrease in customer satisfaction. The ideal solution should provide both cost efficiency and customer satisfaction (performance).

We explore this trade-off in Figure 3. The two axes on the graph represent two desirable qualities for compute servers. The x-axis represents the ability of the system to share computing resources between processes. Shared-memory systems with their tight coupling are able to efficiently share resources at a fine grain. The y-axis is isolation, i.e., how well can different tasks or groups of tasks control their expected performance and not be affected by the load placed on the system by others. Here the more distributed solution, in the form of individual workstations or networks of workstations (NOW) [ACP+94], fares significantly better than current shared-memory multiprocessors. The workstation solution has implicit isolation because jobs are run on separate workstations.

Current operating systems have little support for controlling the allocation of resources to groups of processes, or for providing fairness by any abstraction other than individual processes. The existing controls for fairness regulate only access to the CPU by a process. There are few controls for allocating other resources that can affect performance, such as memory and I/O bandwidth. A compute-server may be shared by multiple users, groups of users, or by multiple important tasks, and therefore needs more sophisticated controls for sharing resources.

The lack of control over resource sharing leads to poor isolation in current shared-memory multiprocessors. As a result of unconstrained sharing, the performance seen by an individual user in a multi-user environment is dependent on the load placed on the system by other users. Users have no reasonable guarantee of minimum performance even if they are using much less than the share of the resources they are entitled to. A single user or process can easily load the system unfairly, and tie up a large fraction of the resources. Examples of activities that can lead to unfair load are a user starting many processes, a process touching a large number of pages resulting in a huge working set, or a process making a large number of accesses to a disk. The lack of good resource control mechanisms has generated the perception that shared-memory systems cannot isolate a user from interference, and resulted in many detractors of these systems.

We use an example to illustrate the extent of the interference between jobs caused by the lack of isolation on a shared-memory multiprocessor. Consider a user running a job on a uniprocessor workstation. The job in this case is a parallel make with two simultaneously active compiles. This user now shares in buying a multiprocessor with three other users. This is a four-way multiprocessor that has four times the memory and disks as the uniprocessor. User A expects better performance than on the uniprocessor when the system is idle. Also, user A’s share of the computing resources of the multiprocessor is equivalent to that of the uniprocessor system. Therefore, user A may expect that the observed performance on the multiprocessor should at worst be comparable to the uniprocessor performance when the other users are also active.

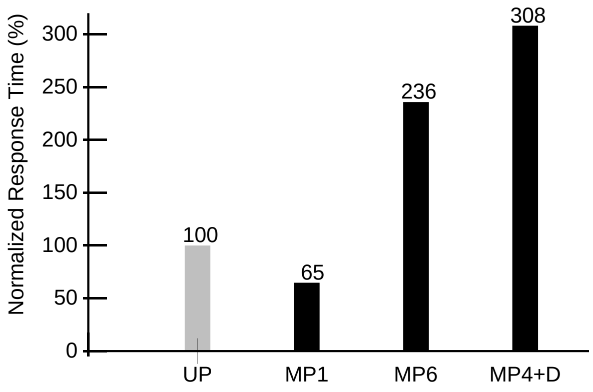

FIGURE 4. Lack of isolation in MP compute servers. The figure shows the response time for a Pmake job on an MP for three different system workloads relative to the uniprocessor case (UP). MP1 is just the single Pmake. MP6 is six simultaneous Pmake jobs accessing different disks. MP4+D is 4 simultaneous Pmake jobs, with one large file copy or core dump happening to the disk of one of the Pmakes.

Figure 4 shows the performance[1] of user A's pmake job on the uniprocessor and the multiprocessor, normalized to the uniprocessor case (UP), under various system load conditions. In the first multiprocessor case (MP1), user A is the only one using the system. User A's job is able to finish quicker (35% improvement) by utilizing the idle processors and additional memory. More available resources under low-load conditions is the primary attraction of moving to a shared-memory multiprocessor for compute-server type workloads. Idle resources are easily exploitable unlike a workstation solution.

We now consider two interference scenarios that may be familiar to users of shared multiprocessors. When other users increase the load on the system, the performance of User A's job steadily gets worse even though its resource requirements do not change. MP6 represents the first such scenario, where the other three users have started five similar jobs on the system for a total of six jobs. If the system was fair, user A who is only running one job might have expected to see a response time approximately that of the uniprocessor workstation. Response time for user A's job has increased to **236%** of the uniprocessor case.

The other scenario is four users each running a job, and one of the other users causes a core dump to the disk on which the sources for user A's compile job reside (MP4+D). In this case, user A

1. The uniprocessor system models a MIPS R4000 based workstation and the multiprocessor corresponds to a similar MIPS R4000 based CHALLENGE multiprocessor from Silicon Graphics, both running the IRIX5.3 kernel. The results shown here are obtained using the SimOS simulation system, and the specific architectural parameters of the systems will be defined in Section 3.2.

might have expected to see a slight increase in response time over that of the uniprocessor case because of sharing of disk bandwidth. Instead, the response time for user A's job in this case is over **three** times what it was on the uniprocessor due to the writes caused by the core dump competing for disk bandwidth with the pmake. These examples clearly show that the performance seen by an individual user on these systems is completely dependent on the background system load and the activities of other users; clearly enough incentive to retreat to individual workstations.

The "Network of Workstations" solution for aggregating resources is not the appropriate model. These systems provide good isolation and response time for an individual's jobs at the cost of overall throughput. There is a much higher overhead for sharing resources because it is a loosely coupled system. Therefore, fine-grain sharing is difficult, and idle resources can only be allocated at a very coarse granularity.

One possible solution to provide isolation on shared-memory systems is to enforce fixed quotas. However this solution is too static, and would not allow the sharing of idle resources. Fixed quotas will significantly reduce the throughput and response time seen on these systems under light-load conditions as we will show later.

The performance isolation problem for the operating system is to allow users to easily utilize idle resources on a lightly-loaded system, while at the same time isolating the performance of a user from the load placed on the system by other users. This can be translated into getting the best of what workstations and SMPs currently provide; the efficient sharing and overall throughput of the SMP and the isolation of the workstation.

1.4.2 Results and Contributions

- This thesis shows that the lack of predictable performance for tasks is not inherent to shared-memory multiprocessors. It is an artifact of the limited CPU-centric resource allocation scheme used by current SMP operating systems, as inherited from single-user workstation operating systems.
- To solve this problem we propose a new comprehensive resource allocation scheme for shared-memory multiprocessors— Performance Isolation. Performance isolation is capable of dealing with any entity, not just a single process, such as tasks, users, or groups of users, and allows the

machine resources to be flexibly partitioned among such entities. It also encompasses all resources that can affect performance, such as memory and I/O bandwidth, and not just CPU time. The objective of this model is to provide groups of processes with predictable performance corresponding to their assigned share of resources independent of system load. This performance is to be achieved without compromising the fine-grain sharing of idle resources and efficient resource utilization provided by shared-memory multiprocessors.

- We outline the general framework for providing performance isolation based on a new kernel abstraction called the SPU (Software Performance Unit). The SPU encompasses facilities for monitoring resources used by groups of processes, limiting resource usage, and allowing careful sharing of idle resources. The actual implementation of the model for CPU time, memory, and disk bandwidth in the IRIX 5.3 kernel from Silicon Graphics is discussed in detail, providing working solutions to problems encountered.
- Our solution for providing fairness for disk bandwidth is quite novel. The solution balances fairness and disk-head position when scheduling disk requests. An implementation of fairness that does not take into account disk-head position will result in lower overall throughput because of higher seek latencies.
- By running specific workloads, we highlight how performance isolation achieves its objectives for individual resources and for applications in general. Overall we show that the performance isolation model is able to provide the best of individual workstation and SMP performance, under different conditions of individual user requirements and system load.

1.5 The Data Locality Problem

In this section, we will first motivate and describe the data locality problem and then present the results and contributions of this thesis.

1.5.1 Problem Description

The higher latency for satisfying a cache miss from remote memory in CC-NUMA systems makes it essential that an application find most of its important data in local memory. On a cache miss, the processor is potentially stalled for the duration of the latency to fetch the required data from memory. As processors get faster applications spend a larger fraction of their execution time stalled for memory. A large number of cache misses to remote memory could severely impact application performance because of the larger resulting memory-stall time. The higher remote-access latency is important even for newer processors that can have multiple outstanding misses to memory, and so overlap and hide some of the cache-miss latency. The number of outstanding misses in these processors is usually a small number, and it is very hard to hide the long latency to access remote memory.

There are many reasons why a process may not find its data in local memory. First, the user or application writer may not care to do data placement. Second, even if they do want to, they may not be able to do so for a number of reasons.

- The static solution of locking processes to processors, and placing data in the appropriate local memory will not work for compute server workloads. In a dynamic environment, where applications enter and leave the system, for load-balance and fairness reasons the operating system will eventually need to relocate a process to a different node. Once a process changes nodes, the data that was initially placed in local memory will now be remote. In such cases the process itself cannot do much to improve data locality because it is not aware of remote memory, and it is not informed when it is moved.
- Even if a process were to stay relatively fixed to a processor for a period of time, processes with a large memory footprint are likely to find that all their data does not fit in local memory. As their access patterns change with time, many of the misses will go to remote memory.
- Even for stand-alone parallel applications, the programmer may not always be able to do static placement because the access patterns may be data dependent or may not be captured through static placement.

The data locality problem for the OS on CC-NUMA systems is to ensure that an application finds most of its important data in local memory while also maintaining load-balance and fairness across the entire workload. In our experiments running compute-server workloads with NUMA-unaware operating systems, the data locality was very poor; 12% of misses to local memory in an eight-node system, which is no better than random (details in Section 6.2).Therefore, it is important that the current working set of the process, currently running on the local processor, be moved to local memory as and when this changes. The overhead for providing good data locality should also be kept low so that the application sees an overall improvement in performance.

1.5.2 Results and Contributions

The solution to the data locality problem for distributed shared-memory (or CC-NUMA) multiprocessors has two parts. Process scheduling decides on which processors different processes are scheduled, and this determines where cache misses originate. The virtual memory system (VM) decides where the data is located in physical memory, and this determines where cache misses are satisfied. Both process scheduling and the VM need to work in conjunction to solve the data-locality problem.

Earlier work by us for process scheduling on CC-NUMA multiprocessors [CDV+94] showed that:

- For sequential applications, *affinity* scheduling is very effective in reducing the memory-stall time of applications over regular UNIX scheduling. Affinity scheduling schedules processes to improve reuse of data cached at various points in the memory hierarchy. More specifically, cache affinity is affinity acquired for a processor because of state built up in its caches. Cluster affinity is affinity for a group of processors in a clustered environment like the DASH system [LLJ+92] because of state built up in other caches in the cluster.
- Importantly, affinity is also required to keep a process from moving away from the memory that holds its data. Memory affinity is crucial to see benefits from migration and replication of data. The overhead to bring data to local memory is paid up front, but the benefit from data locality will increase the longer the process stays local to this memory.
- Scheduling schemes that are aware of parallel applications can provide modest benefits over standard UNIX scheduling. This is true for both time-partitioned schemes like gang scheduling[1] [Ous82] and space-partitioned schemes like process control[2] [ABL+91] [TuG91]. The improvement is dependent on the relative importance of data distribution and the non-linearity of the speedup curve, the former favoring gang scheduling and the latter favoring process control.
- In multiprogrammed parallel workloads the data placement assumptions of the programmer

1. In gang scheduling multiple parallel applications share a machine through time slicing and all processes of a parallel application are scheduled at the same time on different processors.
2. In process control scheduling each parallel application is allocated a subset of the processors, and the number of processes active in the parallel application changes to match the number of processors allocated to the application.

break down. Even gang-scheduled applications cannot assume a static environment because processes will be moved for load balancing purposes. OS support for automatic data locality becomes very important.

In this thesis we build on these results and focus on the virtual memory aspect of the data-locality problem. We assume affinity scheduling for sequential applications, and consider the impact on parallel applications that are gang-scheduled or support process control. We explore the use of migration and replication of data to improve the data locality of applications. The results from our work are:

- We design a generalized policy framework for page migration and replication based on a carefully analysis of the data-locality problem in terms of costs and benefits. The policy framework incorporates features to maximize benefit and minimize cost because data locality for CC-NUMA multiprocessors is only an optimization and is not needed for correctness.
- In order to maximize benefits, only the pages taking the most cache misses should be considered as candidates for migration and replication. This is a key addition to earlier policies developed for non-cache-coherent shared memory machines. To find the pages taking the most cache misses we recommend counting cache misses in the memory controller. Our analysis shows that counting TLB misses, a potential software-only scheme, is not reliable because the correlation between TLB misses and cache misses is not consistent for all applications. We also find that sampling of cache misses can be used to reduce the cost of cache-miss counting without compromising information quality.
- We implement page replication and migration in the IRIX5.3 kernel from Silicon Graphics. Needed kernel changes fall into two categories, new functionality and performance optimizations. New functionality not found in current kernels includes support for tracking replicas of pages, efficient back mappings from physical pages to page table entries, and implementation of the policy. Performance optimizations include finer-grain synchronization in the VM system, reducing the cost of taking interrupts, and reducing the cost of TLB flushing. TLB flushing turns out to be the cause of significant kernel overhead.
- Our policy and implementation of migration and replication is able to greatly improve the locality of application cache misses; from almost random to greater than 80% in some cases. This greatly improved memory locality and low kernel overheads result in improved applica-

tion performance, reducing the execution time as much as 50% in some workloads. The implementation is also robust, and does not reduce the performance for applications where the access patterns preclude improvement of data locality.

- Another benefit that is not immediately obvious, but equally important, is the system-wide benefit from the reduction in utilization of memory system components, such as the memory controller and the network, because of the improvement in locality. This results in a reduction in network queue lengths, reduction in controller occupancy, and a significant reduction in the observed latency of local misses, as much as 30% in our experiments.
- From exploring the policy space of possible solutions for data locality and the parameters in our policy, we find that dynamic policies are able to outperform even the best off-line static policy. Both migration and replication are required to get the total benefit in all cases, and each by itself is not sufficient. The trigger threshold, which decides how aggressive the policy should be in selecting busy pages, is the important parameter and is dependent on the ratio of local and remote misses and the cost of migrating or replicating a page.

1.6 Organization of the Thesis

The organization of the remainder of this thesis is as follows. In chapter 2 we analyze the reasons for the lack of isolation in current shared-memory multiprocessors. We propose Performance Isolation, a new method for resource management to solve this problem. We then develop a general framework for implementing the performance isolation model and describe the details of our implementation for CPU time, memory and disk bandwidth in the IRIX5.3 kernel from Silicon Graphics.

In chapter 3 we describe the SimOS simulation environment that we use to perform our experimental runs, and discuss the factors that prompted us to use a simulation system for our experiments. We also describe the target machine architecture which forms the basis of our experiments, and justify the choice of the SGI IRIX 5.3 operating system that forms the basis for implementing our various schemes.

In chapter 4 we present the experimental results from running workloads using our implementation of the performance isolation model. These results are compared with results using an unmodified SMP operating system and with a version that uses fixed quotas to provide isolation. Different workloads are used to highlight the effectiveness of performance isolation for each of the resource types, CPU time, memory, and disk bandwidth.

In chapter 5 we discuss the data-locality problem, and provide a general policy framework for implementing automatic migration and replication of data. We then describe the details of the kernel implementation of the policy, and discuss the mechanisms for counting cache misses to drive the policy.

In chapter 6 we present the experimental results of running workloads with the automatic migration and replication of data, and show the benefits possible by comparing it to that of the base operating system without this functionality. We carefully analyze and describe the overheads introduced in the implementation. Finally we discuss a number of issues relating to possible policy variations.

Chapter 7 surveys related work in both the data locality and performance isolation areas, and chapter 8 presents our conclusions.

Chapter 2
Performance Isolation

Shared-memory multiprocessors are well suited to be compute servers for many reasons. As compute servers, these machines are often used in a shared environment by multiple users, by groups of users or by different simultaneous tasks. However, such systems are currently unable to isolate the performance of a process or a group of processes from changes in system load. Users have no reasonable guarantee of minimum performance for their applications even if they are using much less than the share of the resources they are entitled to on the system. One or more users or processes can easily load the system unfairly and tie up a large fraction of the resources, as was demonstrated in Figure 4. This behavior has generated the perception that shared-memory systems are inherently unable to isolate a user from interference, and has resulted in many detractors of these systems.

Our assertion is that this lack of isolation is not inherent to the fine-grain sharing provided by SMP operating systems on these machines, rather the observed degradation of performance is an artifact of the existing CPU-centric resource allocation method. This CPU-centric method might have been appropriate on workstations, where access to the CPU implied access to all resources on the system. Workstations usually have only a single user on the system who can control the number and type of active processes to achieve certain expectations of performance, albeit in an ad-hoc manner. Another potential shortcoming is that CPU allocation was done to maintain fairness on a per-process basis, but there were few controls for other resources that are also important in determining application performance, such as memory and I/O bandwidth. Memory allocation was usually not done explicitly with static quotas per process at best, and allocation of I/O bandwidth was nonexistent. Current operating systems also have little support for controlling the allocation of resources to groups of processes, or for providing fairness by any abstraction other than individual processes.

This resource allocation method is not appropriate for modern multiprocessors (or even uniprocessor compute servers). On a multiprocessor, control of access to a single CPU gives little control over the other resources. Multiple processes are active at the same time on the system, and without appropriate controls any one of them can consume arbitrary levels of CPU, memory and I/O bandwidth. Resource management needs to be done to provide fairness to higher-level logical entities, such as individual users, a group of users, or a group of processes that comprise a task, not just between competing processes. In the absence of such criteria when allocating resources, we get the unwanted behavior seen on most current systems, where a user or task can get a larger share of the system by simply starting more processes.

A multiprocessor is often a shared resource with implicit or explicit contracts between users or tasks on how to share the machine. For example, all users have equal priority and expect equal performance, or project A owns a third of the machine and project B owns two thirds. Unlike the ad hoc control exerted by the single user on a workstation, such contracts would be very difficult to implement in the absence of explicit mechanisms and policies for resource management.

One method to implement such contracts is to enforce fixed quotas per user or task for the different resources. However, this method would not allow the sharing of idle resources, and significantly reduce the throughput and response time seen on these systems under light-load conditions. This thesis presents “**Performance Isolation**”, a new resource management scheme for shared-memory multiprocessors. This scheme isolates processes belonging to a logical entity, such as a user, from the resource requirements of others, and preserves the inherent sharing ability of these machines. Performance isolation seeks to take advantage of the tight coupling and flexibility of shared-memory machines to provide both workstation-like latency under heavy load and SMP-like latency under light load, and SMP-like throughput all the time, getting the best of both worlds.

In Section 2.1 we elaborate on the performance isolation model, explaining the two facets to its implementation. The first is the metrics and mechanisms needed to provide isolation between groups of processes, and the second is the policies needed to enable careful sharing of resources between these groups. In Section 2.2 we discuss our implementation of this resource allocation model in the IRIX5.3 operating system from Silicon Graphics for CPU time, memory and I/O bandwidth resources. We also point out features of current SMP operating systems that make it difficult to provide isolation based on our experience with the implementation. In Chapter 4 we will present the results from running different workloads on our implementation of the performance isolation model.

2.1 The Performance Isolation Model

The performance isolation model for shared-memory multiprocessors essentially partitions the computational resources of the multiprocessor into multiple flexible units based on a previously configured contract for sharing the machine. From a performance and resource allocation viewpoint the multiprocessor now looks like a collection of smaller machines. At the heart of the model is a new kernel abstraction called the ***S**oftware **P**erformance **U**nit* (SPU), and each of these smaller machines is associated with an SPU. This is not a static permanent allocation of resources to SPUs as will become clear soon. SPUs can be created and destroyed dynamically, or could be suspended when it has no active processes and awakened at a later time.

The SPU abstraction has three parts:

1. The first is a criterion for assigning processes to an SPU. This decides which processes have access to the resources of the SPU. The performance of a process will be isolated from the resource requirements of any process that is not associated with its SPU. However, the SPU does not provide isolation between processes that are associated with the same SPU. The desired basis for the grouping of processes can vary greatly, and is dependent on the environment of the particular machine and the isolation goals. Some common possibilities are:

    - Individual processes — An SPU could be created for a web server.
    - Groups of processes representing a task — An SPU could be created for all the processes of a parallel database server or a parallel scientific application.
    - Processes belonging to a user — An SPU could be created for user foo and another for user bar.

- Processes belonging to a group of users — Separate SPUs could be created for the OS, compiler, and hardware groups.

2. The second is the specification of the share of system resources assigned to the SPU. We are primarily interested in the computing resources that directly affect user performance: CPU time, memory, and I/O bandwidth (disk, network, etc.). However, it would be possible to incorporate other resources if required. There are many possible ways of partitioning resources between SPUs, such as:

   - A fixed fraction of the machine — An SPU gets half of all the hardware resources of the machine.
   - A specified fraction of each resource — Two CPUs, 128 Mbytes of memory, and half the bandwidth to all disks and the network.

3. The third is a sharing policy. Resources can be lent to other SPUs, and revoked when needed again by the loaning SPU. The sharing policy decides when and to whom resources belonging to an SPU will be allocated when these resources are idle. There are many possible types of sharing policies, and the following is a nonexhaustive list:

   - Never give up any resources. This will approximate the case of each SPU being an entirely separate machine with its share of resources, or the machine being divided up with fixed quotas; there is no sharing.
   - Share all resources with everyone all the time, without consideration for whether the resources are idle or not. This approximates the behavior of current SMP systems.
   - A more interesting possibility is to share only idle resources with all or a subset of the SPUs that lack sufficient resources and could use the idle ones.
   - Hierarchical sharing. There is a sharing tree of SPUs, with sharing happening at lower levels of the tree before the higher-levels.
   - Priorities could be implemented through the sharing policy. For example SPU 1 is allocated all the resources on the system, and its sharing policy is to give SPUs 2 and 3 half of its idle resources each.
   - There could be a temporal element to the sharing policy, i.e., giving up fractions of the idle resources after different intervals.

The sharing policy of the SPU abstraction, as described above, can be set to customize the behavior of the system as seen by the users. The performance isolation model, which we discuss in the rest of this thesis, will use the SPU abstraction with a specific sharing policy to achieve its goals. An SPU will share idle resources with any SPU that needs the idle resources. With this sharing policy, the performance isolation model should achieve the following two performance goals:

1. **Isolation**: If the resource requirements of an SPU are less than its allocated fraction of the machine, the SPU should see no degradation in performance, regardless of the load placed on the system by others.

2. **Sharing**: If the resource requirements of an SPU exceeds its configured share of resources, the SPU should be able to easily use any idle resources to improve its response time and throughput.

There are two parts to the solution for implementing the SPU abstraction for providing performance isolation, corresponding to the two goals presented above. First, we need to provide isolation between SPUs so that processes in an SPU cannot interfere with the performance of processes in another. This is done by implementing mechanisms in the kernel to count resource usage by SPU and restrict SPU resource usage to allocated limits. These mechanisms will guarantee that an SPU will be able to utilize all the resources it is entitled to even when the system is heavily loaded.

Second, we need to implement the chosen sharing policy for SPUs. To implement the sharing policy we need to detect idle resources, reallocate these idle resources to SPUs that may need them, and revoke the resources when they are needed again by the loaning SPU. This allows sharing of idle resources when the system is lightly loaded, providing for better throughput and response time.

2.1.1 Providing Isolation

In order to provide isolation between SPUs two new aspects of functionality were needed in the kernel not provided by current SMP systems. First, the utilization of resources by individual SPUs needed to be tracked. For example, every time a memory page is allocated, the kernel needs to know which SPU is getting the page and increment its page usage counter. Second, mechanisms are needed to limit the usage of resources by an SPU to allocated levels. For example, currently in the SGI IRIX operating system a request for a page of memory will fail only if there is no free memory in the system. For isolation a page request from a process may need to be denied if the SPU that owns the process has used its allocation of pages, even if there is still memory available in the system. Current SMP systems do not have the appropriate metrics to track short-term usage of all resources by processes or groups of processes, and cannot limit the usage of these resource by a specific process.

A particular problem area in providing isolation is accounting for resources that are actually shared by multiple SPUs, or that do not belong to any specific SPU. Examples of the former are pages of memory accessed simultaneously by multiple SPUs such as shared library pages or code, and delayed disk write requests that often contain dirty pages from multiple processes and multiple SPUs. Examples of the latter are kernel processes, such as the pager and swapper daemons and pages used by the kernel code and static data.

For this problem our current strategy has been to choose the simplest solutions that seem reasonable. More sophisticated solutions may easily be considered in the future, if we encounter instances where these proposed solutions clearly do not work. To address the above problem we introduce two default SPUs in the system: *kernel,* for kernel processes and memory; and *shared* for tracking resources used by multiple SPUs. The cost of memory pages that are referenced by multiple SPUs is counted in the shared SPU, and not explicitly allocated to any of the user SPUs. Memory pages other than those used by the kernel and shared SPUs are divided among user SPUs. Therefore, the cost of shared and kernel pages is effectively shared by all user SPUs. The cost of shared pages could be assigned more precisely if necessary, but this would incur a larger overhead. Shared disk writes get scheduled for service in the shared SPU. The cost of individual non-shared pages in these write requests is allocated to the appropriate user SPUs. The kernel SPU has unrestricted access to all resources.

