Scalable Memory Management through Hierarchical Symmetric Multiprocessing by

Ronald C. Unrau

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
Computer Engineering Group
University of Toronto

Abstract

This dissertation examines scalability issues in the design of operating systems for large-scale, shared-memory multiprocessors. In particular, the thesis focuses on structuring issues as they relate to memory management.

From a set of simple, well-known queuing network formulas, we derive a set of properties that describe sufficient conditions for an operating system to scale. From these properties we first develop a set of guidelines for designing scalable systems, and then develop a new structuring philosophy for shared-memory multiprocessor operating systems, called Hierarchical Symmetric Multiprocessing (HSM).

HSM manages the system resources in clusters, using tight coupling within a cluster, and loose coupling across clusters. Distributed systems principles are applied by distributing and replicating system services and data objects to increase locality, increase concurrency, and to avoid centralized bottlenecks, thus making the system scalable. However, tight coupling is used within a cluster, so the system performs well for local interactions. HSM maximizes locality which is key to good performance in large systems, and systems based on HSM can easily be adapted to different hardware configurations and architectures by changing the size of the clusters. Finally, HSM leads to a modular system composed from easy-to-design and hence efficient building blocks.

Memory management is a particularly challenging service to implement within the HSM framework, because it must provide the applications with an integrated and coherent view of a single system, while distributing and replicating services in order to fully exploit the hardware potential. We describe in detail the implementation of an HSM structured memory management subsystem, and evaluate the performance of our implementation on Hector, a prototype scalable shared memory multiprocessor.

Acknowledgements

First, I would like to thank Dr. Michael Stumm and Dr. Zvonko Vranesic for their supervision and guidance throughout my doctoral work. They allowed me the freedom to be creative and taught me the discipline necessary to complete such a large undertaking.

I believe that the Hurricane/Hector project has been successful primarily because of the quality of the people involved. I would especially like to thank Orran Krieger, who has almost as much invested in this work as I do. Our combined but different strengths resulted in a synergy that made our research exciting and unique. This synergy was also apparent with the other members of the team: Ben Gamsa, Ron White, Jan Medved, Uma Shunmuganathan, Fernando Nasser, Yonatan Hanna, and David Blythe.

I can only begin to express my gratitude to my wife Gail. She continues to be a constant source of encouragement and understanding, and in many ways suffered more for this degree than I did. This dissertation is dedicated to her.

Finally, I would like to thank the Natural Sciences and Engineering Council, the Information Technology Research Center, and the University of Toronto for their financial support.

Contents

Chapter 1

Introduction

This dissertation examines scalability issues in the design of operating systems for large scale shared memory multiprocessors. In particular, the dissertation focuses on structuring issues as they relate to memory management. The motivation for our research stems from recent advances in shared-memory multiprocessor architectures, which will soon be able to support hundreds of processors. Such multiprocessors achieve their scalability through segmented architectures, which allow increased bandwidth as new segments are added. Scalable architectures challenge both application and operating system designers to develop software that can exploit their potential effectively. To meet this challenge, we have developed a new operating system structure called *Hierarchical Symmetric Multiprocessing* (HSM). The advantages of HSM are demonstrated by the memory manager of HURRICANE, a new operating system we have developed to study issues in scalability.

In this introductory chapter, we define the basic goals and constraints of the dissertation. The environment for which HURRICANE is targeted is defined in Section 1.1, followed by a description of basic memory management responsibilities in Section 1.2. The goals of our work are then defined in more detail in Section 1.3. Section 1.4 introduces the basic philosophy of hierarchical symmetric multiprocessing, and Section 1.5 defends the decision to evaluate the architecture through implementation. The chapter concludes with an overview of the remainder of the dissertation.

1.1 Target Environment

All memory management systems have two primary responsibilities: the support of application execution, and the management of memory resources. In meeting these responsibilities, the memory manager is constrained by the needs of the applications, by the operating system of which the memory manager is a part, and by the hardware on which the system runs. Together, the constraints define the target environment of the memory management subsystem. This section defines the attributes of each component that forms HURRICANE's target environment.

Application Workload: Hurricane is intended to be a general-purpose, multi-user operating system. The workload has typically been a mix of development tools, and engineering and scientific applications. The memory manager sees this workload as multiple independent programs running simultaneously, each with different memory demands and access patterns. For example, the application mix could include independent concurrent instances of the same program running on different processors, or a parallel application running on several processors. In both types of workload, the code and data are accessed in parallel on different processors, but the sharing patterns are very different. The memory manager must balance its data structures and management policies to respond effectively to these different workloads.

Operating System Environment: Hurricane is based on a message-passing microkernel, similar in concept to the V system [23]. The kernel is intentionally minimal, providing support for processes, interprocess communication (IPC), and exception handling. The memory manager runs as part of the kernel because it must handle page faults and manage processes that are waiting for I/O. The remainder of the operating system services, including high level I/O, are provided for by servers that typically run as normal user-level programs.

The operating system also defines the memory abstraction seen by application programs. Address spaces are protected containers of virtual memory in which any number of processes can execute. An address space is composed of multiple non-overlapping regions, where each region is a contiguous block of memory that is bound to a corresponding contiguous file region on secondary store. Once a binding is established, accesses to the memory within a region behave as if they were accesses to the file directly. This powerful abstraction is known as a single-level store or mapped file interface.