2.1.2 Policies for Sharing

The second part of the performance isolation model is the careful sharing of idle resources between SPUs, based on the sharing policy feature of the SPU abstraction. Conceptually, each SPU maintains three resource levels to implement resource sharing. The first level is the amount of resources that the SPU is **entitled** to initially. This level is decided by the division of system resources based on the sharing contract for the system. The second level is the amount of resources that the SPU is **allowed** to use currently. The third level is the amount of resources currently **used** by the SPU.

Sharing is implemented by changing the allowed level for SPUs based on resource requirements and availability. In a system under load where all SPUs are utilizing their share of the resources, all three levels will be at about the same value for the SPUs. The used level will be equal to the entitled, indicating full utilization. The allowed level will be equal to the entitled because there are no idle resources to share. At some point in time the situation may change, one or more SPUs may go idle or be under utilized, and some other SPUs may be heavily loaded. For the under utilized SPUs, their used level will now be much less than their entitled level, indicating idle resources. The sharing policy can now transfer some of these idle resources from the under-utilized SPUs to the others by increasing the value of the allowed level for the latter. In the presence of sharing, the allowed level will be greater than the entitled level for the heavily-loaded SPUs. The situation may change again at a later point in time when the utilization of formerly idle SPUs increases, and they need their resources again. The sharing policy will now revoke resources from SPUs that had borrowed them by lowering their allowed level, potentially to the entitled level.

The isolation of performance of an SPU can be adversely affected if the sharing policy is not careful when transferring idle resources. Transferring all idle resources from underloaded SPUs to overloaded SPUs would result in the best system throughput. However, the resources that are currently idle in an SPU may be needed in the future. The base performance of an SPU will be adversely affected if the resources that are loaned are unavailable when they are needed.

The key factor in making the decision to transfer resources is the **revocation cost** for these resources when they are needed again by the loaning SPU. If the revocation cost were zero, then transferring all the resources would not be a problem as they could be instantly revoked when needed. However, most resources have a non-trivial revocation cost, and this cost plays a part in deciding when resources are transferred and how much of the idle resources are transferred. When making sharing decisions, the policy module needs to ensure that the cost of revocation does not adversely affect the performance of the loaning SPU and break isolation. The revocation cost and the details of sharing are explained for each resource in Section 2.2 below.

2.2 Kernel Implementation

In the previous section we discussed a framework for implementing performance isolation in terms of providing isolation and sharing between SPUs. We now describe the details of our implementation of the performance isolation in the IRIX5.3 kernel from Silicon Graphics. Most of the ideas in this implementation are not specific to IRIX, and would apply to other operating systems

Name	Unit of allocation	SPU metrics tracked	Isolation Mechanism	Sharing Policy
CPU	Time in milliseconds	CPUs *entitled* CPUs *idle* CPUs *loaned*	SPU ID of process and CPU must match while scheduling	SPU ID ignored if CPU is idle
Memory	Page (4K)	*entitled* count *allowed* count *used* count	Page granted only if *used* count < *allowed* count for SPU.	Increase *allowed* for SPU with memory pressure if *total idle* > *Reserve Threshold*
Disk BW	Sectors per second	Per disk: Sectors transferred & Pending requests	SPU skipped if *transferred* > (share of bandwidth + *BW difference threshold*)	Head-position scheduling of pending requests unless SPU is to be skipped

TABLE 1. Summary of the performance isolation implementation. For each resource we show the metrics that need to be kept per SPU, the mechanism to isolate the performance of an SPU and the basic sharing policy.

as well. The system resources included in the implementation are: CPU time, memory, and disk bandwidth (as an example of I/O bandwidth). Though we do not implement performance isolation for network bandwidth, the techniques we describe would apply to it as well. Our implementation is based on the assumption that all resources are to be divided equally among all the active SPUs, though it will be clear from the implementation that different shares can easily be supported.

For each resource we describe the metrics used to count usage, the mechanisms put in place to provide isolation, how these mechanisms differ from the ones currently in IRIX, and how the sharing policy enables sharing by reallocating idle resources. A high-level summary of the implementation for each resource is given in Table 1. For our implementation we picked reasonable policies and mechanisms that allow us to clearly demonstrate the effectiveness of performance isolation. Other mechanisms are also possible for each of the resources, and we will discuss them with related work in Chapter 7.

2.2.1 CPU Time

In IRIX, CPU time is allocated in time slices to processes — 30ms unless the process blocks before that for I/O. A priority-based scheduling scheme is used in which the priority of a process drops as it uses CPU time. A CPU normally picks the runnable process with the highest priority when scheduling a new process. This scheme maintains fairness for CPU time at a process level.

Isolation requires a mechanism to provide fairness at the SPU level, which usually includes more than one process. On a multiprocessor, CPU time can be allocated either through time multiplexing or space partitioning of the CPUs. We chose a hybrid approach. First, each SPU is allocated an integral number of CPUs using space partitioning, depending on its entitlement. If in the division, fractions of CPUs need to be allocated to SPUs, then time partitioning is used for the remaining CPUs with the share of time allocated to an SPU corresponding to the fraction of the CPU. The SPU to which a CPU is allocated is its home SPU. Kernel processes can run on any CPU. To provide isolation the normal priority-based scheduling behavior is modified by having CPUs select processes only from their home SPUs when scheduling, thus ensuring that an SPU will get its share of CPU resources, regardless of the load on the system. Between processes of the same SPU, the standard IRIX priority scheduling disciplines apply.

Our choice of a hybrid scheduling policy, favoring space partitioning, fits better with our model of partitioning the machine, and is based on the assumption that there will be fewer active SPUs than CPUs. There are two issues with our current choice of a scheduling policy for CPUs. If our assumption of more CPUs than SPUs does not hold it will be beneficial to switch to a more explicit time-partitioning policy. Also, parallel applications that use a space partitioning policy [ABL+91][TuG91] can be easily accommodated in our current scheme. Accommodating gang-scheduled [Ous82] parallel applications would require some modifications.

Sharing is implemented by relaxing the SPU ID restriction when a processor becomes idle. If an SPU is lightly loaded, one or more processors belonging to this SPU may be idle. If a processor cannot find a process from its home SPU, it is allowed to consider processes from other SPUs. Currently, the process with the highest priority is chosen. As a result, the SPU getting the idle processor is not explicitly chosen, but the process with the highest priority is likely to be one from a relatively heavily-loaded SPU. An SPU could be explicitly picked if the home SPU's sharing policy indicated a preference.

Processors that have been loaned to SPUs are tracked. If a process from the home SPU now becomes runnable, and there are no allocated processors in the home SPU available to run this process, then the processor loan is revoked. In our policy, the revocation of the CPU happens either at the next clock tick interrupt (every 10 milliseconds), or when the process voluntarily enters the kernel. Therefore the maximum revocation latency for a CPU is 10 milliseconds.

Another possibility would be to send an inter-CPU interrupt (IPI) to get the processor back sooner. This might be needed to provide response time performance isolation guarantees to interactive processes.

There are other hidden costs to reallocating CPUs, such as cache pollution. A more sophisticated implementation of the sharing policy could try to reduce these costs by preventing frequent reallocation of CPUs for sharing. One possibility is to not loan every idle CPU, i.e. keep one or more idle CPUs in reserve for the home SPU. Another possibility is to not loan a CPU if the algorithm detects that the allocation is being revoked frequently.

2.2.2 Memory

The IRIX5.3 kernel has very few controls for memory allocation. It has a configurable limit to the total virtual memory a process can allocate. It also tries to place a fuzzy limit on the size of actual physical memory that a process can use. The problem is that these limits are per-process, and cannot provide the strict isolation that our model requires. Being essentially fixed quotas per-process, they may actually inhibit sharing of idle resources in the system.

Isolation and sharing for physical memory closely follows the method outlined in Section 2.1.2, keeping three counts of pages for each SPU — entitled, allowed and used. The page allocation function in the kernel is augmented to record the SPU ID of the process requesting the page, and to keep a count of the pages used by each SPU. In addition to regular code and data pages, SPU memory usage also includes pages used indirectly in the kernel on behalf of an SPU, such as the file buffer cache and file meta-data. Memory pages are conceptually space-partitioned among the SPUs, and the *entitled* count represents the initial share of memory for an SPU. Isolation between SPUs is enforced by not allowing an SPU to use more pages than the *allowed* limit.

A particular problem is tracking and accounting for pages that might be accessed by multiple SPUs, as mentioned in Section 2.1.1. When a page is first accessed, it is marked with the SPU ID of the accessing process. On a subsequent access by a different SPU before the page is freed, the page will be marked as a shared page (SPU ID of the shared SPU). The SPU ID for the page is reset when the page is finally freed. The cost of these shared pages is assigned to the shared SPU. Similarly the cost of pages used by the kernel is assigned to the kernel SPU. Only the remaining pages are actually divided among the SPUs based on their entitled share of memory. The allocation of pages to SPUs is periodically updated to account for changes in the usage of the shared and kernel SPUs.

Changes are made to the paging and swapping functions to make them aware of per-SPU memory limits. A request for pages beyond the *allowed* limit of the SPU will fail, and the process will be descheduled pending the freeing of pages. The pager process wakes up when a SPU is low on memory, and selectively frees pages from such SPUs. The original paging mechanism in IRIX5.3 only triggers when the system as a whole is low on memory, and it reclaims memory from all processes. The swapping of processes is modified to take into account the memory situation of SPUs when picking processes to swap in or out.

Sharing of idle memory is implemented by changing the *allowed* limit for SPUs. The SPU page usage counts are checked periodically to find SPUs with idle pages and SPUs that are under memory pressure. The sharing policy redistributes the excess pages in the system to the SPUs that are low on memory by increasing their allowed limits. The memory re-allocation is temporary, and can be reset if the memory situation in the lending or borrowing SPUs changes.

Excess pages are calculated as the total idle pages in the system less a small number of pages that are kept free. The small number of free pages is called the **Reserve Threshold**. The Reserve Threshold is needed to hide the revocation cost for memory, which is the time to reclaim any pages that have been lent to other SPUs. The revocation cost for pages of memory can be high, especially if they are dirty, because the dirty data will need to be written to disk before the page can be given back. The Reserve Threshold reduces the chance of a loaning SPU incorrectly being denied a page temporarily. The Reserve Threshold is configurable, and we chose 8% of the total memory. This is the value that IRIX uses to decide if it is running low on memory.

2.2.3 Disk Bandwidth

IRIX5.3 schedules disk requests based only on the current head position of the disk using the standard C-SCAN algorithm [Teo72]. The only consideration in this algorithm is total throughput to the disk, and there are absolutely no controls for any sort of fairness. In the C-SCAN algorithm the outstanding disk requests are sorted by block number and serviced in order as the disk head sweeps from the first to the last sector on the disk. Therefore, the next request serviced is the one ahead of and closest to the current head position. When the head reaches the request closest to the end of the disk, it then goes back to the beginning and starts again. This technique reduces the disk-head seek component of latency and prevents starvation. The process requesting the disk operation is not a factor in the algorithm, and there is total lack of isolation between SPUs. The sectors of a single file are often laid out contiguously on the disk. Therefore a read or write to a large file (e.g. a core dump) could monopolize the disk, causing all requests of one SPU to be serviced before requests from other SPUs are scheduled.

To provide isolation we need to account for the disk bandwidth used by SPUs, and incorporate this information into the decision process for scheduling requests for the disk. We encountered a few difficulties in providing isolation for disk bandwidth. First, disk requests have variable sizes (one or more sectors), and breaking up requests into single sector operations would be inefficient. This implies that the granularity of allocation of bandwidth to SPUs will be in variable-size chunks. Therefore it is not enough to just count requests, rather the size of the request needs to be accounted for. Our metric for disk bandwidth is sectors transferred per second.

Second, the writes to disks are frequently done by system daemons that are freeing pages, such as bdflush (flushes file buffer cache data) and pdflush (flushes page frame data). Therefore, these write requests contain pages belonging to multiple SPUs. Our implementation schedules these shared write requests as part of the shared SPU, which is given the lowest priority. Once the shared write request is done, the appropriate user SPU is charged for the bandwidth used for each individual page.

Third, disk bandwidth is a rate, and as such measuring the instantaneous rate is not possible. Therefore it is approximated by counting the total sectors transferred and decaying this count periodically. The decay period is configurable, and we currently decay the count by half every 500 milliseconds. A finer grain decay of the count would better approximate an instantaneous rate, but would have a higher overhead to maintain. This count of sectors transferred represents the bandwidth used by each SPU, and is kept for each disk.

Disk requests can incur a considerable latency for the disk head to seek to the appropriate spot on the disk. Implementing strict isolation requires a round-robin-type scheduling of requests by SPU based on bandwidth shares of each of the SPUs. However, completely abandoning the current disk-head position based scheduling would result in poor throughput because of excessive delays caused by the extra seek time (see results in Section 4.6). Therefore, performance isolation employs a compromise that incorporates both disk-head position and a fairness criteria when making a scheduling decision.

In our policy, disk requests are scheduled based on the head position as long as all SPUs with active disk requests satisfy the fairness criteria. A SPU fails the fairness criteria if its bandwidth usage relative to its bandwidth share (current count of sectors/bandwidth share) exceeds the average value of all SPUs by a threshold (the **BW difference threshold**). Once an SPU fails the fairness criteria it is denied access to the disk until there are no more queued requests, or it once again passes the fairness criteria because other SPUs get their share of disk bandwidth. The fairness criteria is checked after each disk request. The choice of the BW difference threshold allows a tradeoff. Smaller values imply better isolation, with a choice of zero resulting in round-robin scheduling. Larger values imply smaller seek times, and a very large value results in the normal disk-head-position scheduling.

Sharing happens naturally because an SPU cannot fail the fairness criterion if no other SPU has active requests. The revocation cost for the disk bandwidth resource is the time to finish any currently outstanding request, and for the disk head to scan to the desired position. Therefore, if a disk is shared then an SPU with high disk utilization can affect the performance of another SPU using the same disk. However, we will show that performance isolation can provide fairness and considerably reduce the impact of such shared access.

2.2.4 Shared Kernel Resources

In addition to the physical resources discussed above, there are shared kernel structures that must be considered in the implementation of the performance isolation model. These shared structures may be accessed and updated from any processor, and contention for these resources is a potential source of problems. Examples of shared kernel structures are: the list of free physical pages kept by the VM; in-core copies of file-system inodes; and the scheduler’s global queue of processes. The severity of these problems scales with the number of processors and the kernel activity of the workload. However, most of these problems are not specific to performance isolation, and need to be addressed when designing scalable multiprocessor operating systems. We outline the sources of these problems and our solutions to some important ones we encountered.

Updates to shared structures need mutual exclusion, and this is provided in the kernel through spinlocks and semaphores. Excessive contention for these structures will lead to reduction in performance as compared to a smaller system. Contention for semaphores results in blocking of processes, and this increases the response time of the application. We describe below two specific semaphore problems that we encountered and fixed in our implementation of performance isolation. These two changes were required to provide performance isolation, but also improved the response time of the base IRIX system. The reduction in response time was as much as 20-30% on a four processor system for some workloads.

Mutual exclusion for access to file system inodes was provided through a semaphore per inode. There is considerable contention for the inode semaphore at the root and high levels of the file system, allowing interference between unrelated processes. This contention for the root inode has the potential to completely break performance isolation, as we observed on a four processor system for a filesystem-intensive workload like parallel make. The dominant operation on these high-level inodes is a lookup. The lookup is essentially a read-only operation which takes a directory inode and a string representing the next part of the path, and returns an inode that corresponds to the child. Our solution was to modify the inode lock to be a multiple-readers single-writer lock. Now multiple lookups could proceed in parallel, and the contention problem for the inode lock disappeared. In general, I/O to a shared file system seems to have the potential to break performance isolation, and care should be taken when designing such systems.

Another example is the page_insert_lock semaphore protecting the mapping from file vnode and offset to pages of physical memory. This is a single semaphore for the entire hash table of mappings, a much larger granularity than necessary. By splitting this semaphore using a hash function on the vnode/offset, we were able to reduce the contention on this semaphore greatly. There are other semaphores that we noted, but did not address because their effect on performance was not significant in our workloads.

In addition to straight contention, a high load SPU starved of resources and holding an important semaphore could block a process from a light load SPU. This could affect the ability of the kernel to provide isolation between SPUs. This problem is similar to the well-studied priority inversion problem, and the solution is similar to priority inheritance [SRL90]. The solution requires that a process blocking on a semaphore transfer its resources to the process holding the semaphore until the semaphore is released. Though we were aware of this as a potential problem, we did not actually implement the solution because the priority inversion problem did not have a significant impact on performance in any of our workloads.

The other kernel abstraction for mutual exclusion is the spinlock. Contention for spinlocks results in additional kernel time spent spinning on a location. An example of a spinlock with high contention is `memory_lock`, which protects many of the important shared VM structures in the kernel. We did not address the `memory_lock` problem, as it would require pervasive changes in the VM system. We will quantify the performance impact of `memory_lock` contention in Section 4.3. The problem of `memory_lock` contention has been fixed in newer versions of IRIX.

The final source of performance loss in the kernel is additional stall time. Updates from a processor to shared kernel structures will invalidate the data in all the other processors' caches if it is there. This requires that the data be loaded from memory or another cache when it is needed again by the processor, leading to additional memory stall time that would not be seen in a system with a smaller configuration, especially in a uniprocessor system. We will quantify the performance impact of additional memory stall time because of updates to shared structures in Section 4.3.

2.3 Summary

The perceived lack of predictable performance on current shared-memory multiprocessors is a result of the current resource allocation schemes on such machines. These schemes lack adequate controls for allocating resources among processes belonging to competing logical entities, such as users. We propose "performance isolation", a new resource management scheme that allows a multiprocessor to be flexibly shared by multiple logical entities. Performance isolation provides both predictable performance for tasks based on their share of the machine and good throughput for the entire machine.

To implement performance isolation we introduce a new kernel abstraction, the software performance unit (SPU). The SPU associates the processes belonging to a logical entity with a share of the machines resources, and controls how these resources may be shared with other SPUs. Using SPUs we implement performance isolation in the IRIX5.3 kernel for CPU time, memory, and disk bandwidth. We introduce metrics and mechanisms to provide isolation between SPUs, by counting and limiting the usage of resources by SPUs to their allocations. We then introduce careful sharing of idle resources between SPUs to improve throughput without affecting isolation.

Finally we analyze some of the limitations of current SMP kernels that can make performance isolation difficult. These involve contention for spinlocks and semaphores, and additional stall time because of updates to shared kernel structures. However, these limitations are not specific to performance isolation, but actually need to be addressed in the design of scalable operating systems for shared-memory multiprocessors.

In Chapter 4 we will demonstrate the effectiveness of performance isolation by running different workloads on the implementation described here, after describing our experimental environment in Chapter 3.

Chapter 3
Experimental Environment

The experiments in this thesis are done using SimOS [RHW+95][RBD+97][WiR96], a complete machine simulator. In this section we will describe SimOS, discuss the features that we use extensively, and in the process justify why we use a simulator instead of running the experiments on a real machine. We then describe the general architecture and system parameters of the multiprocessor systems that we consider in our experiments. Finally we justify our choice of operating system for this thesis.

3.1 SimOS

We conduct our multiprocessor experiments using the SimOS machine simulator. SimOS is a machine simulation environment that simulates the hardware of a uniprocessor or multiprocessor computer system. Unlike many other simulators that simulate only one or more subsystems, SimOS simulates the complete computer system including processors, caches, memory systems, and a number of different I/O devices including SCSI disks, ethernet interfaces, and a console. As a result, SimOS is able to boot a commercial operating system and run unmodified application binaries. It currently simulates a machine based on the MIPS instruction set architecture, and it boots the IRIX5.3 operating system from Silicon Graphics that we use and modify in our studies.

There are a number of reasons why we chose to use SimOS for our experiments rather than running on a real multiprocessor system. We outline them below:

- **Detailed statistics:** SimOS being a simulator is able to provide extremely detailed statistics about the workload that is being run with no perturbation of the workload execution. Hardware statistics at this fine detail cannot be obtained even on current systems that implement a number of counters in hardware. In addition to the expected hardware statistics, SimOS is able to provide visibility of events in the operating system and the applications being run, again with no perturbation of the system. Examples of these statistics include the number of times a specific routine is called, the breakdown of the time spent in the routine into stall time, synchronization time, time spent waiting for semaphores, disks, etc. The statistics collection mechanism is also extremely flexible and can be customized for each run. The only cost of collecting detailed statistics is the time required to complete the simulation. The capability to collect and analyze detailed statistics was central to this dissertation because this dissertation is about performance and performance optimizations.
- **Configuration flexibility:** Trying new hardware configurations with SimOS is as easy as changing a few parameters in a configuration file. We are able to vary almost any aspect of the system. Some examples are number and speed of processors, memory size, architecture, and latency, and cache size. As we shall see this flexibility is very important when doing the type of studies we describe in this dissertation. Achieving this flexibility on a real system would be painfully difficult or nearly impossible.
- **New architectures:** Using a simulator allows us to propose and study architectures and hardware features that are not currently available. Without this flexibility we would be restricted to designing for currently available hardware and architectures. The FLASH machine that is the motivation for this work, and is required for the data locality study, is not yet available. By using a simulator, we are able to conduct real experiments before the hardware is available. Another example is the cache-miss counters that we propose for driving migration and replication in the operating system. Such counters were not available when we did the study. They are just becoming available in systems, but do not provide the functionality that we propose, and do not provide the flexibility required for experimentation.

- **Repeatable runs:** SimOS supports a facility for generating checkpoints that provides a snapshot of the entire system state at any point during the execution of the workload. Using this facility we can take a checkpoint at the beginning of an interesting part of the workload, and repeatedly restart from the checkpoint for different values of interesting system parameters. This is crucial for studying the performance of multiprogrammed workloads such as the ones that we study in this thesis. Such workloads are extremely chaotic and difficult to synchronize on real systems, leading to great variability in successive runs. By taking and restarting from checkpoints we can reduce the variability due to external factors.

There are some negatives to using a simulator instead of real hardware.

- **Simulation speed:** The length of workload runs that we are able to study is limited by the speed of simulation. SimOS provides multiple simulation modes with a speed-detail trade-off [RHW+95]. Using the faster modes (10 times slowdown), we are able to position workloads at interesting points that can then be studied in detail using the slower detailed mode. In the detailed mode that we use, there is a 100 – 500 times slowdown over running directly on the hardware for each processor simulated. This slowdown imposes restrictions on the duration of the workload that we can observe.
- **Processor Model:** SimOS supports multiple processor models. The model that we use for our study is based on the MIPS R4000 that is super-pipelined and achieves almost one instruction per cycle. A more complex processor model available is based on the MIPS R10000, and supports features such as multiple issue of instructions, out of order execution, speculation, and lockup-free caches. However, the slowdown when using this more complex model was too large for it to be used effectively on the studies that we wanted to do for this dissertation.
- **Realism:** SimOS has not been explicitly validated against any particular machine. SimOS has been partially validated as reported in [RHW+95], and studies done using SimOS have shown that performance effects observed on the simulator are valid on similar real systems too. The correctness of execution has been validated in numerous cases; unmodified applications generate the same results as on real systems. It is difficult to imagine the kernel code successfully completing these complex workloads if SimOS did not execute it correctly.

- **Variations in runs:** The flip side of the repeatability argument is that the statistical variations between different runs, even when not restoring from a checkpoint, is less than on real hardware. This will cause less potential corner cases to become visible during runs. This is less of a concern for this dissertation because we are showing the feasibility of an idea, and the performance of a prototype implementation to prove that the concept will work in real life.

For this dissertation we chose to use SimOS because the benefits of using SimOS greatly outweigh the negatives. For more information about the details of SimOS, the reader is directed to [RHW+95] [RBD+97] [WiR96].

3.2 Machine Architecture

For this thesis we use SimOS to simulate multiprocessors in a variety of configurations, both for the performance isolation study and the data locality study. We will now describe the basic parameters of the system that we use for our studies. First, we describe the parameters that are common across both studies. Then we describe the memory system parameters that are different for each study. The parameters are summarized in Table 2.

3.2.1 Common System Parameters

The multiprocessor we consider in most cases has eight CPUs. This choice is a trade-off; eight processors is large enough to demonstrate most of the important effects we need to consider, and small enough to simulate in reasonable time. Each processor models a 300 MHz MIPS R4000 processor. The R4000 TLB has 48 entries, each of which can map two pages. The page size is 4Kbytes for our experiments. The primary instruction and data caches are write-back, two-way set associative, 32 Kbytes in size with a line size of 64 bytes. Both caches have a hit time of 1 cycle. The secondary cache on these systems is a merged instruction and data cache that is write-back and two-way set associative with a line size of 128 bytes. The cache size is 1 Mbyte for the performance isolation study, but is varied for the data locality study because the amount of stall time in the workload is an important determinant of the benefit provided by improving data local-

Parameter	**Value**
Processor	300MHz MIPS R4000
Number of Processors	Eight
Page Size	4Kbytes
TLB	48 entries each maps two pages
Primary Cache	Instruction and Data each 32 Kbytes, 2-way, write-back, 64 byte lines, 1 cycle hit time
Secondary Cache	(256 Kbyte - 1 Mbyte) combined Instruction and Data, 2-way, write-back, 128 byte lines, 50 ns hit time
Memory System (Performance Isolation)	Bus-based UMA memory, 300ns for cache-line, 1.2 Gbytes/s split transaction bus
Memory System (Data Locality)	CC-NUMA memory, 300ns local memory access, 1200ns remote memory access
Operating System	IRIX 5.3 from Silicon Graphics
Disk	HP97564

TABLE 2. Parameters for the experimental environment. The values for relevant parameters used for the experiments in later sections are summarized in this table.

ity. The hit time to the secondary cache is 50 nanoseconds. The disk is based on a validated model of the HP97564 disk [KTR94].