Target Architecture: The general structuring and concepts of Hurricane are applicable to a wide range of MIMD (Multiple Instruction Multiple Data) architectures [29]. However, our research efforts have so far focused on scalable shared memory multiprocessors. These machines typically comprise multiple workstation class microprocessors, each with its own hardware cache. The machines are called shared memory multiprocessors because global physical memory is directly accessible by all processors. Global memory is accessed through an interconnection network, although many machines provide memory local to each processor to help reduce network traffic. Hurricane does not require that memory or cache coherence be supported in hardware, but the machine should support paged virtual memory and some form of atomic lock operation.

1.2 Responsibilities of Memory Management

The memory manager is responsible for the support of application level memory abstractions, and for the effective management of the physical memory resources of the machine. Throughout the dissertation, we shall use the term virtual resource management when referring to structures or operations in support of the application interface; physical resource management refers to structures or operations dealing with the physical memory of the machine. The rest of this section looks at the responsibilities at both levels in more detail.

At the virtual level, the single-level store interface is actually a three-tiered abstraction. At the top level, an address space contains all the memory resources a program may access directly. The non-overlapping regions within an address space describe the attributes of a contiguous sequence of pages. The page is the lowest level of the hierarchy and is the smallest unit to which attributes and mappings can be applied. The memory manager is responsible for the maintenance and integrity of the data structures describing these application-level abstractions and for supporting operations on them. Maintenance includes allocation and deallocation of address spaces and regions, as well as checking and setting access permissions. Operations on virtual memory resources include moving data between address spaces, and sharing data within and across address spaces. The protection semantics for these operations are defined by the HURRICANE communication protocol, so the memory manager works with the kernel to ensure the semantics are not violated. Most activity involving virtual resources is initiated by application requests, so the memory manager is responsible for the call interface and for ensuring fast servicing of requests.

In a single level store system, all main memory can be considered a cache of secondary store. Thus, the memory manager's chief responsibility at the physical level is as a cache manager. The cache analogy implies page placement and replacement policies, and the mechanisms and data structures needed to support them. If cache or memory coherence is not supported by the hardware, the memory manager must also ensure the consistency of data items replicated within and across the levels of the storage hierarchy. Operations at the physical level are on a per page basis and are primarily demand driven, which means they are initiated by the hardware as the result of memory-related faults (ie. page faults or page access violations).

1.3 Goals

The primary goal of the dissertation is to study scalability and structuring for scalability as it relates to memory management. This work represents the first results of a larger research effort to study scalability of operating systems in general, and eventually to extend the findings to the application level. This section identifies several challenges facing the architect of a scalable system, by drawing on a formal definition of scalability developed by Nussbaum and Agarwal [48]: *the scalability of a given architecture is the fraction of the parallelism inherent in a given algorithm and problem size that can be exploited by a machine of that architecture.*

From the definition, the primary goal of the memory manager is to allow applications to exploit the hardware to the full extent of the parallelism inherent in both. This implies that a scalable memory manager must have as much inherent parallelism as either the algorithm or the architecture. In particular, the data and control structures of the system should not saturate before the algorithm or architecture. In many instances, the memory manager can help reduce contention of both the interconnection network and memory by supporting policies that allow the placement or replication of data so as to reduce average access times.

The definition above specifies scalability in terms of a *given* algorithm on a *given* architecture. Each algorithm has its own characteristic memory access patterns, just as each architecture has different resource bandwidths. The structuring and policies of the memory manager must be general enough to support different classes of applications on various architectures, yet flexible enough to allow performance tuning for both. Given the target workload and architectures described earlier, this goal is very challenging. In particular, we want to support large parallel scientific applications, but not at the expense of smaller scale parallel or sequential tasks. As for hardware, the operating system should at least run efficiently on different configurations of a given architecture. More ideally, the operating system should be adaptable to run efficiently on different architectures.

As a third point, we note that Nussbaum and Agarwal define scalability in relative terms independent of the number of processors. Yet one intuitively expects that the techniques used to manage a 16 processor system are likely to be different from those used to manage 256 processors. We expect the data structures appropriate for a large system to have a greater potential for concurrency, but they may also have a higher constant cost compared to the structures used for a small system. Because of the challenges this issue presents, one of our research goals is to study the cost-concurrency trade-offs of scalable data structures.

In defining the areas addressed by the dissertation, it is equally important to identify areas that are not addressed. For example, the dissertation does not address policy issues, but instead focuses on structuring and mechanism as applied to memory management. Where policies are described, it is with the purpose of showing how the underlying data structures and mechanisms interact — the policies themselves should be viewed as examples only. Also, while the dissertation deals with scalability and structuring for scalability, we do not claim to *prove* that our design is scalable. While we can make informed arguments, we do not know how to prove scalability and in fact, it may well be impossible to prove that an operating system is scalable.

1.4 Hierarchical Symmetric Multiprocessing

Existing multiprocessor operating systems have typically been scaled to accommodate a large number of processors in an ad hoc manner, by repeatedly identifying and then removing the most contended bottlenecks. This is done either by splitting existing locks, or by replacing existing data structures with more elaborate, but concurrent ones. The process can be long and tedious, and results in a system that 1) is fine-tuned for a specific architecture and hence is not easily portable to other hardware bases with respect to scalability; 2) is not scalable in a generic sense, but only until the next bottleneck is reached; and 3) has a large number of locks that need to be held for common operations, with correspondingly large overhead.