3.2.2 Performance Isolation Specific Parameters

For the performance isolation study we use a shared-bus memory model. This allows us to clearly show the effects of performance isolation without the confusion of NUMA effects. The parameters for the shared-bus model are based on that of the SGI CHALLENGE SMP multiprocessor with the POWERpath-2 memory bus. The latency to fetch a cache line from memory is 300 nanosecond in an unloaded system. A 1.2 Gbps split transaction bus is assumed for modelling bus contention. Even though we chose to model a specific multiprocessor, the conclusions from the performance isolation experiments are quite independent of the memory system parameters and should be applicable to a wide range of shared-memory multiprocessors.

3.2.3 Data Locality Specific Parameters

For the data locality study we use a CC-NUMA memory system modelled on the FLASH multiprocessor [KOH+94], and generally representative of NUMA memory systems. We assume a single processor per node, and eight nodes. The benefits of migration and replication should be seen even with this small configuration because the probability that a process would randomly find a page in local memory is already quite small (12.5%). The ratio of local to remote cache-miss latency is 1:4 as expected in the FLASH multiprocessor. The 1:4 ratio is also about the middle of the range for announced commercial machines from SGI and Sequent. The memory-access latencies without contention are 300ns for a local memory access and 1200ns for a remote memory access. There is 256 Mbytes of main memory in the system. The secondary cache size is set at either 512 Kbytes or 256 Kbytes depending on the workload; we will discuss the choice to cache sizes later.

3.3 Operating System

We have chosen to do our operating system studies by modifying the IRIX 5.3 operating system from Silicon Graphics. Silicon Graphics' IRIX operating system is based on the UNIX® operating system and includes merged features from both Berkeley and AT&T System V versions. We outline the reasons that prompted us to choose IRIX for this study:

- Most importantly, we had access to the source code for IRIX.
- IRIX had already been parallelized and ran on small bus-based multiprocessors. Therefore, we had to make changes only when adding new functionality or optimizing existing functionality.
- This thesis work was done in the context of the FLASH project that is building a CC-NUMA multiprocessor system based on MIPS processors. Each node in FLASH is based on an SGI workstation. The operating system for the FLASH machine will be derived from IRIX.
- IRIX is a production quality operating system, which makes our performance studies more credible.

Chapter 4
Experimental Results for Performance Isolation

The goal of this chapter is to demonstrate how well performance isolation is able to achieve its twin objectives of isolation and sharing. To this end, we will run a number of workloads using our implementation of *performance isolation* (**PIso**), as described in Section 2.2. Each workload has jobs from multiple SPUs, the units of performance isolation. For each workload, we will demonstrate that performance isolation is able to provide isolation between SPUs independent of the load within each SPU. We will also show that performance isolation is able to share resources between lightly-loaded and heavily-loaded SPUs to provide good throughput without affecting isolation.

The rest of this chapter is organized as follows. In Section 4.1, we introduce the two other resource allocation schemes, fixed quotas (Quo) and SMP (SMP). We use these schemes as references to demonstrate the effectiveness of performance isolation through comparison. In Section 4.2, we describe the different workloads that we use to demonstrate the results of performance isolation. In Section 4.3, we show the results, demonstrating isolation and sharing, for a workload running parallel make jobs and using eight SPUs. This workload, because of its characteristics, is designed to stress our implementation of performance isolation. For this workload, we also compare the results using performance isolation with the results of running one of the SPUs

Configuration	Description
Fixed Quota (**Quo**)	Fixed quota for each SPU with no sharing.
Performance Isolation (**PIso**)	Performance isolation with policies for isolation and sharing.
SMP operating system (**SMP**)	Unconstrained sharing with no isolation

TABLE 3. Different resource allocation schemes for MPs. Each workload is run with three different resource allocation schemes, Performance Isolation (PIso), Fixed Quotas (Quo), and IRIX5.3 representative of current shared-memory operating systems (SMP). Quo is the ideal for providing isolation, and SMP is the ideal for sharing and total throughput. PIso will attempt to match the best of Quo and SMP

on an equivalent individual workstation. The point here is to show how the shared kernel resources, such as spinlocks and semaphores, described in Section 2.2.4 can affect performance isolation. Sections 4.4, 4.5, and 4.6 highlight performance isolation for each of the resources CPU, memory, and disk bandwidth respectively. In Section 4.7, we revisit the workload described in Section 1.5 that was used to motivate the need for performance isolation, and show the improvements possible in that scenario when performance isolation is used. Section 4.7 also summarizes the results from this chapter.

4.1 Experimental Setup

To clearly demonstrate the effectiveness of performance isolation, we will also run each workload on two other resource allocation schemes, as shown in Table 3. The first uses *fixed quotas* (**Quo**) to statically allocate the resources on the system to the different SPUs. This scheme represents the best case for achieving isolation between SPUs because resources are dedicated to SPUs and there is no sharing. The second is unmodified[1] IRIX5.3 (**SMP**) that provides only the current unconstrained sharing of resources, and is representative of the resource allocation scheme in current shared-memory multiprocessors. SMP represents the best case for sharing of resources to achieve good workload throughput. For each workload, we will demonstrate that performance isolation is able to provide isolation between SPUs comparable to fixed quotas and sharing between SPUs for good throughput comparable to SMP; the best qualities of these two schemes.

1. The IRIX5.3 kernel used for these experiments has been modified to include the semaphore fixes described in Section 2.2.4, and therefore has better performance than the standard IRIX5.3 kernel.

4.2 Workload Description

The workloads for the experimental runs are chosen to demonstrate the effectiveness of performance isolation in general and for each of the different resources, CPU time, memory, and disk bandwidth. The workloads with applications, system configurations, and the details of SPU configurations are summarized in Table 4. We briefly describe each workload, and what part of performance isolation it will demonstrate.

Pmake8: This workload has eight SPUs on an eight-processor system. Each SPU runs either one or two jobs concurrently. All jobs are similar, and each is a parallel make of six files from the gnuchessx program. The parallel make spawns two compiles in parallel. Each job has its sources on a separate disk, and the file buffer cache is warm with all the necessary files. This workload is kernel and filesystem intensive with many processes starting and finishing. This entry and exit of processes leads to changing CPU and memory requirements for the SPUs, and significant kernel activity that leads to interference between processes in the kernel. All this activity makes it a challenge to provide performance isolation for such a workload.

CPU Isolation: This workload is used to highlight performance isolation for CPU time. It consists of two SPUs on an eight processor system and sufficient memory for all the applications. One SPU consists of a parallel version of the OCEAN application with four processes. OCEAN is a scientific application from the SPLASH [WOT+95] suite. The other SPU contains three copies each of VCS and Flashlite. Flashlite is a sequential application that accurately simulates the Flash memory system, and in this workload it is driven by a four-processor version of the FFT application. VCS is a verilog simulation of a complicated integrated circuit, also a sequential application. In VCS, the input circuit is compiled into a large code segment and then executed. This is a compute intensive workload with unbalanced CPU loads, four processes in one SPU and six in the other.

Memory Isolation: This workload is similar to the Pmake8 workload. However, there are only two SPUs on a four processor system. Each SPU runs either one or two jobs. Each job is a parallel make of twelve files from the gnuchessX program. Each job spawns as many as four compiles simultaneously. The important difference is that the total memory in the system is reduced to 16 Mbytes. This workload highlights the effectiveness of performance isolation for memory. Each

Workload	System Parameters	Applications	SPU Configuration
Pmake8	8 CPUs, 44 Mbytes memory, separate fast disks	Multiple Pmake jobs (two parallel compiles each)	Balanced: 8 SPUs (1 job) Unbalanced: 4 SPUs (1 job), 4 SPUs (2 jobs)
CPU Isolation	8 CPUs, 64 Mbytes memory, separate fast disks	Ocean (4 processors), 3 Flashlite, 3 VCS	2 SPUs: 1 SPU Ocean, 1SPU all Flashlite and VCS
Memory Isolation	4 CPUs, 16 Mbytes memory, separate fast disks	Multiple Pmake jobs (four parallel compiles each)	Balanced: 2 SPUs (1 job) Unbalanced: 1 SPU (1 job), 1 SPU (2 jobs)
Disk Isolation	2 CPUs, 44 Mbytes memory, shared HP97560 disk	Pmake and File copy	1 SPU pmake, 1 SPU file copy

TABLE 4. Description of the workloads used for the performance results. For each workload we show the relevant system parameters, the applications used in the workload, and the SPU configuration for performance isolation.

SPU has enough memory to run one job, but is memory starved when running two jobs. Each job has its sources on a separate disk and the file buffer cache is warm with all necessary files.

Disk Isolation: This workload focuses on disk bandwidth, and consists of two SPUs on a two CPU system with sufficient memory. One runs a parallel make job as in the Pmake8 workload. The other makes a copy of a large file. The source for the parallel make, and the source and destination files of the copy share a disk. The file buffer cache is cold. The copied file is large enough to cause the buffer cache to overflow, causing both reads and writes to the disk. This causes contention for the disk, and the workload will be used to demonstrate the effectiveness of performance isolation for disk bandwidth.

4.3 Experiments Using the Pmake8 Workload

The first workload consists of a number of pmake jobs as described in Table 4. There are eight SPUs for performance isolation corresponding to eight different users on an eight-way multiprocessor. The hardware resources are shared equally between the eight SPUs. The sharing policy is to share all *idle* resources with any of the other SPUs that need resources.

We consider two different scenarios for the distribution of processes to SPUs as shown in Figure 5. The first is a balanced configuration with eight jobs, one per SPU for performance isola-

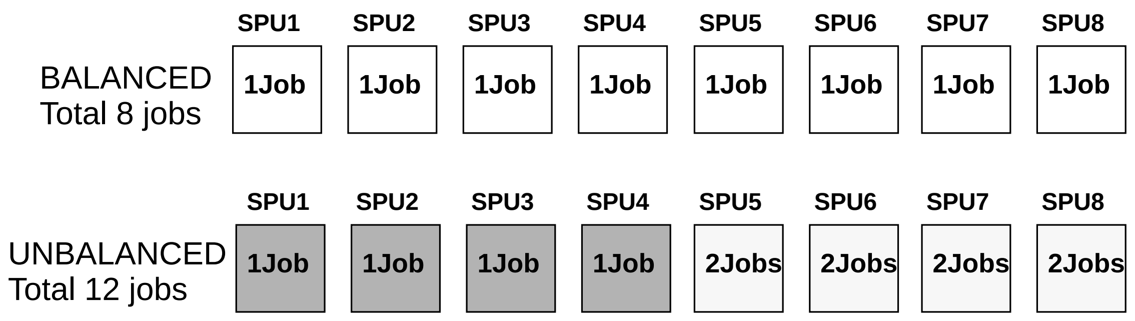

FIGURE 5. Balanced and Unbalanced configurations for the Pmake8 workload. The figure shows the distribution of jobs to SPUs in the balanced and unbalanced configurations for the Pmake8 workload. The **BALANCED** configuration has eight jobs, one per SPU. The **UNBALANCED** configuration has 12 jobs. The lightly-loaded SPUs (1-4) have 1 job each, and the heavily loaded SPUs (5-8) have 2 jobs each.

tion. The second is an unbalanced configuration where four SPUs (1 - 4) run one job each, and the other four SPUs (5 - 8) run two jobs each. The first configuration has a balanced load in each SPU, and so should not be affected by isolation or sharing. This is our base configuration. The second configuration has an unbalanced load. SPUs 1 - 4 have only one job each, and should see a benefit from being isolated from SPUs 5 - 8 which now have two jobs each. On the other hand, SPUs 5 - 8 that have a heavier load should see some benefit from sharing of resources that may be idle in SPUs 1 - 4, and also those that become available when the single jobs in these SPUs finish. The jobs in SPUs 1 - 4, because of the lighter load, will finish before those in SPUs 5 - 8. This workload will therefore be used to demonstrate performance isolation for both processor and memory resources, and the effect of contention for kernel resources. We will show that performance isolation is able to both isolate SPUs from overall system load, and allow the sharing of idle resources to improve the performance of SPUs under heavy load.

4.3.1 Isolation

We will first study the provision of isolation, i.e. whether performance isolation can isolate SPUs from changes in system load. To do this we compare the performance of the jobs in SPUs one to four for the balanced and unbalanced configurations. These SPUs maintain constant resource requirements (one job) in both the balanced and the unbalanced configurations. However, the unbalanced configuration has higher system load because of the additional jobs in SPUs 5 – 8. Therefore, if a system provides good isolation, the performance of the jobs in SPUs 1 – 4 should

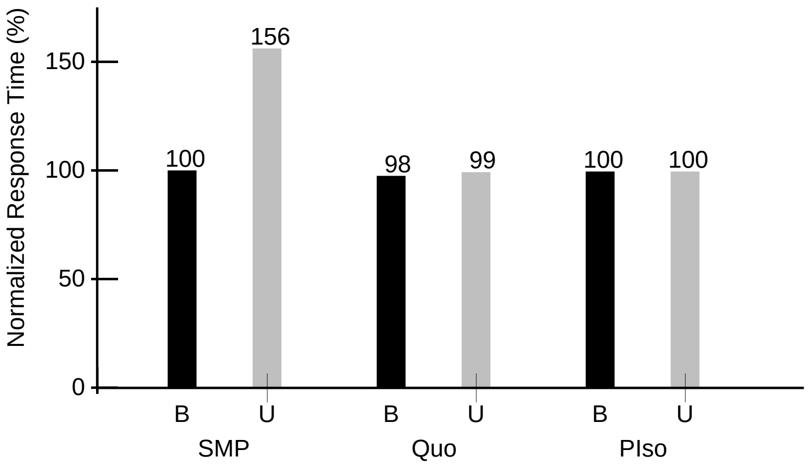

FIGURE 6. Effect of Isolation in the Pmake8 workload. Average response time for jobs in the lightly-loaded SPUs (1–4) for the balanced (B) and unbalanced (U) configurations normalized to the SMP time in the balanced configuration. Performance Isolation (PIso) and Fixed Quotas (Quo) are able to maintain performance as system load increases, whereas SMP is unable to do so.

not change. Figure 6 shows the average response time for these jobs in the balanced and unbalanced configurations, normalized to the case of SMP in the balanced configuration.

Performance Isolation (PIso) is able to keep the performance of jobs in the lightly-loaded SPUs (1 – 4) the same in the balanced and unbalance configurations, despite the increase in overall system load in the unbalanced configuration. It does this by allocating resources based on SPUs, and effectively isolating jobs in an SPU from the load of jobs in other SPUs. The fixed quota scheme (Quo) is the ideal for providing isolation. Performance Isolation (PIso) is able to achieve the same level of isolation for jobs as the fixed quotas scheme (Quo).

In contrast, the regular SMP kernel (SMP) is unable to provide any isolation between jobs. The response time for the jobs in SPUs 1 – 4 increases by 56% when going from the balanced configuration with 8 jobs to the unbalanced configuration with 12 jobs. SPUs 5 – 8 introduce two jobs each and thereby increase the load on the system. This kernel does not differentiate between the jobs, and gives all jobs approximately the same share of resources. Therefore, there is an increase in the response time of all jobs including those of the lightly-loaded SPUs (1 – 4).

4.3.2 Resource Sharing

We next study the other aspect of performance isolation, controlled sharing. Can performance isolation help the performance of heavily-loaded SPUs, by allowing them to utilize resources that are

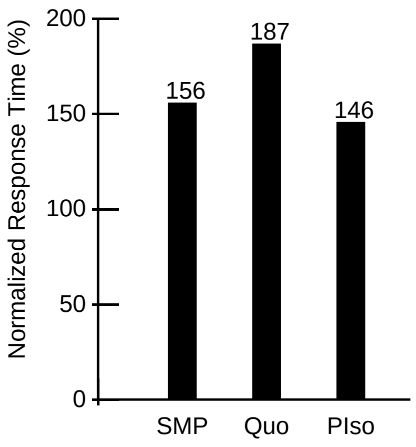

FIGURE 7. Effect of resource sharing in the Pmake8 workload. Response time for jobs in the heavily-loaded SPUs (5–8) for the unbalanced (12 jobs) configuration normalized to the SMP time in the balanced configuration. Performance Isolation (PIso) allows the heavily-loaded SPUs to use the idle resources in the system to improve performance. Fixed Quotas (Quo) does not allow this and so performance suffers.

idle in the lightly-loaded SPUs? Better utilization of resources would result in better throughput for the workload. To see the effect of controlled sharing, we consider the performance of the heavily-loaded SPUs (5 – 8) in the unbalanced configuration. The performance of the jobs in these SPUs is shown in Figure 7. The average response time is shown for each of SMP, Quo, and PIso normalized to the SMP performance in the balanced configuration.

The SMP kernel represents the best case for sharing and throughput. The jobs in SPUs 5 – 8 do well under SMP because they are able to take up more than their “fair share” of resources. This scheme treats all jobs equally, and gives them all the same level of resources. As a result the response time of the jobs in these SPUs increases by only 56% even though their resource requirements double.

Fixed quotas (Quo), the ideal for isolation, fails when it comes to sharing. Quo increases the response time for these jobs by 87%, performing much worse than the SMP case. By imposing fixed quotas, Quo was able to isolate the lightly-loaded SPUs, but leaves resources idle after the jobs in these SPUs finish. The jobs in the heavily-loaded SPUs are unable to use these resource because of the static fixed quotas.

In contrast, performance isolation is able to provide controlled sharing of resources in addition to isolation between SPUs. The performance of these jobs with PIso is as good as that with SMP. Performance isolation achieves this by carefully allowing these heavily-loaded SPUs to utilize resources that are idle in the lightly-loaded SPUs (1 – 4). From the isolation numbers for SPUs 1

– 4 in Figure 6, we know that this sharing is achieved without breaking isolation for the lightly-loaded SPUs.

Actually, the response time for PIso is a little better than that of SMP. From a pure CPU-scheduling viewpoint, they should have performed about the same. The difference is a result of the effect of different amounts of memory available during the run. This happens because the light-load SPUs finish early, and they release memory in addition to CPUs. This memory then becomes available to the heavy-load SPUs. In the SMP case all the jobs are equal, and finish at about the same time, using their share of memory till the end.

4.3.3 Comparison with a Single Workstation

We now compare the performance of an SPU on a shared-memory multiprocessor and an equivalent smaller machine. This comparison is used to highlight some of the hidden problems with providing performance isolation because of contention for kernel resources. These problems, additional stall time and contention for mutual exclusion structures such as spinlocks and semaphores, were discussed in Section 2.2.4.

An equivalent smaller machine has resources corresponding to that of the SPU on the shared-memory multiprocessor. For the Pmake8 workload, the equivalent smaller machine would be a uniprocessor workstation. The equivalent workstation numbers are not really done on a workstation configuration because of the difficulty of specifying an equivalent memory configuration. Specifying an eighth of the memory would weigh against the workstation configuration because the kernel takes up a fixed number of pages leaving less pages for the workload. Therefore, these runs are done by setting up eight user SPUs, with only one SPU running the workload and the other SPUs idle. In this configuration, the equivalent workstation gets one-eighth the user memory of the shared-memory multiprocessor. This configuration could be considered slightly favorable to the workstation case.

Figure 8 compares the performance the Pmake8 workload running with performance isolation (PIso) to that of an equivalent workstation (Ref) running the workload of a single SPU. The performance of SPUs 1–4 is compared with running one job on the equivalent workstation, and that of SPUs 5–8 in the unbalanced case with two jobs on a workstation. Performance isolation pays a

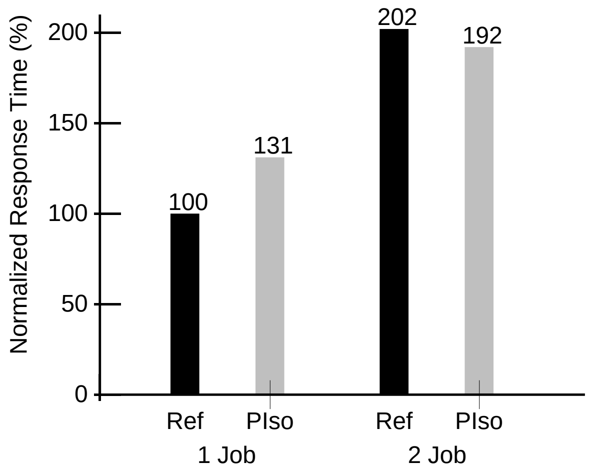

FIGURE 8. Comparison with a single workstation. Response time for the Pmake8 workload with performance isolation (PIso) compared to that on an equivalent single workstation (Ref). The numbers for both 1 Job and 2 Jobs in an SPU and on a workstation are shown. The response times are normalized to that of 1 Job on a workstation. In the two job case PIso does better by using idle resources in the MP system.

penalty of 31% when compared to that of the equivalent workstation in the one job case. It does performs better in the two job case by using resources not utilized by the lightly-loaded SPUs. As mentioned in Section 2.2.4, there are a number of reasons related to contention for kernel resources that can cause this overhead. It should also be noted that this workload is kernel intensive and so presents the worst side of this problem.

Let us carefully consider where the extra time is spent. Figure 9 shows the non-idle execution time for the one job run for three configurations: the equivalent workstation (Ref), Performance Isolation (PIso), and IRIX5.3 (SMP). A number of interesting observations can be made from Figure 8 and Figure 9.

- The overheads in the performance isolation case, though higher than Ref, are the same as for the SMP case. No significant additional overhead is introduced by the mechanisms that provide performance isolation.
- The increase in execution time is entirely because of increase in kernel time, i.e., differences between an uniprocessor and a multiprocessor kernel. There is no increase in user time.
- The increase in response time from Figure 8 (31%) is matched almost exactly by the increase in computation time in Figure 9. No significant additional idle time is spent waiting to acquire semaphores that are currently held by another process. Our implementation was successful in removing the main causes of semaphore waits.

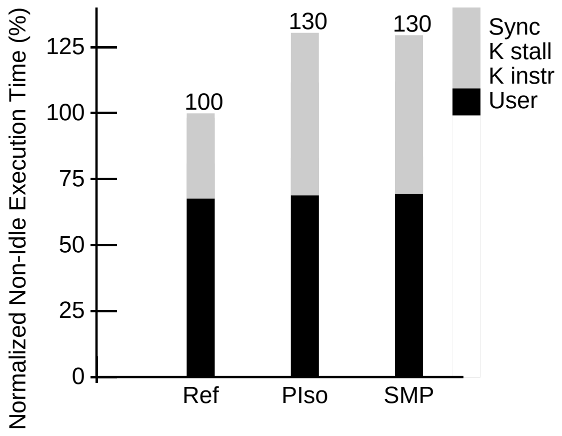

FIGURE 9. Non-idle execution times for Pmake8 workload. The times for all cases are shown normalized to that of the appropriate equivalent single workstation reference case (Ref). Each bar shows User, Kernel instruction, Kernel stall and Synchronization time for that entire run. There is additional kernel stall time and synchronization time for the MP schemes (PIso, and SMP) compared to the Ref, and this is not because of providing Performance Isolation.

- The increase in overhead from the single workstation to the multiprocessor is mostly because of two causes, additional synchronization time for spinlocks (about 60%) and additional kernel stall time because of updates to shared structures (40%).

Analyzing the synchronization time in more detail, we found that more than 80% of this time is spent contending for a single lock, `memory_lock`. This is a coarse lock that protects most of the important structures and flags in the VM system — the lists of free pages, the hash table that maps vnodes and file offsets to memory pages, the flags for each physical page frame descriptors, and a number of other smaller structures. A more scalable design would replace this lock with a set of fine-grain locks.

We next analyzed the kernel stall time in detail by function, comparing the eight processor runs with the equivalent single workstation ones. About half the extra stall time is attributable to VM structures and functions, and about a third is because of filesystem and buffer-cache related structures and functions. This is not surprising because the Pmake8 workload, with many short-lived processes and much file activity, is very memory system and filesystem intensive. For the memory system, the contended structures are the heads of free page lists and hash tables.

These synchronization and stall issues are general scalability issues for multiprocessors and SMP kernels, and are not particular to performance isolation. Some of these problems, especially the VM bottlenecks, have been addressed in newer versions of IRIX.

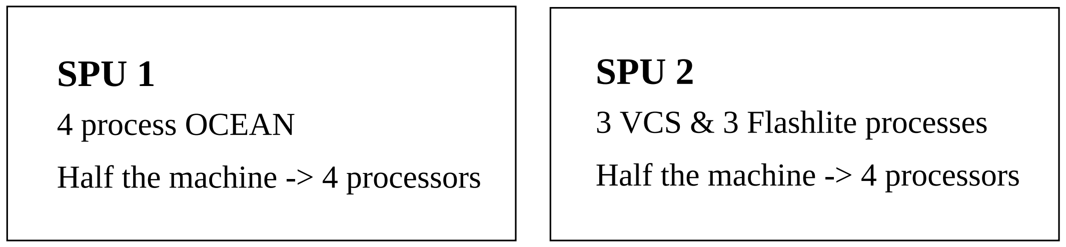
SPU 1
4 process OCEAN
Half the machine -> 4 processors
SPU 2
3 VCS & 3 Flashlite processes
Half the machine -> 4 processors

FIGURE 10. The structure of the CPU Isolation workload. The CPU isolation workload has two SPUs each of which gets half the machine. SPU 1 runs a four processor parallel Ocean application. SPU 2 runs three copies of VCS and three copies of Flashlite. SPU 2 has more CPU load than SPU 1.