We believe that a more structured approach to designing scalable operating systems is necessary, and propose a new technique called *Hierarchical Symmetric Multiprocessing*. Our approach allows systems to scale by providing increased service bandwidth as the system grows, and by attempting to maximize locality through the distribution and replication of data structures and service sites. Increased service capacity is achieved by constructing a large system out of smaller building blocks called *clusters*, where each cluster provides the complete functionality of a small scale symmetric multiprocessor operating system. Multiple clusters cooperate and communicate to give users and applications an integrated and consistent view of a single large system. Hierarchical symmetric multiprocessing is hierarchical because of the two-tiered decomposition of the system into clusters of processors; it is symmetric because, within each level, no particular cluster or processor has specialized service capabilities.

HSM incorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. On the one hand, scalability is obtained by using the structuring principles of distributed systems, where services and data are replicated: a) to distribute the demand, b) to avoid centralized bottlenecks, c) to increase concurrency, and d) to increase locality. On the other hand, there is tight coupling within a cluster, so the system is expected to perform well for the common case, where interactions occur primarily between objects located in the same cluster.

Hierarchical symmetric multiprocessing is described in more detail in Chapter 3. Its application to the Hurricane memory management subsystem is the main focus of this dissertation, and is described in Chapters 5 and 6.

1.5 Methodology

The results in the dissertation are based on implementation and evaluation through experimentation. This section briefly motivates our choice in the context of the other two primary evaluation methodologies, namely analytic modeling and simulation.

Analytic models are powerful analysis tools that allow investigations into architectural trade-offs without concern for implementation details. With proper parameterization, these models can be used to explore a large part of the design space, including hardware variations that may better support memory management and applications. However, analytic models are only as accurate as the assumptions on which they are based, making validation an important part of the modeling process. As well, proper parameterization of parallel workloads is difficult, as is modeling of transient phenomena such as program start-up, or non-linear behavior such as hardware caches. Consequently, analytic modeling is primarily used to quantify upper and lower bounds, to estimate average high-level behavior, and to predict unexpected anomalies that are a consequence of the assumptions.

Simulation is a second valuable analysis tool that can capture transient and non-linear behavior while still allowing the study of a broad range of parameters. Unfortunately, the detail required to model this behavior comes at the cost of increased time, to the point where the benefits of simulation may be completely lost. Like analytic models, simulation also suffers from appropriate workload generation. Trace-driven studies can be difficult to generalize to new systems, since they can contain timing dependencies inherent in the systems from which they are derived, and artificial workloads are difficult to parameterize adequately.

Evaluation through implementation is often criticized for providing only a single data point in the design space, and even then the results may be clouded by implementation idiosyncrasies that hide higher level architectural issues. However, implementation is valuable in serving as a proof of concept. It allows validation of analytic models and simulation studies, and provides a test-bed for program development from which workload characteristics can be derived. Also, implementation forces a detailed exploration of the interaction between different components, often yielding insights that could not be gained otherwise. These considerations, coupled with the ready availability of the Hector multiprocessor [65], have motivated our decision to c hoose implementation as our research methodology.

1.6 Contributions

This section reviews the three main contributions of the dissertation: a set of criteria sufficient for an operating system to scale; a structuring framework, called hierarchical symmetric multiprocessing, that incorporates the criteria for scalability; and the application of HSM to the Hurricane memory manager.

Our research is the first effort that we know of to address the problem of scalability of operating systems for shared memory multiprocessors in a structured and integrated way. To address the problem of structuring for scalability, we first had to determine what scalability really means in the context of an operating system. Our study led to the first major contribution of the dissertation: from a set of simple, well-known queuing network formulas we derived a set of properties that describe sufficient conditions for an operating system to scale. From these properties we were able to develop a set of guidelines for designing scalable systems.

Hierarchical symmetric multiprocessing is a new structuring philosophy that evolved out of these scalability considerations. Scalability is achieved while still maintaining an integrated and consistent system by capitalizing on locality inherent in our target environment. The concepts of HSM can be applied not only to the components of the operating system, but to application programs as well.

The evaluation of the HSM architecture through its implementation in Hurricane has turned out to be an important contribution. This is true not only because the implementation serves as proof of concept, but because it allows both system and application developers to experiment with various trade-offs on a running system. We believe that operating system design is and should be one of iterative refinement, and so the lessons learned from our experiences will allow future generations of systems to scale even further.

1.7 Overview of the Dissertation

The general development of the dissertation is to present the architecture of hierarchical symmetric multiprocessing, describe its application to memory management in Hurricane, and evaluate the performance of the prototype. This section outlines the progression in more detail, on a chapter by chapter basis.

The memory manager has responsibilities to both the applications above it and the hardware below. The systems architect must understand the behavior and interactions at both levels because any decision must take all the factors into account. Chapter 2 provides some background in these areas, with the aim of defining common terms and concepts needed for later development. Towards this end, the chapter first gives an overview of scalable architectures and some of the issues involved with them, and then describes virtual memory hardware and techniques in some detail.