4.4 Experiments Using the CPU Isolation Workload

The CPU isolation workload consists of compute-intensive scientific and engineering jobs with kernel time only at the start-up phase. The structure of the workload is shown in Figure 10. The workload has a total of ten processes on eight processors, and it will be used to demonstrate CPU isolation. There is adequate memory for all applications and so memory is not an issue for performance. For the performance isolation runs there are two SPUs corresponding to two users. Therefore, each SPU is allocated four CPUs. One SPU runs the four process Ocean application, and the other SPU runs the three Flashlite and three VCS jobs. In this workload, Ocean runs in the SPU with less load as it has four processors for four processes. The other SPU has a heavier load with four processors for six processes. Figure 11 shows the results for this workload. Response time numbers are averages of all the jobs of a type normalized to the SMP case. We will show how well performance isolation is able to satisfy the two goals of isolation and sharing for CPU time.

For isolation we will concentrate on the performance of Ocean, which runs in the SPU with a lighter load and should benefit from isolation. For the Ocean processes, performance isolation (PIso) is able to improve the response time compared to SMP. PIso does this by isolating the processes within an SPU, preventing interference from the other applications. In the SMP configuration, the Ocean processes run slower because all the processes are treated equally. Therefore Ocean gets less than its “fair share” of CPU time, and sees interference from the other processes. Fixed quotas (Quo) the ideal case for isolation does a little better than PIso.

For sharing we will focus on the Flashlite and VCS processes that are running in the SPU with heavier load. SMP is the ideal for sharing of resources. The VCS and Flashlite processes perform significantly better with Performance Isolation (PIso) than with Fixed quotas (Quo). Performance Isolation achieves this by carefully utilizing CPU resources that would have been idle in the Ocean SPU, in such a way as to not affect isolation for the Ocean processes. Performance isolation (PIso) is also able to keep the performance of the VCS and Flashlite processes comparable to that of the SMP configuration in this case. Because the workloads are dissimilar this last result will not generally be the case, and will be dependent on the relative durations of the applications.

4.5 Experiments Using the Memory Isolation Workload

The memory isolation workload will highlight performance isolation for main memory. This workload and the experiments are similar to that of the Pmake8 workload, but with a focus on memory. The structure of this workload is shown in Figure 12. There are two SPUs on a four processor system. The balanced configuration has one job in each SPU, and the unbalanced configuration has two jobs in SPU2 and only one job in SPU1. The total memory size is deliberately made small (16 Mbytes). This memory is enough to run one job in each SPU, but leads to memory pressure in a SPU with two jobs. The results highlighting isolation and sharing are shown in Figure 13. The graphs are similar to the ones for the Pmake8 workload, and can be interpreted

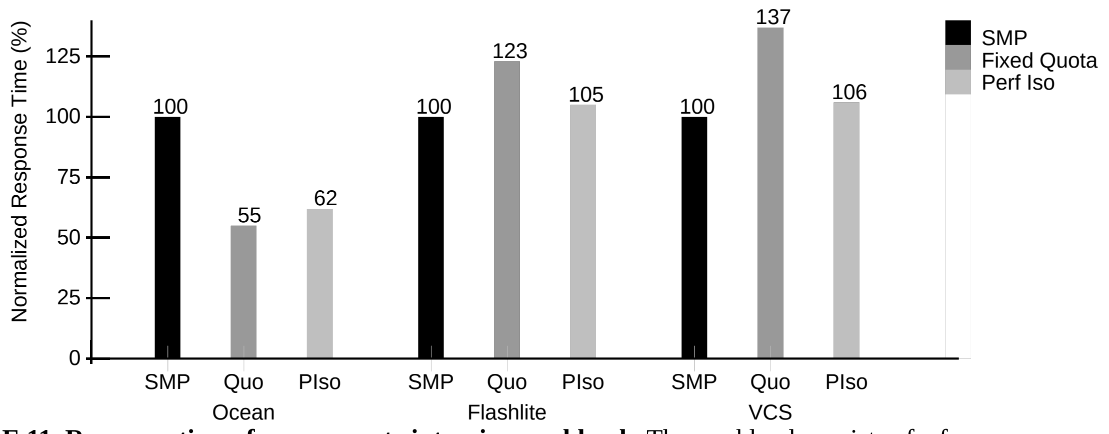

FIGURE 11. Response times for a compute intensive workload. The workload consists of a four-process parallel OCEAN, three Flashlite, and three VCS jobs. The system has eight processors and plenty of memory. For performance isolation, Ocean runs in one SPU (four processes on four processors) and all the Flashlite and VCS jobs in another (six processes on 4 processors). The response time (latency) shown is the average for all jobs of a type, and is normalized to that for the SMP case.

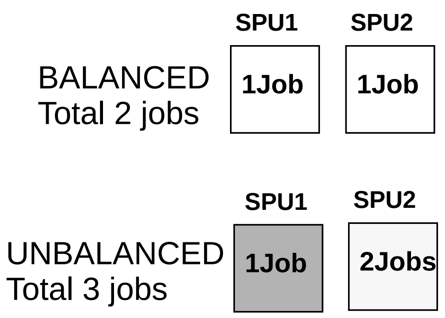

FIGURE 12. Balanced and Unbalanced configurations for the memory-isolation workload. The figure shows the distribution of jobs to SPUs in the balanced and unbalanced configurations for the memory-isolation workload. The BALANCED configuration has two jobs, one per SPU. The UNBALANCED configuration has three jobs. The lightly-loaded SPU 1 has 1 job, and the heavily loaded SPU 5 has 2 jobs. The machine resources are divided equally among the SPUs. To create memory pressure, the system is configured with only 16 Mbytes of memory.

similarly. We focus first on isolation and then on sharing, and show that performance isolation can provide both.

The effect of providing isolation is illustrated by the graph on the left that shows the performance of the job in SPU1 in the balanced and unbalanced configurations. Performance isolation is able to provide isolation to maintain performance as the background system load increases. Only a 13% decrease in performance compared to the SMP case of 45% decrease. SMP treats all processes the same, resulting in less resources and lower performance for the processes of SPU1 as system load increases.

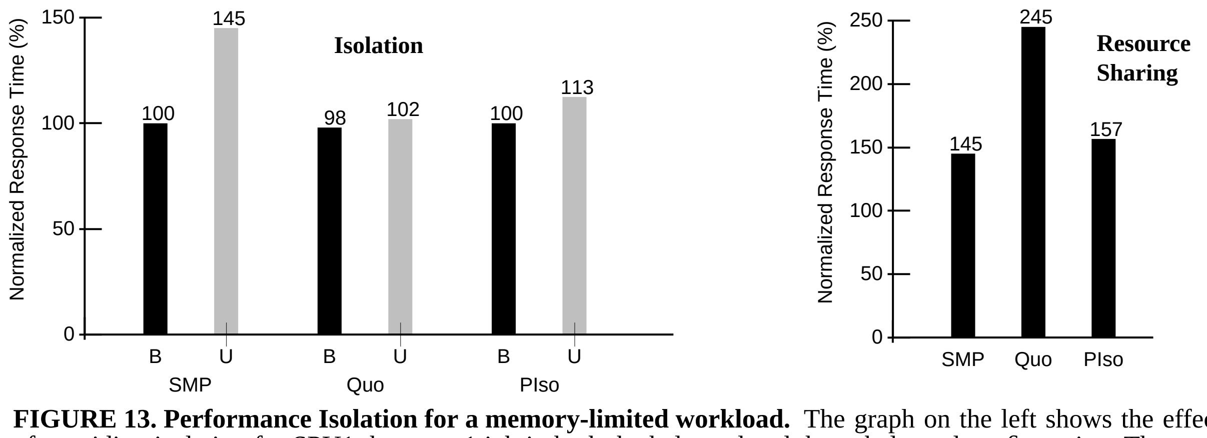

FIGURE 13. Performance Isolation for a memory-limited workload. The graph on the left shows the effects of providing isolation for SPU1 that runs 1 job in both the balanced and the unbalanced configuration. The graph on the right shows the effect of resource sharing for SPU2 in the unbalanced configuration running two jobs. The response time (latency) for all jobs are shown normalized to that of the balanced SMP case. Both performance isolation (PIso) and fixed quotas (Quo) are able to provide isolation against background system load. Only performance isolation (not Quo) is able to help the heavily-loaded SPU2, by sharing idle resources.

The effect of sharing is illustrated by the graph on the right that shows the performance of the jobs in SPU2 in the unbalanced configuration. The loss in performance with fixed quotas is large, 145% decrease in performance compared to the balanced configuration. A 100% reduction in performance is accounted for by CPU time because there are two jobs instead of one. The additional 45% is because of the memory limitation when running two jobs in one SPU without sharing. Performance isolation through the careful sharing of resources — memory and CPU in this case — delivers significantly better performance, close to the SMP case.

4.6 Experiments Using the Disk Bandwidth Isolation Workload

We demonstrate the effect of performance isolation on disk bandwidth using two different I/O intensive workloads. For these runs we use the disk model based on the HP97560 disk. To reduce the length of the simulation runs we use a scaling factor of two for the disk model, i.e., the model has half the seek latency of the regular disk. We also make sure that the file buffer caches are cold for these experiments. To keep the experiment simple, the machine is a two-way multiprocessor.

The first workload (pmake-copy) has two SPUs, one running a pmake job, and the other a process copying a large file (20Mbytes). A single disk contains both the source and results of the pmake, and the source and destination file of the copy. The pmake makes a total of 300 requests to the disk, and these are not all contiguous as they represent multiple files and many repeated writes of meta-data to a single sector. The copy makes a total of 1050 requests to the disk. These are mostly contiguous sectors as it is reading and writing large files. There are multiple outstanding reads because of read-ahead by the kernel. The buffer cache fills up causing writes to the disk.

The disk-request scheduling algorithm in IRIX 5.3 is optimized for throughput as was described in Section 2.2.3. Its primary consideration is to reduce disk seek latency. Therefore, it considers only the current head position and the sector number of the requests when scheduling. As a result, the reads and writes of the copy that are to contiguous sectors, can lock out the more random requests from the pmake. This locking out of other requests is the same behavior that users sometimes observer when a large core file is dumped to disk.

There are many issues to consider when providing performance isolation for disk bandwidth. The additional seek time to move the disk head between two potentially different sets of disk sectors becomes an issue. Also, a SPU that is using the disk less frequently has a greater likelihood of finding a request from other SPUs currently being serviced by the disk. The isolation mechanism can let this new request bypass currently queued requests for bandwidth fairness, but it still has to wait for a currently active request to complete. Our goal with performance isolation is to provide fairness in disk bandwidth allocation and good throughput.

To show the effect of all the issues and how performance isolation is able to deal with them, we consider three different policies for scheduling disk requests for the experimental runs.

1. **Pos:** The standard head-position based scheduling, currently in IRIX.
2. **Iso:** This a blind performance isolation policy. This policy ignores head position, and only strives to provide fairness for disk bandwidth to the SPUs.
3. **PIso:** The performance isolation policy described in Section 2.2.3. This policy gives weight to both isolation and the head position when scheduling requests. The goal is to balance fairness of bandwidth and effective throughput to the disk.

The results for the pmake-copy workload for the three cases are shown in Table 5.

The performance isolation policy (PIso), by incorporating fairness, significantly reduces the response time for the pmake job (39%). In the regular IRIX case (Pos), the copy job was able to lock out the pmake requests from the disk, significantly slowing down the pmake job. The fairness provided by PIso can be seen in the significantly lower average wait time for the requests from pmake. These requests now do not have to wait for all copy requests to be processed. For the pmake job, the average time a request spends waiting in the disk queue decreases by 76%. Performance isolation, by incorporating head-position information in the scheduling decision, does not significantly change the average seek latency for the disk. The copy job, as expected, does see a reduction in performance (23%).

The blind performance isolation policy (Iso) that ignores head position is also able to improve fairness, and consequently the response time for the pmake. In this workload its performance is similar to the performance isolation policy because the pmake makes fairly irregular requests,

	Response time (sec)		Avg. Wait Time (ms)		Avg. Queue Length	Avg. Latency (ms)
Configuration	Pmake	Copy	Pmake	Copy		
Pos	15.7	12.8	94.3	38.4	5.1	10.6
Iso	9.2	15.8	23.4	63.5	3.7	11.6
PIso	9.6	15.6	23.0	75.4	4.3	11.4

TABLE 5. The effect of performance isolation on a disk-limited workload. The pmake-copy workload consists of two SPUs, one a pmake process and the other a large (20 Mbyte) file copy to the same disk as the pmake. There are three runs. First with the standard head-position based scheduling (Pos), next with the blind performance isolation scheduling (Iso), and finally with the disk bandwidth performance isolation policy that takes into account both head-position and isolation (PIso). The response time and the average queue time per request for each job is given, along with the average queue length and average disk latency.

therefore ignoring disk-head position does not result in a large penalty. However this is not always true, and completely ignoring disk-head scheduling could lead to reduced performance.

The second workload (big-and small-copy) will illustrate the importance of maintaining disk-head position as a factor in the scheduling decision. In this case also there are two SPUs, one with a process copying a small file (500 Kbytes), and the other with a process copying a larger file (5 Mbytes). Both jobs in this workload can benefit from disk-head position scheduling because they are both accessing contiguous sectors on disk in a regular manner. The results of the experiment are shown in Table 6.

The big difference between the two workloads is that in this workload, the smart performance isolation policy (PIso) is able to significantly outperform the blind one (Iso). In IRIX (Pos), the larger copy by happening to issue requests to the disk earlier than the smaller copy, is able to lock out the requests of the smaller copy. Both the PIso and Iso policies provide fairness, improving the response time of the small copy and allowing it to finish sooner than the larger one. However

	Response time (sec)		Avg. Wait Time (ms)		Avg. Queue Length	Avg. Latency (ms)
Configuration	Small	Big	Small	Big		
Pos	0.93	0.81	155.8	12.1	5.17	6.4
Iso	0.56	1.22	68.9	23.7	3.26	8.2
PIso	0.28	0.96	31.9	16.6	3.10	6.6

TABLE 6. The advantage of considering both head-position and fairness. The big-and-small-copy workload consists of two processes copying files, one a small 500Kbyte file and the other a larger 5 Mbyte file. There are three runs, first with the standard head-position based scheduling (Pos), next with the blind performance isolation scheduling (Iso), and finally with our disk bandwidth performance isolation policy (PIso). The response time and the average queue time per request for each job is given, along with the average queue length and average disk latency. The performance isolation policy performs significantly better than the Isolation only policy in this case.

for this workload, the PIso policy by also incorporating head-position information is able to provide better response times for both processes as compared to the Iso policy. By incorporating head-position information, the average seek latency per request for the PIso policy is about the same as the IRIX position-only scheduling policy (Pos), whereas by ignoring head-position scheduling the Iso policy pays almost a 30% increase in average seek latency. The average time a request spends waiting in the disk queue decreases from Iso to Piso for both the small and the large copy, 54% for the former and 30% for the latter. This workload shows that it is important to also include the disk head position information when providing performance isolation.

4.7 Summary of Results

It is clear that performance isolation has been successful at achieving both its stated goals of isolation and sharing. Using a stressful eight SPU workload and other more resource-specific workloads, we have shown that performance isolation can provide fairness when a multiprocessor is to be shared by multiple tasks, leading to more predictable performance. Performance isolation can provide isolation between SPUs comparable to that provided by a static fixed quota scheme, and at the same time provide sharing of resource and good throughput comparable to current SMP kernels.

We have shown that our implementation of performance isolation for each of CPU time, memory and disk bandwidth works. Especially interesting are our results for disk-bandwidth isolation that justify the necessity of incorporating head-position information in the performance isolation scheduling decision.

Finally, we see that most of the problems with providing performance isolation related to contention for shared kernel resources, such as updates to shared structures and synchronization, are not specific to performance isolation. The solution to these problems is required for any scalable operating system for multiprocessors.

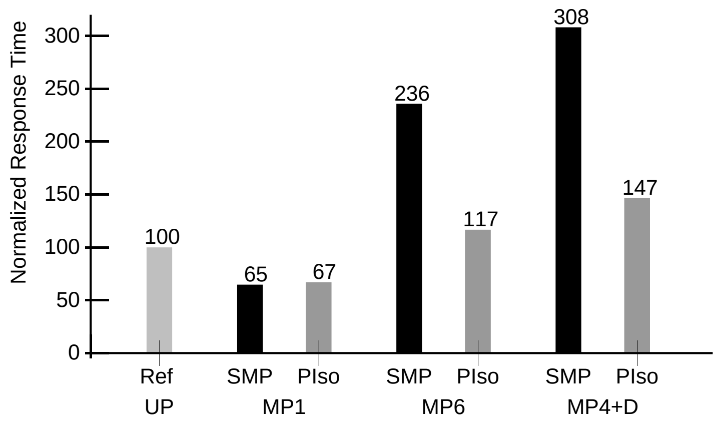

FIGURE 14. Motivation workload with and without performance isolation. The runs are the same as the ones described in Section 1.4. For each run there are two bars; with the performance isolation model (PIso) and with the regular SMP kernel (SMP).

Having demonstrated the effectiveness of performance isolation for a number of workloads and different scenarios, we now revisit the experiments done in Section 1.4 that defined the problem and motivated the need for performance isolation. Figure 14 compares the performance of the regular IRIX kernel (SMP) and the kernel with performance isolation (PIso).

- In the MP1 case of an otherwise idle multiprocessor, PIso provides good response time for the pmake job, comparable to SMP, by being able to utilize idle resources from other SPUs.
- In MP6, where the other users submitted five additional jobs to the system, PIso was able to isolate the single pmake job in its own SPU. The response time of this pmake job is 50% better than the regular IRIX case (SMP).
- In MP4+D where one of the users was dumping core causing contention for the disk, PIso was able to provide fairness (isolation), allowing the single pmake job to get its share of disk bandwidth in a timely manner. The response time of this pmake job is 52% better than the regular IRIX case (SMP). The response time shows an increase of 47% over the uniprocessor case (UP) because the pmake is now sharing a fixed resource with the core dump.

Chapter 5
Data Locality

The second major resource management issue addressed in this dissertation is data locality. Data locality is an important issue for shared-memory multiprocessors with the CC-NUMA architecture that have different latencies to access local and remote memory. As was explained in Section 1.2, this is the dominant architecture for larger, scalable shared-memory multiprocessors and it is likely to be used even for smaller ones in the future. On CC-NUMA machines, good data locality reduces memory-stall time, and is therefore important for good application performance. The problem becomes especially important when the machine's workload is constantly changing because of the frequent entry and exit of applications, as would be characteristic of compute-servers. For such workloads, the operating system will periodically move processes between processors for maintaining load balance and fairness. Any mechanism for data locality has to respond to and cooperate with such process moves, in addition to responding to changes in the working set of applications with a large memory footprint, and changes in the memory access patterns of parallel applications.

The solution to the data-locality problem has two parts, process scheduling and data movement. Process scheduling decides where and when processes are scheduled, and this determines where the cache-misses originate. Data movement done by the virtual memory system (VM) decides where the data is located in physical memory, and this determines where the cache-misses will be satisfied. In the context of data locality, these two functions are closely interrelated. Process scheduling needs to carefully consider the location of data belonging to a process in the cache and memory hierarchy. Data movement needs to carefully consider where the processes accessing a particular page of data are running.

Process scheduling on CC-NUMA multiprocessors has been studied earlier by us [CDV+94] and will only be briefly discussed here. The main focus of the data-locality part of this dissertation will be on how the operating system can improve data locality, once a process has to be moved or in response to changing access patterns in the application. The operating system deals with memory at the granularity of pages. Therefore, we will study how the VM system can use page migration and page replication to provide good data locality for applications. Because the hardware provides a single physical address space and cache coherence, the migration and replication of data is only a performance optimization and not a correctness issue. We will show that good data locality can be maintained and application performance improved using a rational policy for data movement and a careful implementation to reduce costs.

The migration and replication of pages has been studied for earlier nonuniform-memory-access shared-memory multiprocessors [ScD89][BFS89][CoF89][BSF+91][LEK91] [Hol89][LHE92]. These systems either had no caches, or did not cache remote or shared data because they did not have hardware cache coherence. The techniques developed in these studies have influenced our work, but their results are not applicable to and cannot be extrapolated to current cache-coherent systems. These works are discussed in more detail in related work in Chapter 7.

The organization of this chapter is as follows. Section 5.1 briefly describes earlier work dealing with process scheduling issues, mainly covering the areas where there is interaction with the VM system for migration and replication. In Section 5.2, we develop a framework for understanding the problem of moving data and an abstract policy for making page movement decisions. In Section 5.3, we discuss the mechanisms for counting cache misses that will provide the information required for page migration and replication. In Section 5.4, we discuss the implementation of the policy in the kernel. The implementation requires both new functionality and performance tuning of parts of existing functionality. In Chapter 6, we will demonstrate the effectiveness of our solution to the data-locality problem by running workloads on the policy and kernel implementation presented in this chapter.

5.1 Process Scheduling

Process scheduling on multiprocessors plays an important role in providing good performance for applications, both sequential and parallel. This is in addition to its basic function of providing fairness and processor load balance. By scheduling processes carefully, the number of cache-misses a process suffers can be reduced, thereby improving the performance of the application. The choice of scheduling schemes for parallel applications can also have a big impact on performance, and is dependent on other issues too, such as the ability to do data placement and the shape of the speedup curve for the application. Process scheduling issues on CC-NUMA multiprocessors are discussed extensively in earlier work by us [CDV+94], and will not be discussed in detail here. We will briefly describe a few of the important process scheduling issues for CC-NUMA architectures, and how they interact with the VM support for data movement.

For sequential applications, strong affinity scheduling aids application performance in two ways. The first benefit to applications is the reduction in the number of cache misses going to memory. In affinity scheduling, the location in the cache and memory hierarchy of the data belonging to a process plays an important part in picking the processor to run a given process. The key issue is to place the process so as to reuse cached data close the processor. There are different forms of reuse that can be exploited by affinity scheduling.

- *Cache affinity* scheduling favors processes that might have state in the processor’s cache. These are the process that last ran on the particular processor, and the processes that have not run on another processor since they ran on this one.
- *Cluster affinity* is beneficial on a clustered architecture such as DASH [LLJ+92], with more than one processor per memory. When scheduling a process, cluster affinity favors one of the other processors in the cluster if the process cannot be scheduled on the last processor that ran it. This allows quick access to data that may be in various cluster-level caches.
- *Neighborhood affinity* is beneficial on architectures whose remote access time has a soft slope, i.e. the access time is shorter for nodes that are closer. The affinity function may be somewhat more complex and will depend on the topology of the interconnection network.

The second benefit to applications is that affinity scheduling can be used to keep processes from switching processors and local memories too frequently. We call this memory affinity, and it is required to obtain any benefit from the movement of data. Once the cost is incurred to move data to the memory local to a process, it is important that the process be able to benefit from the local data before the process is moved away. Frequent movement of processes away from their data will also require frequent data movement for data locality. Therefore, page migration and replication would not be profitable in situations where a process moves frequently. Therefore, for our data movement studies we assume that cache and memory affinity scheduling is required, and our experimental runs are done with affinity scheduling as described in [CDV+94] implemented in the kernel.

For parallel applications the interaction is more complex. Many parallel applications are written with the assumption that they are running stand-alone on a multiprocessor system. These applications do careful data placement in different memories to provide good data locality. However, when these parallel applications are included as a part of a multiprogrammed workload, the stand-alone assumption and careful data placement can easily break down. *Gang scheduling* [Ous82], a time-multiplexed parallel scheduling scheme, tries to maintain the illusion of stand-alone execution by scheduling all the processes of an application at the same time. However if all applications do not utilize the maximum parallelism of the machine, then application processes may need to be moved periodically for load balancing purposes. Any such movement can invalidate static data placement. The data-placement problem gets worse for *process control* [ABL+91] [TuG91], a space-multiplexed scheme, where each application is allocated a subset of processors on the system and the applications changes the number of active processes to match the available processors. Now the operating system can potentially change both the number and location of processors assigned to a parallel application in response to changes in the workload mix. For the above dynamic conditions and for applications where data placement cannot be inferred easily, the automatic migration and replication of data done by the operating system becomes important for parallel applications too.

5.2 A Framework for Providing Data Locality

In order to implement a data-movement scheme, we first need to define the problem carefully. The cache coherence in CC-NUMA systems provides correctness regardless of the location of the data, therefore the problem becomes an optimization one and needs to be carefully analyzed in terms of costs and benefits. The careful definition of the problem provides a basis for a policy structure that will allow the operating system to automatically decide when to migrate or replicate a page. This policy structure can then be translated into an implementation.

5.2.1 The Problem Definition

Data movement of pages to local memory is not needed for correctness and is only a performance optimization. The cache coherence provided in CC-NUMA systems ensures the correctness of data regardless of whether it is accessed from local or remote memory. Because data movement is a performance optimization, we need to make careful decision when moving data to ensure that the costs don't overrun the benefits.

Our goal is to minimize the runtime for the user's workload by reducing the component due to memory stall. On CC-NUMA machines, memory stall can be reduced by converting remote misses to local misses through the migration and replication of pages. The problem is that we have to first pay the cost to move data, and then hope that we will obtain enough benefit from the improved data locality to cover the cost of having moved the page. To maximize the overall reduction in execution time, we first need to be able to pick the appropriate pages to move by predicting future access patterns, and then keep the costs of doing migration and replication as low as possible to maximize the overall benefit.

The first aspect, improving data locality, involves finding the pages that suffer the most remote misses and converting them to local misses through data movement, if possible. The *access patterns to pages* can be broadly classified into three groups, and this determines the policy action to be taken for that page.

1. The first group consists of pages that are *primarily accessed by a single process*. These pages are candidates for migration when the process accessing them migrates to another processor. They include data pages of sequential applications, data pages of parallel applications where the accesses from the processes are to disjoint sections of the data, and the code of sequential applications when only one instance of the application is running on the machine.

2. The second group consists of pages *accessed by multiple processes, but with mostly read accesses*. These pages are candidates for replication. They include code pages of parallel applications, code pages of concurrently executing copies of a sequential application, and read-mostly data pages of parallel applications.

3. The third group consists of pages *accessed by multiple processes, but with both read and write accesses*. These pages are not candidates for either replication or migration. They cannot be replicated because there are writes to the page, and they will not benefit much from migration because they are being accessed by multiple processes. This group includes the data pages of parallel applications where there is fine grain sharing with updates from multiple processors.

The second aspect of the problem concerns the costs of page migration and replication. We classify these costs into four categories.

1. The first cost is that of *gathering information* to help determine which pages to migrate or replicate, and what actions need to be taken for the page.

2. The second cost is that of the *kernel overhead* for migration and replication. This cost includes the overheads for allocating a new page, removing all mappings to the old physical page, establishing the new mappings, flushing TLBs to maintain coherence and eliminating replicas on a store to a replicated page, to name a few.

3. The third is the cost of *data movement* to physically copy a page from one location to another.

4. The fourth is the indirect cost of *increased memory use* (memory pressure) resulting from page replication. It is important to keep all these costs low, and we will describe the optimizations done later in this thesis.

5.2.2 Policy Structure

For the operating system to improve data locality, it needs to automatically figure out which pages to move, and what to do with each of these pages, migrate or replicate. Figure 15 shows a decision tree that will enable the OS to automatically implement a data movement policy. We assume that the decision process may be triggered on any cache miss. Parts of this decision tree are similar to that used in earlier studies on non-cache-coherent NUMA machines, but ours accounts for reuse of cached data, and is driven by cache misses not memory references.

The first step (node 1) is to determine the pages to which a large number of misses are occurring, the “hot” pages. We are using this trend as an indicator of the future, hoping that the current hot pages will continue to suffer many misses. Considering the page-movement costs, this will maximize the benefit of moving the page into local memory. If the page is taking very few misses, then we are not interested in considering any action for the page.

The second step (node 2) is to determine the type of sharing that applies to the page in question. The OS needs to automatically figure out the page-access pattern that best characterizes a page of interest. If the page is referenced primarily by one process then it is a candidate for migration. If on the other hand it is being accessed by multiple processes, then it is a candidate for replication. Therefore, on the basis of the sharing pattern, we take the replication (high sharing) or the migration (low sharing) branch.

The third step (nodes 3a and 3b) is to help control the overhead cost under different conditions. We do not want to replicate a shared page that is being written often. A replicated page would have to be collapsed at every write to maintain consistency. This would be an unacceptable overhead. We also do not want to replicate pages if the system is under memory pressure. If we do not control for memory pressure when replicating, we may introduce paging unnecessarily and potentially make performance worse. It is possible for the policy to frequently decide to migrate a page because of its sharing patterns. We need to control for such a policy decision by migrating a page only if it has not been migrated too often in the past.

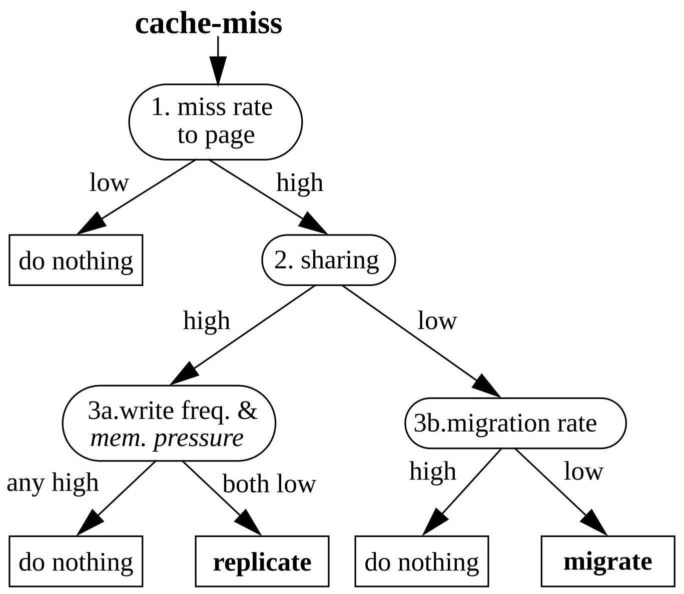

FIGURE 15. Replication/Migration decision tree. The flowchart shows the decision process for a page to which a cache miss is taken. The possibilities are to replicate, migrate, or do nothing.

5.3 Cache Miss Counting

In the policy described above, the decisions are driven by cache misses, therefore cache-miss count information is at the core of our data-locality solution using migration and replication. The counts serve two purposes. First, they help identify the “hot” pages. These are pages that are the destinations of cache misses from remote processors. This is important because data locality is only a performance optimization. Therefore, data movement should address only the busiest pages, which are the ones that are likely to yield maximum benefit for the cost incurred. Second, a per-processor count identifies potential recipients of the page or a copy by tracking the processors that are currently suffering cache misses to this page. Also, a comparison of the counts from different processors indicates whether a page is shared, driving the decision of whether to migrate or replicate a particular page, as described earlier.

Although we are concerned about cache-misses, it is possible to approximate cache-miss information using other metrics. In Section 6.5.1, we will discuss the effectiveness of another metric, TLB misses. We will show that they are not effective in all cases, or as in the case of hardware reloaded TLBs not visible. Therefore, efficient cache-miss counting becomes crucial to this implementation.

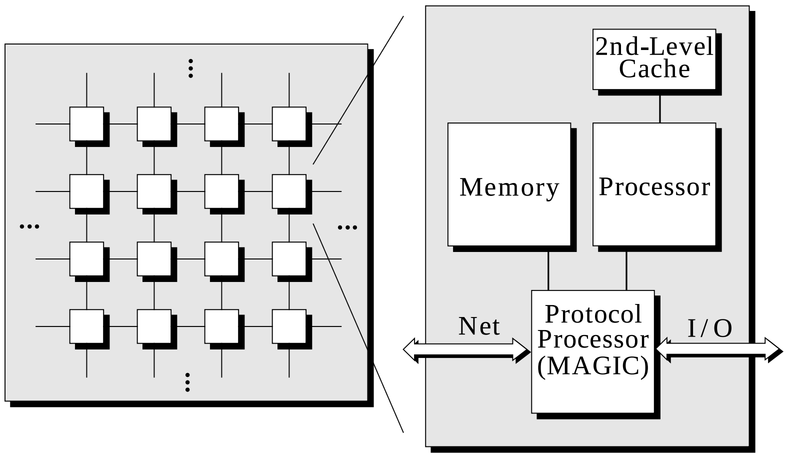

FIGURE 16. The Stanford FLASH multiprocessor. In the FLASH multiprocessor, the MAGIC chip connects the processor to local memory, remote memory and I/O. MAGIC has a protocol processor that runs short sections of code (software handlers) to service requests from local and remote processors.

The natural choice for cache-miss counting in CC-NUMA systems is the memory controller because it is already doing work on every cache miss to maintain cache coherency in these systems. There are two possibilities for mechanisms to count cache misses, based on the architecture of the memory controller. One possibility is to do the cache miss counting using hardware counters as done by SGI Origin systems [LaL97], where the memory controller has a hardwired architecture. A counter per processor is kept for each physical page in local memory. For every cache miss the appropriate counter is incremented by the hardware.

The other possibility is to count the misses in software, as we did in the FLASH machine [KOH+94]. The FLASH machine is shown in Figure 16. The core of the directory controller in the FLASH machine (MAGIC) is a programmable statically-scheduled dual-issue processor that runs software handlers to service cache misses. We added code to the relevant handlers to count the misses from each processor to each page of memory. The counting code is added to the end of the handler, and is run only after all necessary processing is done and any needed reply sent to the processor. Therefore the latency of the handler for servicing the miss is practically unchanged, and only the run time of the handler is increased. To further reduce the impact of cache miss counting on the system, we use sampling, and run the counting code only once every ten invocations of the handler. The sampling frequency could be made even larger if required without loss of accuracy. Our measurements show that the inserted cache-miss counting code has no measurable impact on the latency of cache misses or the application execution time. In Section 6.5.1 we will show that using sampled cache misses is as effective as full cache-miss information. The counters are periodically reset to approximate a cache-miss rate and reset stale information. The reset is done by a handler that runs periodically when the controller is idle.

Our implementation maintains a set of counters per physical page, with the appropriate counter updated every time there is a miss to the page. When the counter for a remote processor crosses a configured threshold, the page is considered hot. The page number is saved, and an interrupt is generated to the node that crossed the threshold. The rest of the processing is done by the interrupt handler in the operating system. The counters are stored in memory, and this incurs a space overhead because the policy requires one counter for each processor for each page. On the eight node system we evaluate, with one byte counters and 4 Kbyte pages, this is a 0.2% overhead. On larger systems, for example with 128 nodes, this would be a 3.1% overhead. With sampling, the maintained counts and thresholds are smaller, and the space overhead is reduced to 1.6% of memory by using half-size counters. Also, we require only one counter for all the processors associated with a memory. Therefore, in clustered architectures with multiple CPUs per memory fewer counters are required. For example, on DASH or Sequent STiNG with four CPUs per memory, eight counters per page would be sufficient to track 32 processors.This overhead is to be contrasted with a 7% memory overhead for the per cache-line directory information that the controller already keeps to maintain cache-coherence in FLASH. In Section 6.6 we will discuss the issues in scaling this solution to larger numbers of processors.

In the FLASH implementation we also maintain a bit per page to mark if the page has been written recently. This bit is important for detecting read-mostly data pages for replication. These are pages that are written very infrequently (just initialization), and are mostly accessed through reads. There is OS-level information that identifies read-only regions, such as program text. By using the write bit to identify other read-mostly pages, we are able to increase the space of pages that could be replicated.

5.4 Kernel Implementation

The rest of the implementation of the migration and replication functionality is in operating system code. For our experiments we modified the SGI IRIX 5.3 operating system to implement the

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

TABLE 7. Key parameters used by the policy. These parameters are used along with the counters to approximate rates. The counters used by the policy include a per-page per-processor miss counter, a per-page write counter, and a per-page migrate counter.

page migration/replication policies. IRIX is an SMP OS that runs on the SGI CHALLENGE, a bus-based multi-processor system. The main changes were:

- The addition of the interrupt handler that responds to the hot page interrupt from the controller, and implements the policy.
- Additional functionality added to the VM subsystem needed for the implementation of the policy.
- Important performance tuning changes required because of the additional stress on the VM system due to migration and replication related activities.

The performance overhead of these changes, when not migrating or replicating pages, was insignificant (less than 0.5% of non-idle execution time, except in certain pathological cases for kernel-intensive workloads because of the poor scalability of the IRIX5.3 kernel.).

5.4.1 Policy Implementation

To implement the policy, we first need to translate the abstract decision tree developed in the previous section into concrete parameters that can be observed in the system. It is difficult to track rates, therefore we approximate them using counters with a periodic reset. For each page there is a *miss counter* per processor, a *migrate counter*, and a *write counter*. Table 7 defines a *reset interval* parameter and the *trigger*, *sharing*, *write*, and *migrate* thresholds that together with the counters approximate the rates in the decision tree.

- The *reset interval* is the period in which a cache-miss counter is reset.

- A page is considered hot when a cache-miss counter for a processor reaches the *trigger threshold*; if that page is in a remote memory, further action is considered.
- If the miss counter for this page on any other processor has exceeded the *sharing threshold*, we consider the page to be shared, otherwise the page is not shared. A shared page is a candidate for replication and an unshared page is a candidate for migration.
- A candidate page is replicated only if the write counter has not exceeded the *write threshold*. The write counter as mentioned earlier is kept by the memory controller. In practice it is only one bit and so the write threshold is effectively one.
- A candidate page is migrated only if the migration counter has not exceeded the *migrate threshold*. The migrate counter and threshold functionality is implemented in the operating system by tracking when a page was last migrated.

The directory controller generates an interrupt when a page crosses the trigger threshold. The interrupt is delivered to the node whose counter just crossed the threshold. Another option is to interrupt the node that is the home of the page. We tried both options, and found that the latter has some clear drawbacks. The node that just crossed the threshold potentially has many of the cache lines from the page in its cache. Also the new page, after migration or replication, will most likely be local to this node. Hence the page copy can be done with minimal number of remote misses. Interrupting the node that just crossed threshold rather than the home node also spreads the overhead of migrating and replicating pages across the nodes, potentially avoiding a hot-spot on a node that may be the home of many pages of interest to many other nodes.

The flow chart in Figure 17 describes the low-priority interrupt handler, which services this pager interrupt by replicating or migrating pages, if necessary. The eight steps in the process are described next.

1. As described earlier, the memory controller counts cache misses from processors to pages. When a cache-miss counter for a remote processor crosses the trigger threshold, the memory controller interrupts the remote processor.

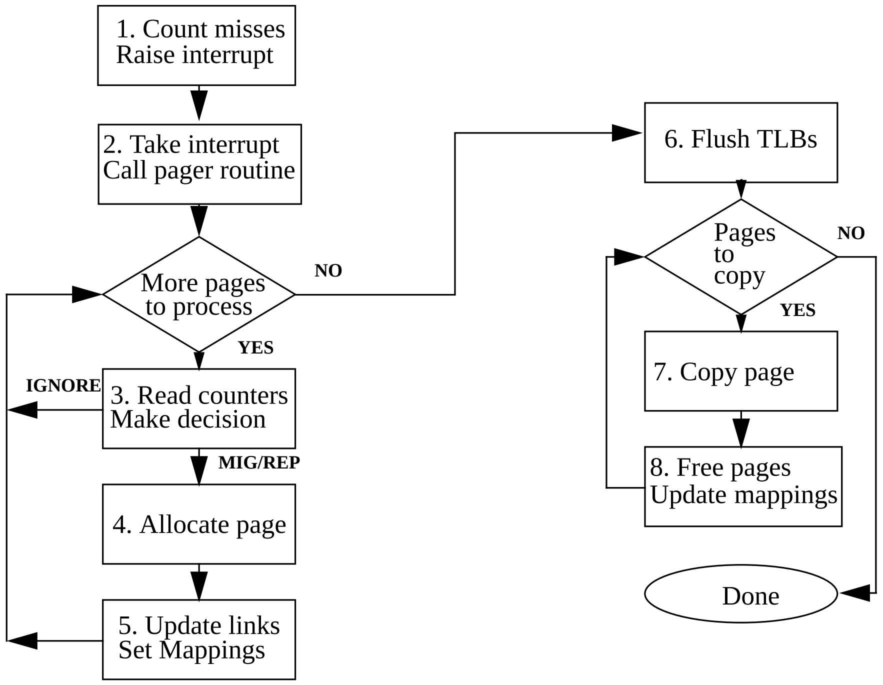

FIGURE 17. Flow chart for migration or replication. The flowchart shows the steps for the migration or replication of a page in the kernel implementation. A page collapse goes through only a subset of these steps and is not shown.

2. The processor takes the interrupt. The interrupt service routine reads the physical page number from the memory controller using an uncached operation, and stores it in a data structure. It then sets a low priority software interrupt that will trigger the pager routine that implements the page movement code.

3. The pager routine obtains the appropriate locks for the page and its replicas if necessary. It then reads the relevant counters and write information for the page. The counter information along with past migration information, memory availability information, and the relevant thresholds are used as inputs to the policy module. A decision is made to migrate or replicate the page or to do nothing. In some cases, there is already a copy in the local memory, and all that is required is to set the appropriate page table entries to point to this page. In this case or when the policy decision is to do nothing, the remaining steps are skipped.

4. If a migration or replication is to be done, a new page is allocated from the appropriate memory. The cache-miss counters for this page are reset. This page is also locked.

5. The new page is now linked into all the appropriate data structures maintained by the OS. If it is a migration, the old page is removed from VM data structures in preparation for freeing. The page table entries are changed to point to the new page, but are marked invalid to prevent access or changes to the page during copying. The old page is now no longer accessible by any process, except from processors that have mappings in the TLB. If it is a replication, the new page is linked into a chain of replicas. If it is the first replication for the page, all mappings to the page and its replica are changed to be read-only to prevent writes to replicated pages. Now the page cannot be changed by a faulting processor, because the page is locked.
6. The TLBs on all processors are flushed to remove all TLB mappings to the current page. This TLB-flush action is taken for all migrates and the for the first replicate of a page, as the existing mappings either point to the wrong page or do not have the read-only protection set. If a processor now tries to access the page, it will get a TLB fault. The entry loaded in response to this fault is either invalid (migration) or read-only (replication). This prevents writes to the page while the data is being copied.
7. The data is now physically copied to the new page. The copying mechanism used in our current implementation is word by word copy without pre-fetching. The benefit from using pre-fetching or a hardware-copy engine will be architecture dependent.
8. Once the data is copied, old pages are freed in the case of migration. Finally, the page table entries are set correctly; valid in the case of migration and to point to the appropriate physical copy for replication. All pages are unlocked.

As mentioned earlier, data locality is just an optimization. Therefore the implementation is optimistic, and can choose not to take any action on a page if there is a chance of spinning or blocking. In step 3, the page will be ignored if the page lock cannot be acquired immediately. In step 4, the page will be ignored if there is no page available in the appropriate memory.

Taking interrupts and flushing TLBs can be a large overhead as will be discussed later. To reduce the per-page overhead for taking the interrupt and flushing TLBs, the directory controller attempts to collect multiple pages before generating an interrupt (in our implementation two pages). Therefore, the interrupt handler code iterates over steps 3 to 5 for each page. Then a single TLB flush operation is done, followed by steps 7 and 8.

There is an additional page collapse function that is required for correctness. The page collapse is invoked on a write access to a replicated page. The page table entries for replicated pages are marked read-only, and a write to a replicated page causes a trap that vectors the processor to the protection fault handler (pfault). This handler recognizes that this is a write to a set of replicas and calls the page collapse routine. The replicas are collapsed to a single copy, the page-table mappings are updated to point to this copy and made writable, the TLBs are flushed to remove old mappings for this page, and then the write is allowed to proceed.

5.4.2 New Kernel Functionality

Additional functionality had to be added to the IRIX VM system to enable the implementation of the policy just described, primarily support for replicas and back maps.

Replica Support: In IRIX a hash table is used to translate logical pages designated by a vnode and offset to physical pages. This is an open hash table indexed by vnode and offset, and physical page frame descriptors (PFDs) are linked on hash chains. When a page fault occurs and a mapping is not found in the page table, this hash table is used to establish the mapping from the virtual address of the fault to the physical address of the page that contains the data. This mapping is entered into the page table to allow fast reloads of the TLB in the future.

With replication, support was needed to track all the replicas of a physical page. There are many places where this functionality is needed: when a page fault occurs from a processor it is important that a local copy be found for the page, and a mapping to it returned if it exists; a set of replicas will need to be collapsed when a write occurs to any one of the copies; and all the replicas of a page need to be identified when creating a new replica.

The changes to the hash table for supporting replicas are shown in Figure 18, along with the original IRIX structure. An extra pointer field was added to the PFD structure, and this was used to singly link the replicas of a page in a chain. One of the replicas is designated as a master and is linked into the hash chain like a regular page, thus allowing access to any replica of the page. All replicas also have a pointer to the master PFD. This makes it possible to find the master and traverse the replica chain starting from any replica, a needed function in the implementation. This pointer is the same as would be used to link to the next PFD in the hash chain, so no extra storage is used.

The paging code was also changed to reclaim replicas sooner than unreplicated pages. A cost of replication is the increased memory used by the replicas. If this increased memory use were to cause other required pages to be swapped to disk, the overhead for getting these pages back might negate any benefit from improved locality. To avoid this problem, replicas of pages are aged faster than unreplicated pages. Normal pages are aged down from a count of seven. The count is reset when a page is accessed and if it reaches zero the page is reclaimed. Replicas are aged down from a count of two. Therefore, replicas may be reclaimed earlier if the system is running low on memory, avoiding the paging of other pages if possible.

Back Mappings: In IRIX, as in most operating systems, there is a clear path to translate from virtual addresses to physical address. A linear page table is used with page table entries (PTEs) for virtual addresses pointing to PFDs. However, there is no link from the PFD directly back to the PTEs that are pointing to it. This is not functionality that is needed in normal kernel activity. With the implementation of migration and replication this back mapping functionality becomes important. The cache-miss counting and interrupt mechanism deliver a physical page number to the operating system. The operating system then needs to find the page table entries that point to this physical page and all its replicas. This is required by the policy implementation to invalidate mappings, to change mappings to read-only, or to change the mapping to point to a different replica of the page.

The implementation of the back mappings is shown in Figure 18. Each PFD has a chain of links, each of which points to a page table entry. These page table entries map a virtual address to this physical page frame. For each PFD, the chain includes all entries mapping the physical page corresponding to the PFD. The back map pointer structures are allocated dynamically when required, when establishing a new virtual to physical mapping.

5.4.3 Performance Optimizations

In addition to the functionality changes just described, there were a number of kernel changes for performance optimization. These are fairly crucial to the success of the data-movement imple-

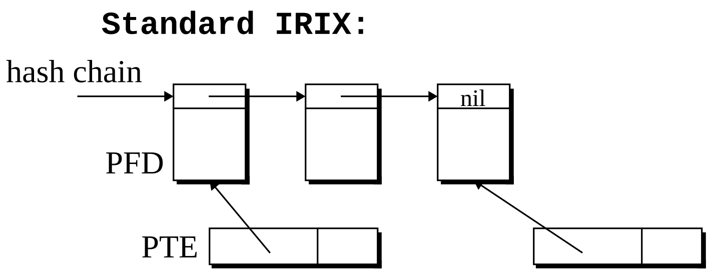

FIGURE 18a. Standard IRIX hash table structure. In IRIX a hash table is used to translate logical pages (vnode, offset) to physical pages. Physical page frame descriptors (pfds) are linked into this open hash table through the hash chain. Also, virtual page table entries (ptes) point at the pfds, but 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.

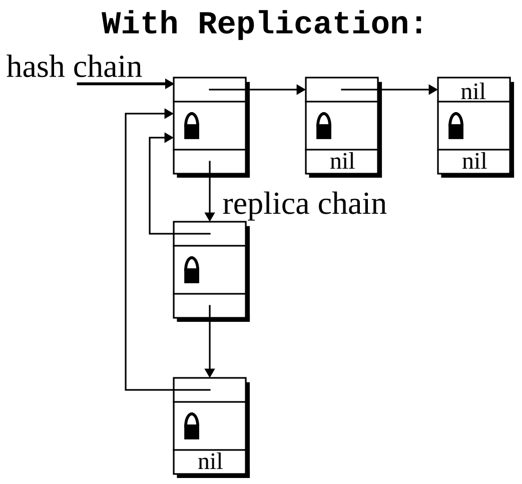

FIGURE 18b. Hash table structure with replication. All replicas of a page are linked on a replica chain. Only one of the replicas is linked into the hash table. Individual page level locking was added to lock the replica chain. Memlock is now only used for changes to the hash chain and for allocation and deallocation of page frames.

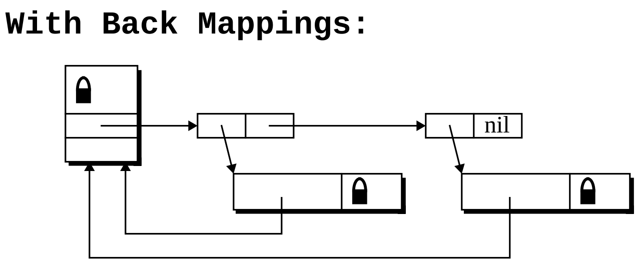

FIGURE 18c. Hash table structure with back mappings. To facilitate easy mapping changes, links were added to the pfd pointing back to all the ptes mapping this page. pte level locks were added so that the region lock did not have to be grabbed during a migration or replication operation. (Although the region still exists to protect other resources.)

mentation because keeping overheads low determines how much overall benefit is obtained or if there is any benefit at all. The big issues are finer grain locking, reducing interrupt overhead, and reducing TLB flushing overhead.

Finer grain locking: The version of IRIX we used (5.3) had fairly coarse locking for VM related structures. There is one lock (`memory_lock`), which protects the global hash table of active physical pages, the global free page lists, the fields of the PFD structures, and a number of smaller VM structures. This lock, `memory_lock,` is acquired before doing most VM related actions, and is a source of contention for workloads that are kernel and VM intensive. There is also one lock per memory region of a process (`region_lock`) that protects the page-table entries and mappings for that region. This lock is acquired when changing page table entries, and can lead to contention for memory regions that are shared by more than one process. Both `memory_lock` and the region locks for shared text or data regions were potential performance bottlenecks in our implementation because of higher resulting VM system activity and the need to manipulate page-table entries more frequently when doing data movement.

Completely changing the locking structure of the VM system is a huge kernel change. Instead we implemented a smaller, though still considerable, kernel change to provide finer grain locking and reduce the contention for locks in the VM system. This implementation is shown in Figure 18. To reduce the contention for the `memory_lock`, page-level locks were added. This is a lock in every PFD structure, implemented as a bit with LL/SC based lock and unlock functions. In most data movement scenarios, `memory_lock` is only held till the appropriate page-level lock is acquired. A similar locking bit was added to every page-table entry to protect the virtual to physical mapping in that entry. The `region_lock` is only held till the PTE lock is acquired, so reducing contention for the region locks when changing mappings during data movement operations.