The architecture of a hierarchical symmetric multiprocessing system is developed in Chapter 3. The architectural definition follows as the result of several properties that we argue are sufficient for a system to scale. The properties are derived from an analysis of scalability and the factors that affect it. The remainder of the chapter shows how the principles of hierarchical symmetric multiprocessing can be applied to several operating system components, including process communication and scheduling.

Chapters 5 and 6 describe the implementation of the Hurricane memory manager in detail. The design of the data and control structures is developed in two stages. Chapter 5 describes how virtual and physical resources are managed within each cluster; Chapter 6 shows how the structures and communication protocols are extended across clusters to provide an integrated and consistent view of the system.

Finally, Chapter 7 evaluates the performance of our implementation on Hector, a prototype scalable shared memory multiprocessor. The first set of experiments consists of synthetic benchmarks that stress separate aspects of memory management individually. The experiments allow the scalability of the system to be observed in a controlled environment. The second set of experiments evaluate the over-all system performance for real applications. Three different parallel programs with different memory resource demands are run on different cluster configurations to assess the effects of coupling in the system.

Chapter 2

Background

2.1 Introduction

An operating system is arguably some of the most complex software that runs on any computer system. As a resource manager, the operating system must be aware of the hardware resources present, and their demand and service times, if management is to be effective. As a layer between applications and hardware, the system must present a simple, intuitive abstraction of the hardware, and must respond quickly to changing workloads. The demands on the operating system are compounded when the dimension of parallelism is added. The purpose of this chapter is to define common terms and illustrate basic approaches to memory management, focusing on areas fundamental to later developments. The discussion assumes the reader has a working knowledge of sequential computers, but not necessarily of multiprocessors or memory management techniques.

We begin with an overview of scalable shared memory multiprocessor architectures in Section 2.2, focusing primarily on HECTOR, the target architecture of our current implementation. Section 2.3 illustrates the memory consistency problem as it relates to multiprocessors, and reviews an important class of solutions to the problem called directory-based coherence protocols. An overview of virtual memory is given in Section 2.4, which concentrates on architecture and implementation issues for uniprocessor systems. With these fundamentals in hand, the last section opens the door to virtual memory issues on multiprocessor systems, and gives an overview of work related to this important area.

2.2 Scalable Multiprocessor Architectures

Shared memory multiprocessors have existed since the late 1950s [58] but it is only recently that it has become practical to design and build large-scale systems. This newfound practicality has not come from any particular breakthrough, but instead from steady progress in sequential and small-scale parallel processing technologies. Key among these has been advances in component integration, to the point where it is now possible to put an entire small-scale multiprocessor on a single printed circuit board. Higher levels of integration have reduced power and cooling requirements, and helped to shorten wire lengths, which reduces propagation delays and transmission line effects. These technology advances have been exploited commercially, and several companies now offer small-scale multiprocessors based on a single shared bus [12, 44].

Single-bus systems, however, are not scalable; even with large write-back caches the bandwidth of the bus limits their size to a small number of processors. This limitation has spurred renewed investigations into architectures that can scale to larger systems. Currently, the complexity, cost, and increased design cycle time have limited the commercial viability of large shared-memory multiprocessors, particularly since sequential processing power has been advancing so rapidly. Improvements in analysis and design tools have helped reduce the design cycle time, but much work remains before multiprocessor architectures will be able to advance as fast as their sequential counterparts. The remainder of this section steps through several past and present attempts to meet the goal of architectural scalability.

The first scalable shared-memory machines were based on omega networks [59], and include the NYU Ultra-Computer [4], the IBM RP3 [51], and the BBN Butterfly [10]. Each processor in these systems has locally accessible memory, but can access the memory of any other processor through the interconnection network. Because remote memory is accessed through the network, accesses to remote memory take longer than accesses to local memory. For this reason, these machines are called *Non-Uniform Memory Access*, or NUMA, architectures. The remote to local access ratio, ρ, is a key parameter of these machines, and ranges from 13 on the BBN GP1000; to 9 on the IBM RP3; to 3.5 on the BBN TC2000, in the absence of contention. While these architectures were designed to accommodate up to 256 processors, wiring and power constraints made it difficult to build machines larger than 128 processors.

The next generation of architectures overcame the limits of a single bus by connecting many shared bus multiprocessors into a ring or grid topology. The Stanford DASH [41] and KSR [19] architectures, and the HECTOR [65] multiprocessor of Figure 2.1, are examples of this class of architecture. In HECTOR, each processor module contains a microprocessor and hardware cache, on-board memory, and various I/O devices. A *station* is composed of a number of processor modules sharing a common bus, and several stations are connected together via a *local ring*. Finally, a *global ring* connects the local rings together, giving the hierarchical structure seen in the figure.

HECTOR provides a flat, global (physical) address space, where each processor module is assigned a unique range of addresses but can transparently access the memory of any other processor. To allow fast routing and decoding, the address assignment scheme is kept simple: each ring is identified by the most significant r bits of the memory address; the station is identified by the next s bits; and the slot in the station is identified by the p least significant bits. In the current implementation, $r = 2$, $s = 3$, and $p = 3$, for a maximum of 256 processors. To illustrate this addressing scheme, byte 1 on processor 1 of

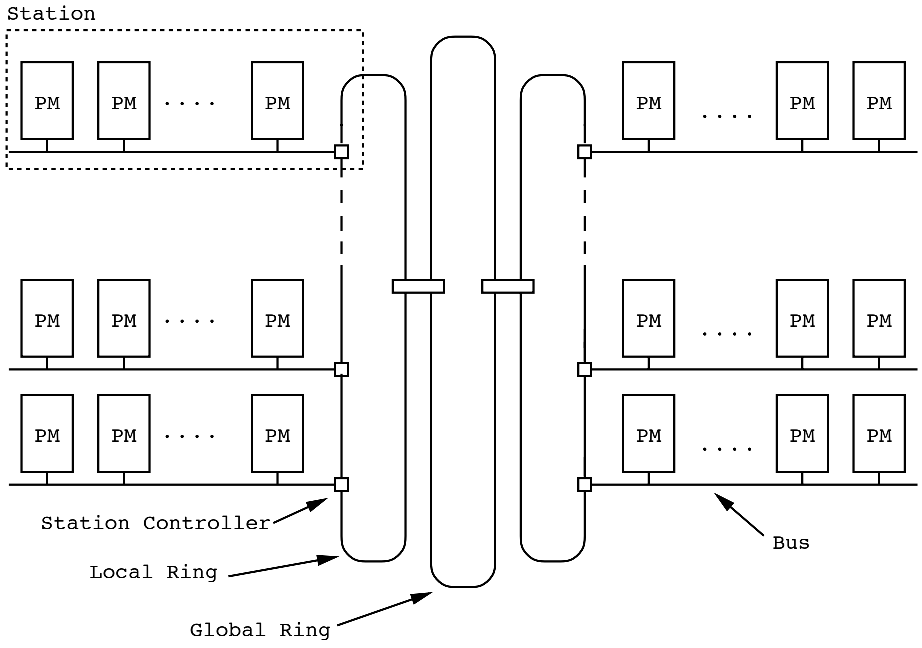

Figure 2.1: The architecture of HECTOR.

station 1, ring 1, can be accessed by any processor in the system at address:

$$\underbrace{01}_{r}\ \underbrace{001}_{s}\ \underbrace{001}_{p}\ \underbrace{0000000\ 00000000\ 000000001_2}_{\textit{local address}} = 49000001_{16}$$

The remote to local access ratio introduced earlier is also applicable to hierarchical architectures, although there is a different ρ for each level of the hierarchy. In HECTOR, the cache and local memory of each processor module can be accessed in 1 and 10 cycles, respectively. The memory of other processor modules within the same station can be accessed in 4 additional cycles (due to station bus arbitration) for a first level remote to local access ratio, $\rho_1 = 1.4$. A read access to a processor module on another station but on the same ring requires this basic remote access time plus an additional cycle per station within the ring, for a remote to local access ratio of $\rho_2 = (T_l + 2T_s + N_s)/T_l$, where T_l is the access time of local memory, T_s is the station arbitration time, and N_s is the number of stations in the ring. The factor of 2 on T_S is because the station bus must be obtained for both the request and the reply packets. These times have all been expressed assuming no contention; the latency is increased if the access must wait for a ring, station, or memory that is busy at the time of the request.

2.3 Memory Consistency

Memory consistency is an issue whenever replicated copies of a data item are potentially modifiable by processes that are accessing the separate copies concurrently[1]. Data is commonly replicated to increase concurrency by reducing contention on the shared item; access latency can also be reduced if the copy is placed physically close to its access point. Examples where data is replicated include multiple caches on a multiprocessor and replicated records in a distributed data base. This section first demonstrates the consistency problem by way of example, and then describes directories, a common method of ensuring consistency. The discussion assumes a basic familiarity with cache architectures; for an overview see Stone [59].

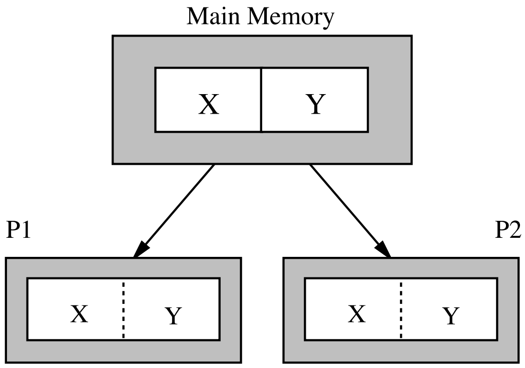

Figure 2.2: A simple multiprocessor with local caches.

The consistency problem can be demonstrated through the simple multiprocessor example of Figure 2.2. The figure shows two processors, `P1` and `P2`, both with local caches, each with a cached copy of the variable `X`. For later discussion, the caches are assumed to have a line size of two words, so that when `X` is read, its neighbor `Y` is loaded as well. If `P1` now modifies its copy of `X`, the value seen by `P2` and the value in main memory are inconsistent with respect to the value seen by `P1`. More formally, the discrepancy in the value seen by `P2` with respect to `P1` is a loss of *cache-cache coherence*; the discrepancy in the value seen by main memory with respect to `P1` is a loss of *cache-memory coherence*.