A locking hierarchy was established for these locks to avoid deadlock. The order of lock acquisition is `memory_lock`, the page lock for the master page in the case of replicas, the page lock for regular pages, and the PTE lock for the mappings. The page lock of the master page in a set of replicas also protects the linked list of replicas. The page lock of any page also protected the linked list of back mappings to that page. These changes reduced the synchronization cost in our implementation, and were needed to see any benefit from data movement. The details of synchronization overhead in the implementation will be given in Section 6.4. Finer-grain locking in the VM system has been added to newer versions of IRIX.

Interrupt Overhead: The memory controller interrupts the processor when a cache-miss counter crosses the trigger threshold. Depending on the access patterns some of the pages may not be migrated or replicated, and so those interrupts are “wasted”. The overhead to process this interrupt should be made as small as possible for maximum benefit. To reduce this overhead, the directory controller collects multiple hot pages before generating an interrupt. In our experiments an interrupt is generated for every two pages, therefore the cost of processing the interrupt is effectively halved. The details of interrupt processing overhead in the implementation will be given in Section 6.4. Collecting multiple pages is a trade-off between amortizing the interrupt cost across multiple pages and waiting too long to collect the multiple pages.

TLB Flushing: An important part of the overhead of migrating and replicating pages is the overhead of flushing TLBs to maintain the consistency of page table mappings. The existing path to flush TLBs is used quite infrequently in IRIX and had too high an overhead for our use. The TLB flushing mechanism needed to be optimized because data movement requires that TLB flushing be done much more frequently. Our implementation took three steps to reduce this overhead from about 175μs as reported in [ENC+96], to about 48μs for eight processors.

First as described in Section 5.4.1, all currently outstanding hot pages are preprocessed before doing a single TLB flush. This amortizes the cost of the TLB flush across all the outstanding pages.

Second, the communication and synchronization required to request and get an acknowledgment for a flush from a remote processor was reduced considerably. The existing mechanism required both the requesting and the acknowledging processor to acquire locks on structures. In the modified scheme, a processor requesting a flush raises a global count and the sends interprocessor interrupts (IPIs) to all the processors. A processor receiving the IPI compares the global count with a local count. If the local count is smaller, it raises its local count to the global count and flushes its TLB. The requestor waits till all the local counts have reached the desired global count before proceeding with the rest of the policy. At this point it is known that all the processors will have flushed their TLBs before returning to user space.

Third, the code path to take the IPI and call the TLB flushing code is fairly long, and requires saving all the registers, and demultiplexing the interrupt. This path was made considerably shorter through an early check for the TLB flush request in the interrupt service path. If this is the case, the actual flush is done in assembly code without saving any additional registers. After the flush, the interrupt is acknowledged and the code returns from the interrupt. Any other interrupts that are multiplexed on this line will be taken at this point. The other interrupts on this line are much less frequent.

5.5 Summary

In this section we have introduced the data-locality problem for multiprocessors with the CC-NUMA architecture. The solution to this problem has two parts, process scheduling and data movement. We focus on the data movement part of the solution, studying the feasibility of migration and replication of pages by the VM system.

On CC-NUMA multiprocessors data locality is a performance optimization. We therefore develop an appropriate policy framework to maximize benefits and keep costs low. The policy proceeds in stages:

- First, detect busy pages.
- Determine the access patterns to these pages from the processors — shared or unshared, read-only or read-write.
- Select the appropriate action for these pages to improve data locality — migration or replication.
- Keep the costs of doing data movement low.

The actual implementation of the policy has two parts. The first part is the counting of cache misses. Cache miss counts are used to drive policy decisions. We describe our implementation of cache-miss counting in software, in the memory controller of the FLASH system. By using sampling, the cost of cache miss counting is made negligible.

The second is the changes to the kernel required to implement the policy. There are three parts to this implementation. The interrupt service routine that responds to the interrupt from the cache-miss counting hardware and implements the policy. New functionality that was required for migration and replication, primarily support for tracking the replicas of a page and back mappings from physical pages to the page table entries pointing to them. Finally, performance optimization of critical code, primarily finer-grain locking in the VM system, lower interrupt processing overhead for interrupts generated as a result of the counting code, and lower cost for flushing TLBs.

Chapter 6
Experimental Results for Data Locality

The goal of this chapter is to experimentally demonstrate that automatic operating system based migration and replication can provide good data locality for dynamic workloads on CC-NUMA machines. In the process, we will also show that the data-locality solution that we developed in this dissertation provides an effective solution to the problem. To this end, we will run a number of workloads on our kernel implementation of the migration and replication policy, carefully analyze the benefits and costs of our implementation, and show the overall improvements in application performance that are possible.

This chapter is organized as follows. In Section 6.1, we will describe and characterize the five workloads that we use to obtain the performance results. In Section 6.2, we will discuss the improvement in data-locality seen by individual application, and the consequent improvement in execution time for each workload as a result of migration and replication. Section 6.3 highlights the system-wide benefit from improved data-locality, resulting from reduced contention for memory system components. In Section 6.4, we will analyze in detail the kernel overheads resulting from migration and replication, providing further insights into the kernel implementation. Finally in Section 6.5, we conduct a trace-based exploration of a number of issues related to the policy chosen and its parameters.

Name	Contents	Notes
Engineering	6 Flashlite 6 Verilog	multiprogrammed compute-intensive serial applications
Raytrace	Raytrace	parallel graphics application (rendering a scene) 256 Kbyte secondary cache
Splash	Raytrace Ocean	multiprogrammed, compute-intensive parallel applications 256 Kbyte secondary cache
Database	Sybase	parallel commercial database (decision support queries), four processor configuration
Pmake	3 four-way parallel Makes	software development (compilation of gnuchess)

TABLE 8. Description of the workloads. For each workload, the table lists the applications in the workload and a short description of the workload. The workloads are run on an eight-processor MP with 512 Kbytes of secondary cache, except where noted.

6.1 Workload Characterization

An important factor in a performance study is the workloads used. We considered a number of issues when selecting the five workloads for this study. They had to be realistic and representative of different uses of compute servers. There had to be some potential for improving data locality and performance improvement. They also had to represent different possible scheduling scenarios, both sequential and parallel, as discussed in Section 5.1. Table 8 summarizes the five workloads. Table 9 shows detailed CPU and memory system statistics for the workloads when run on the target CC-NUMA architecture using the first-touch page placement policy.

Multiprogrammed Engineering Workload (Engineering): The first is an engineering simulation workload, consisting of several large compute and memory-system intensive applications. There are multiple copies of two sequential applications. One is the commercial verilog simulator VCS, simulating a large VLSI circuit; the chip design for the MAGIC memory controller for the FLASH multiprocessor. VCS compiles the simulated circuit into code, and the resulting large code segment causes a high user instruction stall time. The other application is Flashlite, a functional simulator of the FLASH memory system. In this workload Flashlite is simulating a four processor system running the FFT parallel application from SPLASH [SWG92].

This is a multiprogrammed workload, scheduled by UNIX priority scheduling with affinity [VaZ91]; there are six VCS and six Flashlite applications on an eight processor system. This workload spends very little time in the kernel. User stall time is a large component of non-idle execution time, about 70%. This stall time is divided about equally between instruction and data stall. Poor data locality can significantly impact the performance of such workloads because they spend a large fraction of their time in data stall. Therefore, the potential benefit from data-locality is large.

Single Parallel Application (Raytrace): This workload consists of Raytrace [WOT+95], a single compute-intensive parallel graphics algorithm widely used for rendering 3-D scenes. In this workload, Raytrace starts eight processes on the eight processors system. The processes are locked to individual processors; a common practice for multiprocessors running single applications. The configuration of this workload is also representative of one application in a multiprogrammed gang-scheduled workload. If this were part of a gang-scheduled configuration, there would be additional cache misses and memory-stall time because of cache interference from other applications.

In this workload, the main data structure is the representation of the scene to be rendered. The parts of this data structure accessed by each processor depends on the characteristics of the scene being rendered. Therefore, it cannot be inferred by the programmer, and it is infeasible to distribute this data statically among the multiple memories to improve locality.

The working set for Raytrace fits in a 512 Kbyte secondary cache, even with the largest input scene (`car.env`) from the SPLASH suite. The amount of memory stall time seen by an application is an important determinant of the performance improvement from better data locality. The larger the memory stall time, the larger the potential benefit from improved data locality. As the size of the cache increases relative to the working set of the application, the fraction of time spent stalled for memory access decreases. Therefore, we scaled the cache size down to 256 Kbyte because we did not have access to a larger input scene. User memory-stall time is now 36% of the non-idle execution time, mostly because of data stall.

Multiprogrammed Scientific Workload (Splash): The third workload consists of two parallel applications, Raytrace and Ocean [SWG92]. Raytrace was previously described. The OCEAN program simulates large-scale ocean movements based on eddy and boundary currents. We use the version that implements the grids to be operated on with 3-dimensional arrays, allowing contiguous allocation of the partitions that each processor uses. Ocean has only nearest-neighbor communication, and this results in mostly unshared read-write access to data.

This is a multiprogrammed workload of parallel applications, and we simulate the behavior of a space-partitioning approach, similar to scheduler-activations [ABL+91] [TuG91], for scheduling the jobs. Instead of implementing a general space-partitioned scheduler in the kernel, we modified the applications to catch the USR2 signal, and respond by reading a file to identify the processors allocated to it. We then generate the USR2 signal at fixed points, such as application start and end, to force changes in both the number and position of processors assigned to each application.

As discussed in Section 5.1, it is important to have data movement to maintain data locality with space partition scheduling. As processes of applications get rescheduled, their data will need to be migrated to the new locations. As this workload has parallel applications like Raytrace, replication will also be required. This workload spends about 34% of its time in user stall, mostly because of data misses, as shown in Table 9.

Decision Support Database (Database): The fourth workload is a parallel database system running a set of decision-support queries to an off-line main-memory database. We use the Sybase database server that supports multiple engines, but has not been optimized for NUMA architectures or to scale to larger parallel systems. Because of the scalability problem, this workload is run on a four-processor system, whereas all the other workload were run on an eight-processor system. There are four database engines, each locked to a processor.

The database workload has a large user stall time, 63% of non-idle execution time, primarily from data misses. This workload was chosen, expecting benefit from replication of key read-only structures such as the indexes on various tables. In our experiments this did not happen. However, the workload turns out to be interesting for other reasons, as we will explain while discussing results in Section 6.2.4.

Multiprogrammed Software Development (Pmake): Our final workload represents a server being used as a compilation engine for a software development group. The workload consists of four parallel make jobs, each compiling the gnuchess program with four-way parallelism. The workload is I/O intensive, with a lot of system activity resulting from many small short-lived processes, such as compilers and linkers. UNIX priority scheduling with affinity is used.

Workload	Cumulative CPU Time (sec)	CPU Time Breakdown (%)			User Stall Time (% Non-Idle)	
		User	Kern	Idle	Instr.	Data
Engineering	49.9	73	7	20	35	32
Raytrace	32.8	77	8	15	6	30
Splash	45.9	62	11	27	5	29
Database	31.2	58	5	37	3	60
Pmake	23.0	23	56	21	3	6

TABLE 9. Execution time statistics for the workloads. The table shows the execution time of the workload, the percentage of the execution time spent in Idle, Kernel and User modes, the percentage of non-idle time spent stalled on the secondary cache for instructions and data.

This workload spends a large fraction of its time in the kernel (56% of total execution time) and is idle for 21% of the time. The kernel is IRIX 5.3 is run out of unmapped space. Therefore, our data movement does not apply to kernel code or data, and so cannot do much to improve its locality. Only about 10% of non-idle execution time is because of user stall time. Consequently, this workload will not benefit much from data movement, and was selected to see if such a workload is adversely affected by our data movement implementation.

6.2 Execution Time Improvement

We now run the five workloads on a kernel that was modified to implement our migration and replication policy. As a basis for comparison, we will also run the workloads with a *first-touch page placement policy.* First-touch is a simple NUMA-aware page placement policy, that allocates the page in physical memory local to the processor that first touches references the page. As discussed in Section 3.2.3, the workloads are run on an eight processor CC-NUMA machine simulated by SimOS. the ratio of local to remote cache-miss latency is 1:4. The memory-access latencies without contention are 300ns for a local memory access and 1200ns for a remote memory access. The *trigger threshold* in the policy is set to 128 misses, and the *sharing threshold* is set to 32 misses. These were experimentally found to give the best results for our configuration. The choice of trigger threshold also approximately matches the competitive properties of the algorithm in [BGW89], where a replication was triggered when the cumulative additional cost of the remote references equalled the cost of replicating a page. Here we replace references by misses because we have caches. The cost of 128 remote misses is 154µs at the contention-less remote miss latency of 1.2µs. The actual latency of a remote miss is somewhat higher with contention. Therefore, the cost of 128 remote misses (the trigger) is approximately the same as the cost of migrating or replicating a page, which we will show in Section 6.4 is about 200µs. The sharing threshold was set to a quarter of the trigger threshold. In Section 6.5.4 we will explore the sensitivity of the policy to these parameters.

Figure 19 shows the results of the runs. For each workload there are two bars. The first bar shows the performance with the first-touch page placement policy that places the page local to the first processor that faults on the page. The second bar shows the performance with the implemented migration and replication policy. The bars show the normalized non-idle execution time for each run. The execution time is normalized to that with the first-touch policy for that workload.

An important point to note is that we use non-idle execution time as a metric and not response time. We are primarily interested in how efficiently the applications are run, i.e. how little time they spend stalled for memory. This efficiency is best represented by non-idle execution time. Response time also reflects efficiency of execution, but for multiprogrammed workloads it is also affected by the scheduling order of processes.

Each bar consists of four parts: the base time that represents the time spent executing user instructions (without stall) and the kernel time present in the first-touch implementation; the user-stall time for cache misses from local memory; the user-stall time for cache misses from remote memory; and the kernel overhead to migrate and replicate pages. The number at the bottom of each bar shows the *total data locality*, which is the percentage of all cache misses (user and kernel) satisfied from local memory for the workload. Our policy seeks to improve one component of this, the *user data locality*, which is the percentage of user cache misses satisfied from local memory.

The improvement in total execution time because of migration and replication is dependent on three factors. First, the fraction of total execution time spent in user stall. Second, the increase in total data locality because of the migration and replication of pages. Finally, the kernel overhead for data movement required to achieve the better data locality. We will now examine the impact of migration and replication on each of the five workloads.

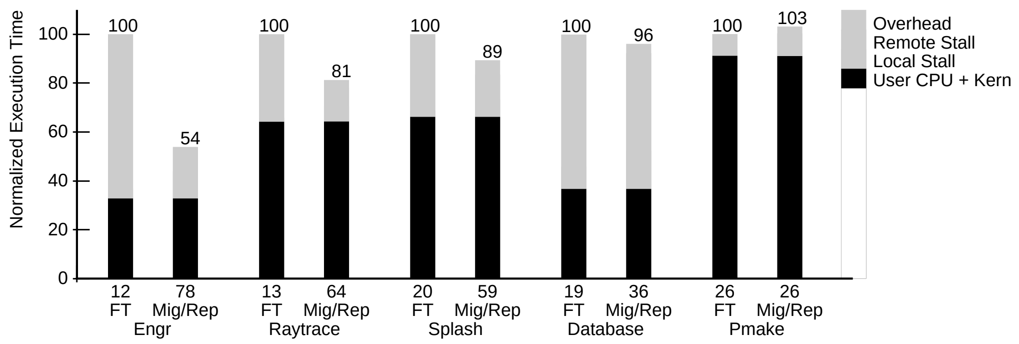

FIGURE 19. Performance improvement with migration and replication. We show the execution time for each workload with the base policy (Mig/Rep) normalized to that with first touch (FT). The execution time is divided into the kernel overhead for migration and replication, the remote and local stall times, and all other time. The number below each bar shows the percentage of misses to local memory.

6.2.1 Results for the Engineering Workload

The engineering workload shows the largest performance improvement with data movement; 46% improvement for non-idle execution time. With first-touch, only 12% of all misses (user and kernel) are serviced from local memory. This is only one in eight, which is no better than random on a system with eight PEs. This low data locality is to be expected in dynamic workloads where processes are moved between processors for load balance across the system. Through migration and replication of pages, the **user** data locality is 83%, and this increases the percentage of all misses (user and kernel) serviced from local memory to 78%. The engineering workload had a large user stall time, and the vastly improved data locality reduces this by 69%. The additional kernel overhead for data movement is quite small.

This workload benefits from both replication and migration. All the applications in this workload are sequential, and only code segments are shared. However, the code segment for the VCS applications is fairly large. Table 10 shows that 43% of the hot pages were replicated, primarily because of the code segment sharing of VCS. Replication of code pages has a large impact because I-cache stall time was 34% of non-idle time. Page migration primarily kicks in when a process is rescheduled on a new processor, and it migrates the hot pages in the process' working set to local memory. Page migration is more applicable to the pages of the data segment that are unshared, than to the pages of the code segment that may be shared with other instances of the same application, and therefore potentially replicated. In this workload, 34% of the pages are migrated. This workload demonstrates that both migration and replication are necessary to fully exploit memory locality.

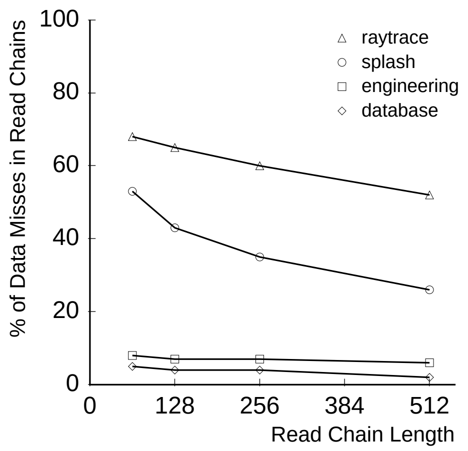

FIGURE 20. Percentage of data cache misses in read chains. A read chain represents a string of reads to a page from a processor, which is terminated by a write from any processor to that page. A long read chain indicates a page that could benefit from replication. The X-axis shows read chain lengths and the Y-axis shows the percentage of the total data misses that are part of read chains of that length or more. Pmake is not included in this graph because it has very few user misses.

6.2.2 Results for the Raytrace Workload

The raytrace workload sees a 19% reduction in execution time with data movement enabled. Through the migration and replication of pages, the user data locality is 77%. This increases the percentage of all misses serviced from local memory from 13% for first-touch to 64%. The improved data locality reduces user stall time by 61%. This reduction is comparable to that of the engineering workload. However, the improvement in execution time is less than the engineering workload because the time spent in user stall was originally only 36% for the raytrace workload compared to 67% for engineering. Also, the additional kernel overhead for data movement is still quite small, but larger than for the engineering workload.

Workload	Hot Pages	% Migrate	% Replicate	% Rearrange	% Kernel	%No Action
Engr.	8613	34	43	14	4	5
Ray.	5370	35	43	1	15	6
Splash	8535	55	29	3	10	3
DB	2003	13	2	0	0	85
Pmake	1355	28	28	15	21	8

TABLE 10. Breakdown of actions taken on hot pages. For each workload the decision taken by the policy for hot pages are broken down into percentages by category. The categories are migrations, replications, rearranging mappings (no movement necessary), kernel pages (no action can be taken), and instances where no action was taken for a number of other reasons, primarily read-write sharing.

Raytrace in this workload is a parallel application, therefore both the data and the code segments are shared. The code segment is quite small for raytrace, and the bulk of the user stall time is spent on data misses. The processes in the raytrace application make unstructured read-only accesses to a large shared data structure, which represents the scene to be rendered. Figure 20 illustrates the read-only nature of the *data* misses by showing the percentage of 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 large fraction of misses in long read chains indicates good potential for benefit from replication. Raytrace has 60% of its data misses in read chains that are 512 misses or larger. However, if a shareable read-only page is not being actively shared by multiple processors, then the policy will decide to migrate the page instead of replicating it. By migrating only pages that are actively shared and not all potentially shareable pages, the policy reduces the memory space overhead. Table 10 indicates that this behavior happens in raytrace with 43% of the hot pages being replicated and 35% being migrated.

6.2.3 Results for the Splash Workload

The Splash workload shows a 11% improvement in performance with data movement enabled. Through migration and replication of pages, the *user* data locality is 73%. This increases the percentage of all misses serviced from local memory from 20% for first-touch to 59%. The improved data locality reduces user stall time by 49%. The improvement in total execution time is less than in raytrace because the additional kernel overhead for data movement is relatively larger, and the improvement in user stall time is somewhat smaller.

In the Splash workload, 55% of the hot pages are migrated and 29% replicated. A higher fraction of the pages are migrated than in the raytrace workload. The splash workload has two parallel applications, Raytrace and Ocean. As the applications enter and leave the system, the simulated space partitioning scheme (and process control) changes the distribution of processors to jobs to best utilize the CPUs. Therefore static placement of data is difficult. In Ocean, the accesses by the different processes are mostly to disjoint parts of a shared data structure. Therefore, it has little potential benefit from replication. However, when processes of an application are moved, page migration will move the data being accessed to keep it local. Raytrace has read-mostly structures that could be replicated. From Figure 20, this workload has 30% of the data misses in read chains longer than 512, primarily from raytrace. Therefore we see gains from replication also.

The benefit from migration and replication for multiprogrammed parallel applications with space partitioning, could be much larger than that seen with the splash workload. In the splash workload, the space partitioning scheme moves the processes more frequently than might happen in real-life situations. This is done to work within the constraints of the length of the Splash applications, and the amount of time that can be reasonably simulated. When the processes of an application are moved, its pages are migrated by the kernel to maintain data locality. Once the cost to move pages is incurred, the benefit from better data locality is decided by how long the processes stay in that configuration before they are moved. Therefore, less frequent moves would produce larger benefits from migration and replication.

6.2.4 Results for the Database Workload

The database workload, running decision support queries, shows only a 4% improvement in performance with migration and replication over the basic first-touch policy[1]. Data movement is able to improve data locality from 19% in the first-touch case to 36%, resulting in a reduction in memory stall time of 12%.

We expected the database workload to benefit from replication because the data accesses would be mostly reads. We did not expect much benefit from migration because we locked the servers to the processors. However, we see very little additional benefit for replication over the first-touch case. We classified the pages based on the type of access, and found that only about 10% of the data misses 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. By studying the backtrace of the procedure call stack and looking at symbols in the binary, we inferred that these pages contain data structures used for synchronization purposes. They are read and written at a fine grain by all the processors. This sharing pattern was outlined in Section 5.2.1, and cannot

1. The actual improvement in performance seen was much larger. However, depending on how the queries are serviced by the four server engines, some of them could spend time spinning while waiting for more queries. We attempted to compensate for the spin time for the first-touch case, and the execution times shown reflect this adjustment. Therefore, the execution times for this workload should be considered approximate. All the other statistics, such as percentages for data locality and the distribution of page movement decisions, are exact.

benefit from replication or migration. Our policy is robust and correctly identifies this type of sharing. For 85% of the pages, the result was to take no action as shown in Table 10.

6.2.5 Results for the Pmake Workload

The pmake workload actually has a 3% increase in total execution time. The user stall time for the first-touch case is only 9% of the total execution time. The total data locality is mostly unchanged because most of the time is spent in the kernel.

A large fraction of pages that become hot are kernel pages. There are also a number of pages that are migrated or replicated. However there is very little benefit from this data movement because the processes are very short-lived. The pmake workload demonstrates that the kernel overheads to support migration and replication are low enough that they do not adversely affect a workload that cannot benefit from data movement. This is an important requirement for a performance optimization scheme.

6.3 System-wide Benefit

In addition to reducing stall time for individual applications, the improvement of data locality has an additional system-wide benefit. A cache miss to remote memory will involve two or more memory controllers, compared to a purely local miss that will involve only one memory controller. Additionally in the FLASH system, the software handlers for processing remote misses have a higher occupancy than for processing local misses. Therefore, an excess of remote misses can cause congestion in the interconnection network, increase occupancy in the directory processors, and increase queuing delays. Contention for resources can increase the latency of remote cache misses. By increasing the occupancy of the memory controller and causing queuing delays, remote misses can also significantly increase the latency of local misses too. Therefore with high contention, the actual latency of cache misses is much higher than the expected minimum.

We ran experiments with a very detailed memory model of the FLASH memory system that models the contention for the directory controller more accurately and gives queue statistics within the MAGIC memory controller. Taking the engineering workload for example, by enabling migration and replication and improving data locality, we also reduced the invocations of the remote mem-

Workload		Intr. Proc	Policy Decision	Page Alloc	Links & Map	TLB Flush	Page Copy	Rearr. Maps	Total
Engr	Repl	14	28	29	24	26	51	34	208
	Migr		15		28			23	188
Ray	Repl	18	24	24	33	28	47	26	199
	Migr		22		50			20	208
Splash	Repl	17	26	23	32	27	41	25	201
	Migr		17		35			18	188

TABLE 11. Latency of various functions in the implementation. The columns correspond to the implementation steps in Section 5.4.1. The numbers are shown separately for Replication and Migration, where applicable. The latencies are in µs and are averaged across the entire run.

ory request handler by 40%, the average network queue length for remote requests by 38%, and the maximum directory processor occupancy by 32%. Consequently, the average latency of a local read miss was reduced by 34%.

6.4 Analysis of Kernel Overheads

We have shown the performance improvements that data movement is able to provide. An important factor determining the extent of the improvement is the kernel cost of doing data movement. Therefore, we now analyze the overheads of doing data movement in more detail. For this purpose, we concentrate on the three workloads that showed significant benefits and moved a large number of pages — engineering, raytrace, and splash.

We present two views of the costs involved in migrating and replicating pages, the latency for a single operation and the total kernel overhead. First, we look at the latency of a page migration or replication operation in our implementation. The latency is the end-to-end time as seen by the interrupt handler when performing the required data movement operation. Table 11 shows the latencies for the different steps that make up a typical page migration or replication. These stages are based on steps 2 – 8 of the migration and replication policy, as outlined in Figure 17. Each value is the average for all the pages that go through a particular step. We provide separate values for migration and replication, where each does different amount of work in that step.

Next we look at a more global view of the costs for data movement, i.e., the increase in total kernel execution time because of migration and replication. The total kernel overhead and the per-

Workload	Kernel Ovhd (secs)	Percentage of Kernel Overhead							
		Intr. Proc.	Policy Decision	Page Alloc	Links & Map.	Page Copy	TLB Flush	RearrMaps	Page Fault
Engineering	1.69	6.5	11.7	13.3	6.2	23.9	17.2	19.2	1.9
Raytrace	1.17	9.7	9.2	10.0	11.3	19.6	22.0	11.4	6.8
Splash	1.91	7.2	8.6	9.5	10.6	16.8	22.1	10.7	14.5

TABLE 12. Breakdown of total kernel overhead by function. We show the percentage of the kernel overhead for migration and replication that can be attributed to the various functions. In addition to the steps from Figure 17, we added a category for additional page faults, due to changes in mappings. TLB flushing, and page copy comprise a large part of the overhead.

centage distribution of this overhead among the different categories is shown in Table 12. This overhead includes three cost components that are not directly reflected in the latency of a migration or replication.

- In addition to the latency for a TLB flush at the requesting processor, we now add the time spent at each of the other processors to field the IPI and flush their TLBs in response to the request.
- Additional page faults and TLB misses caused by the data movement implementation. The reason for these page faults is the temporary downgrade of page table entries to invalid during a migration, or to read-only for a replicated page. The flushing of TLBs causes the additional TLB misses.
- The processing time for pages that do not finally result in a migration or replication. For some pages the decision is “no action”, “kernel page”, or “rearrange mappings”. This additional time is added to the “policy decision” and “rearrange mappings” categories.

The results show that the latency per operation, migration or replication, is approximately 200μs. The largest components of both the latency and the total overhead are “TLB flushing” and “page copy”. These are followed by “rearrange mappings”, “page allocation”, “links & mapping”, and these functions spend a considerable fraction of their time in memory stall and synchronization in the VM system. Further work on the VM system to make it NUMA aware in its data structures and synchronization can reduce these costs. Some of this work has already been done in newer versions of IRIX.

We now consider the overheads in each category in more detail.

Interrupt processing: Interrupt processing is done once on the behalf of multiple pages, and their costs are amortized across multiple pages. The effective value for a single page is shown. Without the multiple page amortization, the interrupt processing cost would be about 30 – 36μs. A hardware interrupt happens once for every two pages. The page numbers are recorded in a structure, and the software interrupt bit is set to trigger the migration and replication code. The overhead for taking the hardware interrupt, recording the data, and taking the software interrupt is included in this category.

Policy decision: In this step the counters for the page and its replicas are read, and a policy decision is made. Each page requires two uncached reads, one for the cache-miss counters and one for the write bit. The latency for replication is higher because the counters for all the replicas have to be read and processed, if there are replicas for the page.

Page allocation: In this step, a page from the appropriate memory is found on the page free list and returned. A set of free lists by cache color is maintained per memory, and all the lists are protected by a single lock (`memory_lock`). About 95% of the time in this phase is spent in memory stall (75%) and synchronization for `memory_lock` (20%).

Links and Mapping: In this step the new page is linked into the appropriate VM structures and the page table entries are changed to be invalid (migration) or read-only (first replication). The latency is larger for migration than replication. During migration, `memory_lock` is acquired to remove the old page from the physical page hash table, and to insert the new page. In contrast, for replication only a page-level lock for the master page is acquired because the replicas are queued only on the master page. In this step too, about 95% of the time is spent on memory stall and synchronization, divided roughly equally between the two.

TLB flush: When we started on the implementation, we knew that TLB flushing was going to be a significant cost. Even in an earlier implementation, TLB flushing was clearly the largest overhead, taking over 50% of the time. We have worked hard to reduce the latency and overhead of the TLB flush as described in Section 5.4.3. Currently on average, a TLB flush is amortized across one and a half to two pages. If not compensating for multiple pages, the TLB flush latency would be 48µs, about the same as the page copy phase. When doing a TLB flush, each of the processors is sent an IPI. The time to field the IPI and perform the TLB flush is about 10µs. In Section 6.6, we will discuss possibilities for further reducing the TLB flushing overhead by reducing the number of IPIs sent.

Page copy: The largest fraction of the total latency is currently spent in the page copy phase. The page copy is done using a word by word copy. As would be expected, 85 – 90% of the time is spent in memory stall. In earlier versions of the implementation, the page copy was barely 10% of the total overhead. Through various optimizations, we have been able to reduce the TLB flushing cost and the synchronization component of other functions, such as “page alloc”. Therefore, in our current implementation the page copy cost is now significant. This overhead can probably be reduced through intelligent prefetching or through a hardware bcopy engine. A bcopy done by the processor in our current implementation takes approximately 50µs. A pipelined memory to memory copy done by the directory controller in FLASH takes about 35µs [HBG+97].

Rearrange mappings: In this step, all the page table entries pointing to the page are reset to their correct values. For a migrated page, the page table entries are reset to be valid. Replication takes longer than migration because we have to determine the closest replica for each process, and set the page table entry to point to it. In this step too, over 90% of the overhead is attributable to memory-stall time (over 60%) and synchronization time (30%).

6.5 Discussion

There are a number of factors that control the effectiveness of the data movement policy, and in this section we explore the effect of these factors on performance using a trace-based analysis. The primary advantage of and the reason for using a trace-based analysis is that it allows a quick (and approximate) exploration of a large policy space. A trace of cache misses and TLB misses is generated by running the workloads with the first-touch policy on SimOS. These traces are then used as input to a simulator of the data movement policy. The simulator tracks the location of pages, and computes effective execution times using the original information from the run that generated the trace, along with the new value for total data locality, and fixed values for the local and remote latency and the overhead for moving a page. The simulator can be customized to investigate the effect of various policies and parameters.

The policy issues considered in this section are studying the use of TLB misses as an approximation for cache misses, comparison of our migration and replication policy with other static and dynamic policies, response to memory pressure, and the sensitivity of the policy to policy parameters. Although we present these issues at the end of this chapter, much of this study was done earlier to guide the design and implementation of the policy.

6.5.1 TLB Misses and Other Metrics

The first issue that we consider is alternative indicators of memory system activity that may be used instead of cache misses. In the experimental runs in Section 6.2, cache-miss information collected by the directory controller was used to drive the policy decisions. Collection of cache-miss information requires some form of hardware implementation, and this hardware may not be available on all systems. We consider the use of TLB misses to avoid the extra counting hardware, and the use of sampling to reduce the cost of counting cache misses.

TLB misses are visible to the OS on systems that have software reloaded TLBs. The software TLB handler on such systems can be instrumented to collect information about the pages that are suffering TLB misses. Both TLB misses and cache misses indicate the regions of memory that are being accessed. The miss behavior of the TLB can be modelled as that of a cache with a large line size (a page) and only a few entries. The miss patterns are going to depend on the size and architecture of both the cache and the TLB. For example in the R4000 architecture that we model, the secondary cache has 4K entries each 128 bytes, mapping a total of 512Kbytes of memory at any time. On the other hand, the TLB had 48 entries each 8Kbytes (two pages) mapping about 400Kbytes of memory at a time. The TLB and the secondary cache map about the same size of memory in this example, but that will not be true in general. Also because of the large disparity in the number of entries (two orders of magnitude), the validity of approximating cache miss patterns using TLB miss patterns would depend on the memory access characteristics of the application.

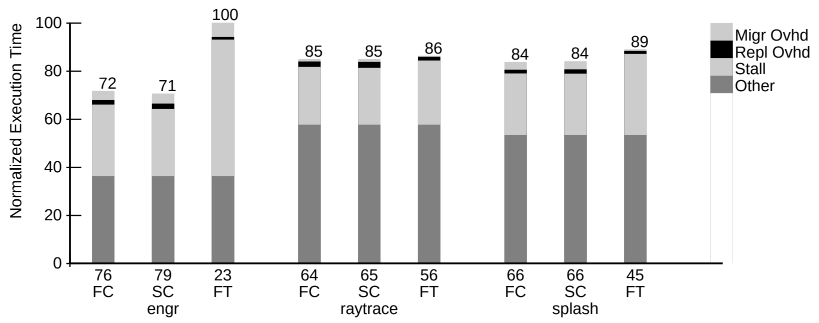

FIGURE 21. Performance impact of approximate information. There are three bars for each workload, Full cache (FC), Sampled cache (SC), and Full TLB (FT). Cache misses are sampled with a ratio of 1:10. Each bar shows the run time normalized to the run time for the first-touch case. The run time is broken down as User stall (local and remote), Overhead for replication and migration, and all other time. The percentage of misses to local memory is shown at the bottom of each bar.

We also study the effect of sampling of cache misses on the performance of the data movement policy. Full cache-miss information can be expensive to collect, especially when not implemented completely in hardware. A memory controller architecture like MAGIC, provides the flexibility to instrument software handlers to also count cache misses. However, the overhead of this extra counting code can adversely affect the latency of cache misses. The overhead of counting cache misses can be reduced by sampling instead of counting all misses. The issue to be studied is whether the sampling of cache misses reduces the quality of the information enough to reduce the effectiveness of the data movement policy.

To evaluate the effect of the different schemes, we compare three cases: full cache-miss information (FC), sampled cache-miss information (SC), and full TLB-miss information (FT). The cache-miss information is sampled at a 1 in 10 sampling rate. This sampling rate is sufficient to make the impact of counting cache misses in the MAGIC software handlers negligible. Figure 21 shows the results for the policies with approximate information. The important question when using partial or approximate information is how faithful is the heuristic to the original for the purpose at hand, in this case for driving the policy. The performance of the policy when using sampled cache-miss information is identical to that using full cache-miss information for all the workloads. Clearly sampled cache-miss information is an effective approximation for full cache-miss information. Using the sampling of cache-misses in our instrumentation of the software handlers in MAGIC, we are able to eliminate any overhead because of information gathering.

Using TLB misses to drive the policy is not always effective. The performance with TLB misses is quite good for the raytrace workload, and reasonably close to that with cache misses for the

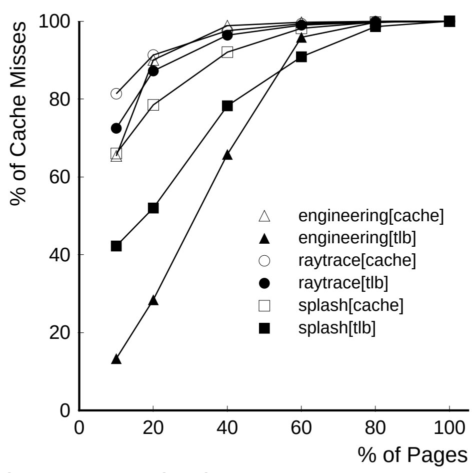

FIGURE 22. Effectiveness of TLB misses for finding hot pages. The X-axis is percentage of pages considered. The pages are sorted in descending order based on the number of misses taken, cache misses for one set of graphs and TLB misses for the other. The Y-axis shows the percentage of the total cache misses that are to the pages considered. A close correspondence between the cache and TLB lines for a workload indicates that hot pages for cache miss are also hot pages for TLB misses. This does not seem to be the case for the engineering workload.

splash workload. However, the engineering workload, which is the one that gets the maximum benefit from data locality, performs quite poorly when using TLB misses to drive the data movement policy.

The problem lies in the relative frequency of cache misses and TLB misses to a page. In [CDV+94], we successfully used TLB misses for the migration of pages of sequential applications because we only needed to know if the page was being accessed from a processor. However, in our current policy we need to know the pages that are accessed *frequently*; the ones that cross trigger threshold. Figure 22 demonstrates that this poor performance is because frequent TLB misses do not always happen to the same pages that have frequent cache misses. In the raytrace workload, there is a close correspondence between pages that take the most TLB misses and the ones that take the most cache misses. In the engineering workload on the other hand, the two sets of pages seem to be quite disjoint. The splash workload falls between the other two.

6.5.2 Alternative Policies

Another interesting study is how the performance of the combined migration/replication policy compares against other possible strategies, both dynamic and static. We consider six strategies, three static and three dynamic. The allocation strategies are:

- Roundrobin (**RR**), a static page placement scheme that is equivalent to random allocation.

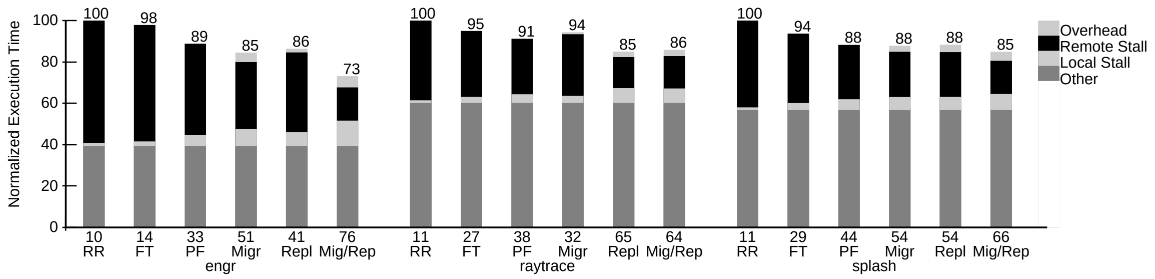

FIGURE 23. Comparison of different static and dynamic policies. There are six cases for each workload, Roundrobin (RR), First touch (FT), and Post-facto (PF), Migration-only (Migr), Replication-only (Repl), and the combined migration/replication policy (Mig/Rep). Each bar represents the execution time for a policy normalized w.r.t. the RR policy. Each bar shows cache-miss stall to local memory, cache-miss stall to remote memory, the overhead to migrate and replicate pages, and all other time. The percentage of misses to local memory is shown at the bottom of each bar.

- First touch (**FT**), a static page placement scheme where the page is allocated to the local memory of the processor that first touches the page.
- Post-facto (**PF**), an off-line optimal static page placement scheme where the placement of the page is decided after examining the entire trace. This scheme is not implementable in practice because it requires future knowledge.
- Migration only (**Migr**), a dynamic scheme similar to our implemented policy, but only migration is considered.
- Replication only (**Repl**), a dynamic scheme similar to our implemented policy, but only replication is considered.
- The full replication/migration policy (**Mig/Rep**) implemented for our runs.

Figure 23 shows the result for these six strategies. The parameters for the dynamic policies are the same as in the experimental runs in Section 6.2. The first point to note is that static placement is clearly not sufficient to provide good data locality in these dynamic workloads. The dynamic policies out-perform first-touch (FT), the implementable static placement policy, in all three workloads. Even the post-facto (PF) policy that assumes perfect future knowledge, is not able to match the performance of a combined migration/replication (Mig/Rep) policy.

The second point to note from these results is that a dynamic policy that does both migration and replication is needed to capture the full benefit from data locality for the workloads; just migration or just replication is not enough in all cases. In the engineering workload for example, the migration-only (Migr) and the replication-only (Repl) policies are only able to achieve a part of the pos-

Workload	Avail. Mem.	Pages Moved		Pages Utilized		Pages Used (%)	Data Locality
		Repl.	Migr.	Avg.	Max.		
Engr.	Unlimited	2394	5071	6731	9339	133	76
	Reduced	5905	5214	5858	7778	111	72
Raytrace	Unlimited	1727	780	6031	8798	119	64
	Reduced	2557	987	5597	7685	104	61
Splash	Unlimited	2013	4131	11341	16355	111	66
	Reduced	2569	4291	10929	14790	100	64

TABLE 13. 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 first-touch case and the percentage of misses that were to local memory. We see a gradual reduction in data locality, as total available pages is reduced.

sible data-locality benefit; 51% data locality for Migr and 41% data locality for Repl with a performance improvement of 15% and 14% respectively. Mig/Rep by being able to choose the appropriate action for each page is able to perform significantly better that either of Migr or Repl; 76% data locality with a performance improvement of 27%.

In the raytrace workload, the replication-only policy (Repl) is able to provide performance comparable to the Mig/Rep policy. Replication is sufficient to get the full performance benefit because the primary data structure in raytrace is read-only. However, the Repl policy is forced to replicate pages in all cases, whereas the Mig/Rep policy can choose to migrate pages where there is no concurrent access to the page. Therefore in this workload, the Repl and Mig/Rep policies provide similar performance improvements, but the Mig/Rep policy has a smaller memory space cost than the Repl policy.

6.5.3 Memory Pressure and the Memory Space Overhead

There is a memory space overhead associated with page replication because of the additional copies of pages. Our policy keeps this space overhead down in two ways. First, it selects only hot pages as candidates for replication. Second, it only chooses to replicate them if they are being actively accessed by multiple processors, otherwise it will migrate the page. An alternative to our dynamic replication policy is to replicate code pages on first-touch. In the engineering workload, replication on first-touch could potentially result in a 500% increase in memory usage for code pages because there are six instances of each application. We use our trace-based analysis to study memory usage because of replication (See Table 13). Our policy, by selecting only hot pages, increased memory usage by only 33% for engineering. Similarly in raytrace, where replication applies to code and data, the increase in memory usage was only 19%.

The policy responds to memory pressure, by stopping replication and reclaiming replica pages that have not been recently referenced. We create memory pressure by reducing the maximum number of pages that are available. Table 13 compares memory usage and data locality, with and without memory pressure. When memory is restricted, data locality reduces. The percentage of additional pages used in the limited-memory case is quite small, 0.3% for Splash to 11% for Engineering. There 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).

6.5.4 Sensitivity to parameters

The policy for migration and replication had two main parameters. The trigger threshold that determines when a page is considered hot. The hold threshold that determines if a page is being actively shared. Using the trace-based simulator, we explore the sensitivity of policy performance to these two parameters. Variations in the reset interval parameter had no impact on the performance of the policy.

Trigger threshold: The trigger threshold controls the aggressiveness of the policy, resulting in a tension between the achievable data locality, and the cost of data movement. Setting the trigger

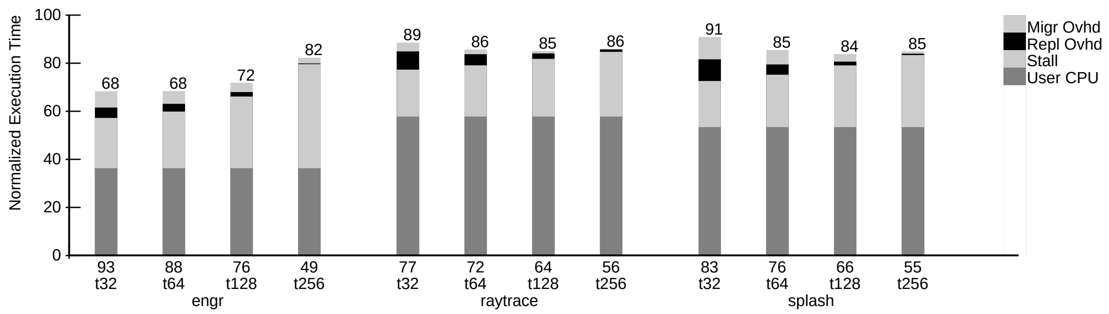

FIGURE 24. Effect of the trigger threshold. Each workload is run with four trigger thresholds, 32, 64, 128, and 256. The sharing threshold is a quarter of the trigger threshold. Each bar shows the run time for a configuration normalized to the run time for the roundrobin case for that application. In each bar, we separately show User CPU, User stall, and overhead cost of replication and migration. The percentage of misses made local is shown at the bottom of each bar.

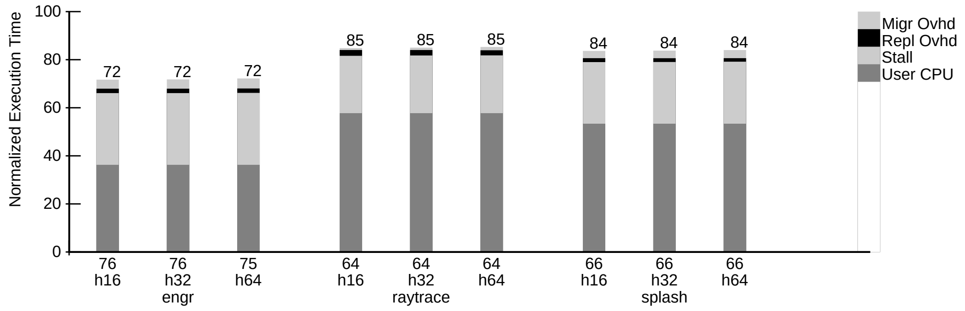

FIGURE 25. Effect of the sharing threshold. Each application is run with three different sharing thresholds, 16, 32, and 64. Trigger threshold is 128 for all cases and sharing threshold 32 is the base case. Each bar shows the run time for a configuration normalized to the run time for the Roundrobin 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 sharing threshold have minimal impact on performance.

threshold low will result in a more aggressive policy that is quicker to move pages, resulting in better data locality and reduced stall time. The down side is a potentially higher kernel overhead because of more pages moved. Another possible problem is that the lower trigger threshold may indicate that a page is hot when it really is not, and such pages will not provide a longer-term benefit from being moved to local memory. Setting the trigger threshold high would result in a conservative policy. Fewer pages will be moved, with consequently lower costs and potentially lower data locality. The down-side here is that the page is moved so late that a lot of the potential benefit from data-locality is lost.

The trigger threshold needs to be set to balance the benefits and costs to achieve the best performance. The appropriate trigger threshold for a machine will depend on the values of the local and remote miss latencies, and the cost of migrating or replicating a page. Figure 24 illustrates this trade-off clearly for the different workloads. As the trigger is lowered from 256 to 32, the stall time decreases (the data locality increases), and the kernel overhead increases. However, the results show that there is the possibility of a maximum performance point (minimum total execution time). For our machine configuration, the best performance seems to be stable around a trigger of 128. We also confirmed this trigger threshold as the best in our experimental runs.

Sharing threshold: The sharing threshold is used by the policy to differentiate between shared and unshared pages, and so decides whether a hot page is potentially migrated or replicated. A higher sharing threshold favors migration, and a lower sharing threshold favors replication. Figure 25 shows the results from varying the sharing threshold from 16 - 64 for a trigger threshold of 128.

The performance of the policy is quite insensitive to the value of the sharing threshold within a reasonable range — an eighth to half the trigger threshold in this case. This behavior indicates that most pages are clearly differentiated, being either shared (code and shared data of parallel applications) or unshared (data of sequential applications). Very few pages in the workloads have an ambiguous sharing behavior, which could significantly change the performance of the policy because of a change in the sharing threshold.

6.6 Scalability

Two features of the implementation present the most concern about the scalability of this solution, cache-miss counting and TLB flushing. The number of counters per page needed to count cache misses currently scales as the number of memories. This might present a problem if the solution were to be scaled to a machine with a large number of processors without any clustering. Our current solution would scale easily to about 32 processors unclustered or 128 processors with a typical clustering of four. The SGI Origin system implements 64 hardware counters per page of memory, and this covers 128 processors with the Origin's clustering factor of two. If it were desirable to scale this solution to larger sizes, the counter problem could be solved in different ways with some loss of precision. One possible solution is to make the counters coarse, i.e. each counter would represent more than one processor when required. Another solution is to have the operating system provide some support for localizing counters. The OS could group processors together, like cells in HIVE [CRD+95]. Each page would have a counter for each of the processors in the local group, and coarse counters for the processors outside the group. Most of the misses to a page would come from processors within the group except when running large parallel applications with many globally accessed pages. The OS will also ensure that processes are rescheduled mostly within a local group, and only when absolutely required, across local groups. This is very similar to memory affinity scheduling.

The other potential problem area is TLB flushing. Currently the TLB flush cost scales with the number of processors — all processors flush their TLB for every request. This is necessary because the OS has no information on what is currently in the TLB of each processor. Experimentally, we found that much fewer TLBs actually needed to be flushed for any request. In the raytrace workload for instance, on average only two TLBs (out of eight) contained the old mappings for any request, and so really needed to be flushed.

The solution to this scalability problem is to flush fewer TLBs for each request, only the ones that have a "good chance" of containing the obsolete mapping. The current implementation already tracks the processes that have a page table entry (PTE) for a physical page, through the added backmap functionality. The operating system also tracks the processor, if any, that is running a particular process. For a particular obsolete mapping, the only processors that need to flush their TLBs immediately, are the ones currently running processes that have a PTE pointing to the page. If a process with a PTE for the page is not currently running, it is marked with the current global generation count for the TLB flush request. The generation count mechanism was described in Section 5.4.3. When a processor schedules a new process, it checks its local TLB generation count against that of the new process. The processor will need to flush its TLB only if its count is less than that of the new process. The cost of a TLB flush with this solution, only scales with the number of processes sharing the page and currently running. This should be fairly small regardless of the size of the machine, except for a few shared pages of large parallel applications.