Cache-memory coherence is restored when the value cached by `P1` is written back to memory, either by a cache line ejection if `P1`'s cache is in *write-back* mode; or by the modification itself if the cache is in *write-through* mode. Cache-cache consistency can be restored by either a *write-invalidate* or a *write-update* mechanism. When `P1` writes `X` back to main memory, a write-invalidate protocol will invalidate the cache line containing `X` in `P2`, so that `P2` must acquire the new value by refetching the line from main memory. Conversely, when `P1` writes `X` back to main memory, a write-update protocol will replace

[1] Non-atomic updates to unreplicated data can also cause problems, but is typically solved using atomic operations supported in hardware.

the stale value in P2 with the current value from P1. Note that while a write-through cache-memory system will update memory immediately, this alone will not ensure cache-cache consistency — the value in P2 must still be updated.

Returning to Figure 2.2, suppose that P1 initially reads X as before, but now P2 reads Y. Both caches will load the cache line containing X and Y, as before. If P1 now modifies X and a write-invalidate protocol is used, the cache line containing X will be invalidated in P2, which will also invalidate Y. Thus, even though P2 never referenced X directly, it is nonetheless penalized by having to refetch its value of Y. This phenomenon is known as *false sharing* and can be a problem if *thrashing* occurs. In the current example, thrashing refers to the repeated invalidations and reloads that would occur if X and Y are frequently modified. Note that thrashing due to false sharing does not occur with a write-update protocol — when P1 modifies its copy of X, the copy in P2's cache is updated with no direct penalty to P2, except that the processor may stall while the update is in progress. However, bus cycles are wasted because the update is in this case unnecessary. Indeed, the implementors of the DEC Firefly, which uses a write-update coherence protocol, found that wasted updates could consume a large portion of the available bus bandwidth unless special effort was made to avoid leaving unnecessary copies in the cache [63].

A key issue in coherence is how to locate those caches containing a particular cache line so that the cache line can be updated or invalidated. Many mechanisms and architectures have been proposed, but most fall under the general classifications of *snooping* and *directory-based*. A snoopy cache contains the functionality necessary to watch the interconnection network for modifications to data that it is caching, so that it can know when to invalidate or update a stale entry. In effect, each processor is responsible for keeping its own cache consistent with memory and the caches of all the other processors. Consequently, architectures based on snooping caches are not scalable, because all processors must observe all memory transactions in case some local action is required.

Directory schemes avoid this problem by keeping the consistency state of each memory line in a well known location, called the directory entry. In its simplest form, called a *full-map* directory, each entry contains a bit per processor (the *copy set*) and a *dirty* bit to identify whether or not an item has been modified. The entries are kept in a table at a well-known location, so that when a processor accesses a data item it first checks the dirty bit to see if the line can be cached. If the line is clean (and the access is a read), the processor loads the data and adds itself to the copy set. If the data is subsequently modified, the dirty bit is set and the caches of all the processors in the copy set (except the modifying processor) are invalidated and their bits are cleared from the copy set.

There are many variations on the basic full-map directory approach. The primary motivation for these variations is that the space cost of the directory as described increases rapidly as the system grows, because there is a directory entry per memory-line, and each entry requires a bit per processor. To reduce the memory requirements, a *limited* directory keeps the sharing information for only a few processors. If the copy set is found to be full, other processors may access the data but not cache it, or they may cache it and set an overflow bit, which forces a broadcast invalidate to all processors whenever the data is modified. The MIT Alewife project [21] employs a hybrid hardware/software approach, in which the hardware implements a limited directory, and overflows are extended in software to support a full-map scheme. Still other proposals use a chain of processor identifiers to allow variable sized copy sets [32].

It is important to note that while the examples above were developed in the context of hardware caches in a multiprocessor environment, the same principles apply whenever shared data is replicated. For example, directories can also be used to support virtual shared memory in a distributed system [61].

This section has shown how concurrent accesses to replicated data can lead to inconsistency. We have outlined the operations needed to restore consistency, and we have described directories, which contain the state needed to support coherence. The development has so far avoided a discussion of consistency models themselves. Simply put, the consistency model defines the acceptable orderings among memory requests. The strictest form is called *sequential* consistency [34], and can be guaranteed by requiring a processor to complete one memory request before it issues the next memory request [56]. This model allows the execution of a parallel program to appear as an interleaving of the parallel processes on a sequential machine. Sequential consistency is conceptually appealing, but imposes a large performance penalty on memory accesses. *Weak* [55] and *release* [30] consistency models relax the restrictions to allow more concurrency in memory accesses, but they require the programmer or compiler to explicitly identify all synchronization accesses.

From the point of view of the operating system, the consistency model used depends on the support provided by the architecture, and the consistency requirements of the applications. This is because the operating system can provide sequential consistency semantics even if there is no hardware support for coherence at all, using software techniques described later. Conversely, even if the hardware supports full coherence, the operating system can often disable this feature at the request of applications that want to manage their own data coherence. The philosophy adopted in Hurricane is to provide a basic coherence policy that is suitable for the common case, but which can be overridden by sophisticated applications.

2.4 Virtual Memory

Virtual memory is a powerful concept that separates logical memory references from the physical addressing used by the underlying hardware. Originally developed as a way to address a storage space larger than the amount of physical memory, virtual memory techniques are now applied to many demand-based memory management policies, including copy-on-write and software memory coherence. This section gives a brief overview of the hardware needed to support a virtual memory system on a typical uniprocessor machine.

Although addresses in a virtual memory system are disassociated from their physical counterparts, any memory reference must eventually be resolved to the physical address at which the data resides. The relationship between a virtual address and the corresponding physical address is held in a *translation entry.* To reduce the number of translation entries, main memory is usually grouped into fixed-sized *pages*, where each page is a contiguous block of 2^p bytes. The v bits of a virtual address and r bits of a physical address are both treated as having two fields: the *page number* is the most significant $v-p$ bits of a virtual address, or $r-p$ bits of a physical address; the *page displacement* is the least significant p bits of both types of address[2]. In a paged system, one translation entry is used for each page. The translation entries are searched by virtual page number; the complete physical address is formed by concatenating the page displacement to the physical page number found in the translation entry.

The page size is an important architectural parameter. A large page size will reduce the number of translation entries and will reduce the number of translations required, because one translation entry serves 2^p different addresses. In addition, if pages are loaded from secondary store, a large page size can reduce the number of accesses to disk because each access transfers a large amount of data. However, a large page size may increase fragmentation if all the data in a page is not needed, effectively reducing memory utilization. As well, permissions and translation protection are at the granularity of the page, so a large page size may limit sharing or protection flexibility.

Many data structures have been used to maintain translation entries — the most common structure is a simple table indexed by the virtual page number, where each entry in the array is the physical translation of the virtual page number that indexes it. This type of translation structure is called a *page table*. Linear page tables tend to be space inefficient, because they require an entry per virtual page, even if the entry for that page is not used. Some systems address this problem by using a two-level page table, where the first level is indexed by the high-order bits of the virtual page number, and the low-order bits index the second level of the page table. The first-level entries contain the address of a linear page table, which forms the second level of the structure. Two-level page tables can save space because the second level array is not needed if the first level index is unused. However, as processor architectures move to 64-bit addressing, even two-level page tables have high space cost if the address range has relatively few valid translations. To overcome the problems of linear page tables, *inverted page tables* only keep translation entries for the physical pages accessed by a program. An inverted page table (IPT) consists of a hash table keyed by virtual address, where each entry in the table is a list of virtual to physical translation records. For sparsely populated address space ranges, an IPT can reduce the amount of memory devoted to translation entries because entries exist only if a particular virtual address is actually referenced by the program.

Since every virtual address must be translated before it can be used to access physical memory, and since the table look-up requires at least one or two memory accesses to perform the translation, the system as described has an unacceptably high overhead. To reduce this overhead, most systems have a *Translation Look-aside Buffer*, or TLB, which is a cache of recent translations. This cache is searched on every translation, so that the

[2] There are 2^v addressable items in the virtual address space, and 2^r items in the physical address space; and typically $v \geq r$. Thus, there are 2^{v-p} and 2^{r-p} pages in the virtual and physical address spaces, respectively, if the page size is 2^p.

page table look-up is needed only on a TLB miss. The TLBs must translate addresses as fast as the processor can request them, which means the TLBs must be implemented with fast logic (and high cost) that grows with the number of entries cached. Fortunately, since each entry can serve an entire page of data, and since programs typically access small groups of addresses at a time, the size of the TLB can be relatively small and still achieve a high hit rate.

To provide protection in a multi-programmed environment, ie. when multiple programs are executing concurrently, the virtual addresses of each program are commonly translated from a separate set of tables. The set used to translate a given address space is called its *context*. In this way, programs only share data if their virtual addresses translate, or *map*, to the same physical address. For protection within an address space, most translation entries maintain a set of access permissions for each translation entry. These permissions may specify, for example, read-only access so that data in that page is protected from inadvertent modifications; the permissions may also specify no access to identify an uninitialized translation entry. These permissions are checked by the hardware on every translation — if a violation is detected, an exception (or *fault*) is raised on the CPU for handling by software. This dynamic permission checking forms the basis for *demand-driven* memory management policies, which allow decisions to be deferred until a fault occurs. Demand-driven policies are important because they allow the system to respond dynamically to changes in memory demand.

Virtual memory was initially developed to address several problems with *real memory* systems, which were based on physical memory management techniques such as overlays and swapping [25]. Real memory systems require that the code and data segments of a program be resident in contiguous physical memory before it can run. Often these systems have no protection between concurrently executing programs, although some provide rudimentary protection using Bounds registers or ID tags. Consequently, architectural and operating system specific details such as memory size and organization are visible at the application level, which complicates program development.

In contrast, a paged virtual memory system requires only those pages actively referenced by a program to be memory resident, and these logical pages may be discontiguous in physical memory. Protection between programs is supported in hardware through multiple page table contexts. The separation of logical storage from its physical counterpart enhances portability, simplifies application development, and allows memory to be managed efficiently. These advantages are demonstrated through the examples given below.

Simplicity of application development in a VM system is demonstrated by the transparent support of programs whose data requirements exceed the size of main memory. Real memory systems require the programmer to explicitly manage the memory of these large programs. In contrast, the memory manager in a virtual memory system supports large programs by keeping only the most recently referenced pages resident in memory — the remainder of the program's pages are left on secondary store and their translation entries are marked invalid. If a page that is not currently resident is referenced, the translation will fail and a page fault is generated to pass control to the operating system. The fault is handled by loading the data from secondary store into an unused physical page and setting the translation and protection entries so that it can be accessed by the application. This approach to memory management is called *demand-paging*, because the pages are not loaded until they are referenced. If all physical pages are in use at the time of a fault, one of the pages is selected for page-out as determined by the *replacement policy*. The current contents of the page are saved to secondary store (if the data has been modified) and all virtual translations to it are invalidated, freeing the physical page for use by other virtual pages.