6.7 Summary of Experimental Results

An important problem on CC-NUMA machines is to ensure that a large fraction of cache misses are serviced from local memory at a lower latency rather than remote memory, i.e., applications have good data locality. We show that this data-locality problem on CC-NUMA machines can be addressed by OS-based migration and replication of pages. The kernel implementation of our policy is able to improve application performance by providing significantly better data locality. In our workloads, as much as 83% of user cache misses are serviced from local memory, and the overall performance improvement from better data locality is as high as 46% for the engineering workload. Our policy for migration and replication is robust and does not degrade performance for workloads such as Pmake and Database where there is no potential for improvement.

There is a substantial system-wide benefit from better data locality. A local miss utilizes fewer memory system resources compared to a remote miss. Therefore, better data locality reduces the contention for memory system resources, resulting in lower cache-miss latencies for the whole system.

A careful analysis of the costs of migration and replication show that the average latency for a page migration or replication is about 200µs. Of the various steps in migrating or replicating a page, flushing the TLBs and the copying the bytes in a page are the largest components of the kernel overhead. Each accounts for about 20%.

Finally, we use trace-based analysis to explore a number of issues related to migration and replication of pages. Our results show that:

- TLB misses are not a consistent approximation for cache misses, and so cannot be used as the sole metric to drive a migration and replication policy.
- Sampling of cache misses is as effective in driving the policy as full cache-miss information.
- Dynamic page-movement policies provide better application performance than static page-placement schemes
- Both migration and replication are needed to get the full data-locality benefit, and neither by itself is sufficient.
- Among the various parameters for the policy for migration and replication, the trigger threshold is the crucial policy parameter. It should be set based on the cost to migrate or replicate a page and the latency of local and remote misses.

Chapter 7
Related Work

Our work has been motivated by the issue of how the operating system for shared-memory multiprocessors needs to respond to the dynamic, multi-user, and multiprogrammed workloads being run on these systems. This type of workload is a significant departure from the single large parallel application that has been studied extensively in more recent research on these systems. In our work, we address two issues that are relevant to the new types of workloads for multiprocessors, performance isolation for groups of processes and data locality on CC-NUMA systems. In Section 7.1 we will consider work related to the performance isolation issue and in Section 7.2 we will consider related work in improving data locality.

7.1 Performance Isolation

Performance isolation has never really been studied as a topic before. The idea of an SPU, as a kernel abstraction, that allows the grouping of processes and resources and enables different sharing policies has not been previously proposed. The combination of SPUs and performance isolation gives multiprocessors a very desirable property; groups of processes can be guaranteed an agreed level of all system resources thus isolating their performance from system load, and at the same time they are able to use additional unutilized resources in lightly-loaded conditions for better performance.

Previous work in resource allocation has taken a rather piecemeal approach, focussing only on allocating individual resources (mainly CPU time) or only to individual processes. What has been lacking in these piece-meal solutions is a comprehensive idea such as performance isolation; a solution that encompasses all resources that can impact application performance, and is able to deal with arbitrary groups of processes. We also specifically target shared-memory multiprocessors, while most other work has been for uniprocessors only. However, our work is applicable to uniprocessors too.

We now describe some of the other techniques that have been proposed for the allocation of individual resources. Waldspurger [Wal95] demonstrates stride and lottery scheduling for providing proportional-share resource management for a variety of computing resources, including CPU, memory, disk and network bandwidth. While their work analyzes the different techniques for each resource individually, it does not consider an unified solution that accommodates all the resources. Their solutions are only proposed for uniprocessors and they do not consider multiprocessors. They provide a real implementation only for the CPU time resource, and the analysis for other resources is done using simple simulations. An important contribution of our work is a real implementation of all the mechanisms and policies described, and actually running real-life workloads to show the overall effectiveness of our solution. Their work is the only one to attempt at fairness for disk bandwidth allocation. Using simulations they show that for certain limited workloads their “funding delay cost” model for scheduling disk requests can achieve fairness. Our implementation that balances head position and fairness is different and more generally applicable.

A number of studies have considered fairness when allocating a single resource. Most of these studies have concentrated on CPU scheduling [Hen84][KaL88][WaW94]. An extension to [KaL88], the SHAREII resource management tool [Sof96] also assigns *fixed quotas* for virtual memory and other non-performance related resources such as disk space. A few proposals consider fairness for memory allocation. [Cus93] describes the scheme used to allocate memory to processes in Windows NT. This scheme consists of assigning shares of pages to *individual processes*, changing these shares dynamically based on page-fault rates, and a local page replacement policy. It is conceptually similar to our implementation for memory isolation in IRIX. However, they operate at process level, and provide no support for grouping processes. [HaC] have a proposal for allocating memory and paging bandwidth to disk using a market approach. They assume that there are enough processors, so CPU time is not an issue they consider. Their unit of fairness is again *individual processes*.

Though we do not discuss performance isolation for network bandwidth, the implementation would be similar to that of disk bandwidth, without the complication of head position. Stride scheduling is used in [Wal95] to study fairness for network bandwidth by changing the order of service from FCFS. Also, Druschel and Banga [DrB96] implement a scheme called lazy receiver processing (LRP) to provide fairness for network bandwidth.

The fairness goals of our work are similar to that of resource allocation schemes on mainframe multi-user systems [Den71][Sch80][ChW81]. These systems also tried to find a balance between fairness to the user and overall system utilization. However, they achieved long-term fairness for users through the allocation of quotas or budgets, and through accounting for resources used. A job stopped running when the user's resource budget was exhausted. In the short-term, fairness was usually provided only at a process level through access to the processor, moderated by system utilization factors such as available free memory and paging. Our model provides a more immediate (or short-term) idea of fairness for groups of processes, and it considers all resources.

The Stealth Distributed Scheduler [KrC91] implements isolation goals similar to ours in a limited sense, in the context of distributed systems. When scheduling foreign processes on a user's workstation, they attempt to prevent the foreign processes from affecting the performance of the workstation owner's processes for all the resources; CPU, memory, paging bandwidth, etc. Their technique is a simple pre-emption of resources from foreign processes if they are needed by local processes. They therefore solve the limited problem of preserving the performance of a single class of higher priority processes, and not on isolation and sharing for all SPUs as we do.

7.2 Data Locality

Significant work related to page migration and replication has been done on architectures other than CC-NUMA. A large body of this work has focussed on implementing shared memory in software on loosely-coupled multiprocessors or clusters of workstations. Some notable examples include IVY [Li88], Munin and Treadmarks [BCZ90], Midway [BZS93], Jade [RSL92], and

SAM [ScL94]. These systems lack any hardware support for accessing memory on another node. When a processor tries to access a remote datum, the software has to bring it to local memory. At this point the decision is made to either replicate or migrate the page or object that contains the referenced datum to local memory. Our work fundamentally differs from these software-based shared-memory systems because the hardware on CC-NUMA machines transparently provides the shared memory abstraction. Therefore page replication and migration on CC-NUMA systems is purely a performance optimization problem and not needed for correctness.

Another body of work has focused on migration and replication on distributed shared-memory machines that are not cache-coherent, such as the BBN-Butterfly and the IBM ACE. Definitive work in this area includes [ScD89][BFS89][CoF89][BSF+91][LEK91][Hol89][LHE92]. In these systems, the hardware enables a processor to directly access data in the memory of a remote node, but these systems either had no caches or the caches were not kept coherent by the hardware. Migration and replication on these systems was done to improve data locality and was considered as a substitute for providing hardware-based cache coherence.

The primary difference between this earlier work and ours is the ability of CC-NUMA machines to cache remote data, which substantially changes the cost function and potential benefits of migrating and replicating pages. In the CC-NUMA environment, although the remote access latency is on the order of a microsecond, with cache-coherence, subsequent accesses could hit in the local processor cache and take only a few nanoseconds. There is an enormous range in the latency (two or three orders of magnitude) to access a remote datum, depending on whether it is fetched from remote memory or a local cache. If the workload exhibits good cache locality, CC-NUMA systems will benefit much less from the migration and replication of pages. Therefore coherent caches force us to be more selective when moving pages, because the page movement overheads we can tolerate are much lower. Unlike non-cache-coherent NUMA systems where migration and replication is triggered by page and coherency faults, we go to great lengths to find hot pages by developing an efficient hardware-based cache-miss counting mechanism.

The workloads for these earlier studies were only single parallel applications, and migration and replication was used to do initial placement and to respond to changes in access patterns of the application. They did not consider multiprogrammed workloads as we do, and so did not have to deal with the important issue of rescheduling of applications, both parallel and sequential.

Though the performance results are not comparable because of the lack of coherent caches, there are some relevant areas in the mechanisms and policies. The freezing and defrosting of shared read-write pages to prevent them from bouncing around excessively is the motivation for our migration counter and threshold. Our kernel implementation is conceptually similar to that done by Cox and Fowler [CoF89], though ours is done in IRIX and the latter is a modification of MACH. They had to deal with the TLB consistency problem as we did, though it was not as big a concern for them as their overheads were dominated by the time to copy bytes. Scheurich and Dubois [ScD89] consider hardware counters for deciding when and where to migrate a page, but their scheme of maintaining only directional counters and moving pages a hop at a time in the right direction would not provide acceptable performance in CC-NUMA systems.

More directly for the CC-NUMA architecture, Black et al. [BGW89] proposed a competitive strategy to migrate and replicate pages using special hardware support (a counter for each page-frame per processor). The workloads evaluated were somewhat limited, a few small scientific applications from the SPLASH suite, run one at a time on a simple simulator for local and remote memory. In contrast, we evaluate workloads of realistic complex applications with multiprogramming, run on a real operating system. The policy space explored is also quite different.

The work by Chandra et al [CDV+94] on the Stanford DASH multiprocessor is a precursor to the data-locality work presented in this dissertation. They explored the scheduling issues for both sequential and parallel workloads on CC-NUMA multiprocessors. They also studied the migration of pages for data-locality, doing an implementation for sequential application workloads and using a simple simulator for parallel workloads. While being closely related, this work is different from ours in three important respects: (i) focus of their work was process scheduling and migration, while we build on their work and study memory locality through both migration *and* replication; (ii) the workloads in our study are more varied; and (iii) our study is based on the SimOS simulation environment. The SimOS environment used here is realistic and flexible, and allows us to choose architectural parameters — processor speeds, cache sizes, and latencies — more relevant to today's systems than when DASH was designed six years ago.

7.3 Other Related Work

We will now briefly outline some of the work that is loosely related to the issues covered in this thesis. There has been work on process scheduling for bus-based shared-memory systems addressing the issue of multiprogrammed workloads, both sequential and parallel; affinity scheduling[VaZ91] , gang-scheduling [Ous82], process control [TuG91], and scheduler activations [ABL+91]. We built on this scheduling work for our data locality studies, by extending it to CC-NUMA architectures [CDV+94].

The HURRICANE work [UKG+95] presents the concept of hierarchical clustering as a way to structure shared-memory multiprocessor operating systems for scalability. They run multiple microkernels on the system and also provide a single system image to the user. They target their work at mixed workloads, sequential and parallel, like we do, but the goal of their work is to provide OS scalability on NUMA multiprocessors.

The HIVE work [CRD+95] addresses the fault containment problem on large CC-NUMA multiprocessors. For these large systems, they propose a cell-based operating system structure to limit an application’s vulnerability to faults, to the resources that it actually uses, rather than the total resources on the system. For example, if an application is not using any memory from a cell, then the crash of a processor in that cell will not affect the application. This work is also targeted at mixed workloads as might be seen on a compute-server, the same as we address, but they address the issue of fault containment.

Chapter 8
Concluding Remarks

In the last few years there has been a gradual shift in the computing paradigm. The distributed paradigm of independent workstations on a network is giving way to a client-server computing paradigm. The latter solution is attractive because of lower costs, both through efficient utilization of and easier support and administration of the computing resources. This change in the computing paradigm has created the need for powerful and efficient servers to service requests from lighter-weight clients. Shared-memory multiprocessors are natural candidates to fill the role of servers because they aggregate and tightly couple multiple processors, large amounts of memory and I/O resources in a single system.

Most recent research on shared-memory multiprocessors has focussed on architectural and software issues related to speeding up large stand-alone parallel applications. Considerably less time has been spent on studying how operating systems for shared-memory multiprocessors need to respond to more dynamic compute-server workloads. In this thesis we have explored two important resource management issues that directly affect such workloads, namely performance isolation and data locality.

Both performance isolation and data locality become important when shared memory multiprocessors are used as general purpose compute servers. Our results show that for both issues it is possible to find an acceptable balance between sometimes conflicting requirements. On one hand the desire of good response time for an application, a user, or a group of processes, and on the other the efficiency (throughput) and fairness (load balance) of the system as a whole. We now separately present the conclusions drawn from our exploration of each issue.

8.1 Performance Isolation

The tight coupling of processors, memory, and I/O in shared-memory multiprocessors enables SMP operating systems to efficiently share resources. There has however been a popular belief that unlike workstations, SMP kernels on multi-user shared-memory multiprocessors cannot isolate the performance of a process or a group of processes from the load placed on the system by others. This had the implication that moving from workstations to compute servers was like taking a step backward to the old mainframe days. This work proves otherwise by showing that with an appropriate resource allocation model, a shared-memory compute server can provide workstation-like isolation in a heavily loaded system and maintain the benefits of resource sharing of SMPs.

We introduce the idea of performance isolation for shared-memory multiprocessors to address the issues described above. Performance isolation replaces the per-process CPU-centric control over resource sharing found in current SMP operating systems that had the undesirable result of allowing unrelated processes to interfere with each others' performance. Performance isolation gives users or tasks significantly better control over the performance they can expect when utilizing a shared machine. Groups of processes are isolated from the background load on the system, and are guaranteed a fixed share of the machine's resources based on pre-configured contracts or agreements. Performance isolation also maintains good throughput by carefully reallocating under-utilized resources to processes that might need them.

To implement performance isolation, we introduce a new kernel abstraction called the Software Performance Unit or SPU. The SPU associates computing resources on the system with groups of processes that are entitled to these resources, and restricts the use of these resources to the owning processes. The SPU is the unit of isolation, and provides a powerful mechanism to enable different contracts between users or services for sharing a larger machine. Each SPU has associated with it a sharing policy that decides how and when resources belonging to the SPU may be shared with other SPUs. The performance isolation model provides good throughput by carefully reallocating resources idle in an SPU to other SPUs through the sharing policy.

We show that the implementation complexity of performance isolation is manageable by describing our implementation in the IRIX5.3 kernel from Silicon Graphics. Our implementation manages the CPU, memory and disk bandwidth resources of the system. Running a diverse set of workloads on this kernel implementation, we demonstrate that performance isolation is feasible and robust across a range of workloads and resources. The results show that performance isolation is successful at providing workstation-like latencies by isolating performance under heavy load (0 - 13% worse than an isolated machine for our workloads), SMP-like latencies under light load, and SMP-like throughput in all cases.

Given the results that we demonstrate, this resource allocation model should be implemented in the operating systems of shared-memory multiprocessor compute servers to provide users of such machines with the performance that they expect.

8.2 Data locality

The CC-NUMA architecture is being used for larger shared-memory multiprocessors, e.g. SGI Origin [LaL97], Sequent STiNG [LoC96], Convex Exemplar [BrA97], and Data General NumaLiine. We believe that in the future even small to moderate sized machines (8 - 32 processors) will use the CC-NUMA architecture because of the increasing speed of next generation processors. In fact, the SGI Origin line uses a NUMA architecture for machines with more than two processors. The key factor that affects the performance of applications on these systems is memory locality. Static allocation of processes to processors, and the placement of their data in local memory is not an acceptable solution for maintaining data locality for compute-server workloads. Eventually, the operating system will need to move processes to balance load and provide fairness across the system. In this environment, we have explored the problem of the operating system providing data locality through the automatic migration and replication of pages.

Based on our analysis of the problem, we implemented and optimized the functionality for page migration and replication in the kernel. We assembled a set of realistic workloads that included single and multiprogrammed parallel applications, engineering simulators, software development tools, and a database server. The results from running these workloads on the kernel implementation of our policy showed that migration and replication of pages done by the operating system is very successful at improving memory locality. On an eight-processor configuration that we used, we were able to improve the locality for user memory accesses to as much as 80% of misses satisfied from local memory. This is to be contrasted with a locality of about 12% using a static first-touch policy. This improvement in data locality reduced the user stall time significantly, and improved application performance by as much as 46%. For applications that could not benefit from migration or replication because of their data access patterns, our policy and implementation were robust and did not hurt performance.

An additional benefit from the improved data locality is a significant decrease in the contention for resources in the NUMA memory system — shorter network queues, lower occupancy, and lower latency for memory accesses. This is an important effect as it reduces the impact that a memory intensive application can have on others.

A careful analysis and tuning of the kernel implementation to reduce the costs and overheads of doing migration and replication was required to obtain the robustness and improvement in performance. The first source of overhead was the synchronization in the virtual memory system. The existing coarse locking was augmented with finer-grain locks to reduce the time spent in synchronization. The next source of overhead is the mechanism to maintain TLB consistency. The overhead to flush TLBs was reduced by implementing a lighter-weight interface to request and acknowledge flushes, and more efficient handling of the inter-processor interrupt. The cost of copying the pages turns out not to be as large a factor as it was in earlier studies on non-cache-coherent shared-memory systems.

Hardware-provided caching can hide the latency when repeatedly accessing data in remote memory, therefore we have to be more conservative about which pages we move. We added functionality for cache-miss counting to drive the decision of migrating or replicating only the busiest pages. We implemented an efficient counting mechanism in software for MAGIC, the programmable memory controller of the FLASH machine. The cost of cache-miss counting was made negligible by using sampling of cache misses. We show that sampling at a rate of 1 in 10 does not degrade the quality of the information. We also investigated the use of other forms of information to drive the policy that did not require hardware changes, and found that TLB misses were an inconsistent approximation for cache misses. TLB misses were reliable for some applications, but not for others. To support page migration and replication, future machines should include the ability to collect sampled cache-miss information.

Our work demonstrates that operating system based data locality is possible and necessary on CC-NUMA machines to maximize both individual application and overall system performance. We show that both migration and replication are necessary for the complete data-locality solution, and that dynamic policies are able to do better than even the optimal off-line static placement scheme. For workloads where memory stall time is a problem, page migration and replication is an effective solution. For applications with access patterns that cannot benefit from replication or migration of pages, our implementation does not degrade performance. The low overheads of our implementation along with the algorithm that we developed, results in a robust migration and replication policy. We believe that a fully optimized migration/replication implementation will allow for a more aggressive policy leading to larger gains.

8.3 Future Work

We have addressed the issues of performance isolation and data locality at a fairly comprehensive level, providing implementations of the ideas in a real kernel and exercising these implementations by running realistic workloads. We now outline some areas for future investigation related to our work on performance isolation and data locality.

8.3.1 Performance Isolation

The SPU abstraction that underlies the performance isolation model is quite general. We picked a specific sharing policy to demonstrate performance isolation. The sharing policy space is rich, and a wide variety of sharing policies and resource allocation schemes could be investigated within the SPU framework to achieve other goals. Closely aligned with sharing policy is the specification of more complex SPU relationships and organizations. An obvious SPU organization to investigate is hierarchical SPUs, where the resources of a parent SPU is divided among two or more other SPUs.

One area that we did not investigate is performance isolation for interactive applications. The requirement for interactive response time is more stringent than throughput rates, and might present some special challenges. Similarly, the performance isolation idea could be extended to the world of “real-time” applications. Currently most real-time schemes only apply to CPU time, and do not consider other resources.

The extension of performance isolation to CC-NUMA architectures may present some interesting issues. On a bus-based shared-memory multiprocessor all pages are conceptually the same, and so we only need to maintain a count of pages used by a SPU. When moving to a CC-NUMA architecture, pages on each memory node are conceptually different. Closer coordination between the allocation of CPUs and the allocation of memory pages would be necessary to maintain good data locality.

The scalability of operating systems for shared-memory multiprocessors is a subject of ongoing research. Any work done in this area benefits performance isolation. There are interesting issues in the contention for kernel resources that affect both performance isolation and the scalability of the kernel.

8.3.2 Data Locality

The scalability of the data locality solution presents interesting avenues of future work. The main issues are TLB flushing and cache-miss counting. Some interesting possibilities for scalability solutions were outlined in Section 6.6, others remain to be discovered.

The tighter integration between process scheduling and the VM system is an interesting next step in the complete solution to the data-locality problem. This sort of integration was also required by the scalability solution for cache-miss counting. A combined approach to the allocation of memory and processors to tasks, integrating and extending the current techniques for process scheduling and data movement, is likely to yield additional performance benefits and a more robust data-locality solution.

Another interesting possibility is the comparison of the performance of OS-based data-locality solutions with that of hardware-based ones, such as the variants of the COMA architecture, investigating the differences, similarities, and possible synergies between these schemes.

References

[ABL+91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pages 95-109, October 1991.

[ACP+94] T. Anderson, D. Culler, D. Patterson. A Case for NOW (Networks of Workstations). Presented at *Principles of Distributed Computing*, August 1994.

[ABC+91] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D.d Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 2-13, June 1995.

[BCZ90] J. K. Bennett, J. B. Carter, W. Zwaeneopoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming*, pages 168-175, March 1990.

[BrA97] T. Brewer and G. Astfalk. The evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring `97*. pages 81-96, 1997

[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.

[BFS89] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Operating Systems Review*, pages 19 - 31, vol. 23, no. 5, 1989.

[BSF+91] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, 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.

[ChW81] D. Chess and G. Waldbaum. The VM/370 resource limiter. In *IBM Systems Journal*, vol.20, no.4, 1981, pages 424-37.

[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 12th ACM Symposium on Operating Systems Principles*, pages 32-43, December 1989.

[CRD+95] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *The 15th ACM Symposium on Operating Systems Principles,* December 1995

[Cus93] H.Custer. Inside Windows NT. Microsoft Press, 1993.

[Den71] P. Denning. Third generation computer systems. In *ACM Computing Surveys*, vol.3, no.4, Dec. 1971, pages 175-216,

[DrB96] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 261-275, October 1996.

[ENC+96] A. Erlichson, N. Nuckolls, G. Chesson, and J Hennessey. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pages 210-221, October 1996.

[Gar94] Gartner Group Inc. The Cost of LAN Computing: A Working Model. Feb. 7, 1994.

[HaC] K. Harty and D. Cheriton. A Market Approach to Operating System Memory Allocation. Stanford TR, *http://www-dsg.stanford.edu/Publications.html*.

[HBG+97] J. Heinlein, R. Bosch, Jr., K. Gharachorloo, M. Rosenblum, and A. Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proceedings of the 11th International Parallel Processing Symposium,* April, 1997.

[Hen84] G.J. Henry. The Fair Share Scheduler. *AT & T Bell Laboratories Technical Journal*, October, 1984.

[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.

[IBM] IBM Virtual Machine Facility/370: System Programmer's Guide. IBM Systems Library, Order No. GC20-1807-7, IBM Corporation.

[JoH94] Truman Joe and John L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture,* pages 82-93, Chicago, IL, April 1994

[KaL88] J.Kay and P.Lauder. A Fair Share Scheduler. *Communications of the AC*M, January, 1988.

[KrC91] P. Krueger and R. Chawla. The Stealth Distributed Scheduler. In *11th International Conference on Distributed Computing Systems*, May, 1991.

[KTR94] D. Kotz, S. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. *Dartmouth PCS-TR94-220*, July, 1994.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The

Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[LaL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241-251, June 1997.

[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, Oct. 1991.

[LHE92] R. P. LaRowe Jr., M. Holliday, and C. S. Ellis. An analysis of dynamic page placement on a NUMA multiprocessor. In *Performance Evaluation Review*, pages 23 - 34, June 1992.

[LeL82] H. Levy and P. Lippman. Virtual Memory Management in the VAX/VMS Operating System, *IEEE Computer*, March, 1982

[LLJ+92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In P*roceedings of the 19th International Symposium on Computer Architecture*, pages 92-103, May 1992.

[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.

[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.

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

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, Aug. 1995

[Ous82] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems, In *3rd International Conference on Distributed Computing Systems*, 1982

[PaH96] D. Patterson and J. Hennessy. Computer Architecture A Quantitative Approach (Second Edition). Morgan Kaufmann Publishers, Inc. San Francisco, CA, pages 635-755.

[RBD+97] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, January 1997.

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

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

[ScD89] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. In *IEEE Transactions on Computers,* pages 1154 - 1163, August 1989.

[Sch80] R. Schardt. An MVS tuning approach (OS performance problem solving). In *IBM Systems Journal*, vol.19, no.1, 1980, p. 102-119.

[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.

[Sof96] SHAREII: A resource management tool. SHAREII data sheet, Softway Pty. Ltd., Sydney, Australia.

[SRL90] L. Sha, R. Rajkumar, and J. Lehozcky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, pages 1175-1185, September 1990.

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

[Teo72] T. Teory. Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems. In *Proceedings of AFIPS Fall Joint Conference*, pages 1-11, 1972.

[TuG91] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In P*roceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 159-166, December 1991.

[UKG+95] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. In *Journal of Supercomputing*, vol.9, no.1-2, pages 105-34, 1995.

[VaZ91] R. Vaswani and J Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared-memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26-40, October 1991.

[VeR97] B. Verghese and M. Rosenblum. Remote Memory Access in Workstation Clusters. *Computer Systems Laboratory, Stanford University, Technical Report: CSL-TR-97-729,* July 1997.

[WaW94] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the first symposium on Operating Systems Design and Implementation,* November, 1994.

[Wal95] C.A. Waldspurger. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. *Ph.D. Thesis, Massachusetts Institute of Technology*, September 1995.

[WiR96] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *ACM SIGMETRICS '96*, May 1996.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, June, 1995.