Keeping only a subset of a program's data resident in main memory does not necessarily hurt execution times, and in fact can lead to improved system response time in multi-programmed environments. Denning argued that a program will run efficiently if its actively accessed set of pages, its *working set*, is resident [27]. This principle arises from the observation that the memory access patterns of programs exhibit *temporal* and *spatial* locality. Temporal locality means that locations accessed recently are likely to be accessed again in the near future; spatial locality means that locations near a recently accessed location are likely to be accessed in the near future.

Locality is shown in common control structures such as loops or sequential execution, and in data structures such as stacks and arrays. The working set theory maintains that as long as the favored subset of pages remains in memory, the program will experience a minimum number of page faults, and therefore execute efficiently. If the working set has not yet been established or if it changes as the program executes, the fault rate will increase until the pages of the new working set are resident. Finally, if the working set is larger than physical memory, or the sum of the working sets of all currently executing programs is greater than physical memory, *thrashing* may occur as the pages fight to remain resident. Thrashing is a condition where the system spends all its time moving data, so that little or no progress is made by the application [26], but note that under the same memory requirements a real memory system will also fail[3].

The principles of working set theory support the argument that virtual memory can help increase performance in multi-programmed systems. One reason is that more working sets can be resident at a given time than in real memory systems that must load a larger portion of each program's data. A second source of improvement is that the amount of swapping in a virtual memory system is typically much less than in a real memory system. If memory is highly utilized, a swapping system will make room for a scheduled task by saving the entire memory image of a currently resident program to backing store. This context must be restored to main memory before the program can be restarted. Under the same high utilization conditions, a paged virtual system can normally avoid swapping by freeing memory incrementally through page-outs.

Virtual memory systems have several properties that can lead to improved memory utilization. First, virtual memory permits sharing of immutable data between programs. Sharing is accomplished transparently by translating virtual addresses in separate address spaces to the same physical address, and is particularly useful for code segments and shared

[3] Thrashing becomes less probable as memory sizes increase.

libraries. Second, pages that are contiguous in virtual memory need not be contiguous in physical memory, so memory fragmentation is limited to the size of a page and the allocation policy is greatly simplified. Third, the hardware permissions checking can be used for optimizations such as copy-on-write [47]. This technique improves memory utilization by sharing mutable data until it is explicitly modified, at which time a copy of the shared page is made that reflects the modification. Copy-on-write can reduce system overhead because if the shared page is never modified, the copy can be avoided entirely. Copy-on-write is discussed in more detail in Section 5.7.

2.5 Related Work

Previous sections have given some background in parallel architectures and virtual memory techniques. This section ties them together by discussing issues in virtual memory support on parallel architectures, and lists some of the research groups investigating these issues. The advantages of virtual memory have led to its wide-spread application on multiprogrammed uniprocessor systems, and the same advantages apply in large part to multiprocessor systems. However, the added dimension of multiple processors and memories with different access latencies creates several challenges.

First, *page placement* is an issue on NUMA[4] architectures. Page placement refers to the decision of which physical page should be assigned to a new virtual page. On UMA architectures, where all memory has the same access time, the physical page can be selected solely on the basis of longest residence time since last access. On NUMA architectures, the decision is also affected by locality considerations, since the access time and interconnect contention can be reduced if the data is placed near where it will be used. The placement decision is more difficult when several processors must access the page — in some instances it may be beneficial to replicate the page to place the data close to all the processors that access it. Page placement is further complicated by changing reference patterns, which may force a re-evaluation of the placement or replication decision. Re-evaluation may be required when the process that was accessing the page migrates to a different processor, or if the data is used in a producer-consumer relationship where the producer and consumer are on different processors. The factors that influence these replication and migration policies have been studied by investigators at Duke [37], Rochester [17], Rice [20], and Stanford [15].

The added dimension of multiple memories can also affect page-out decisions. If the physical pages of a memory module are all in use, the page-out policy selects the page that will be written to backing store to make room for a new page. On UMA architectures, the decision is often based on temporal considerations alone, and is often implemented using a Two-Handed Clock algorithm [40] (essentially a LRU algorithm). On NUMA architectures, spatial factors can come into play, since the page targeted for page-out could be moved to

[4] Recall that NUMA Systems are systems with non-uniform memory access times. In contrast, UMA systems have uniform memory access times.

a less utilized remote memory, or instead of page-out, a new page could be allocated from a less utilized remote memory [35]. Accesses to the remote page have higher latency, but the impact can be less than if a page has to be moved to secondary store.

As alluded to earlier, virtual memory can be used to support cache and/or memory coherence if not implemented directly by the hardware. This technique is generally referred to as *Distributed Shared Memory* [61] and has been applied to memory coherence for a parallel environment at Rochester [24], and for a distributed environment at Yale [43], Rice [11] and Toronto [67]. However, HURRICANE is the only system we know of that supports both memory and cache coherence in an integrated way.

Even if the hardware supports cache or memory coherence, most machines do not implement a TLB consistency protocol in hardware. Since the TLB is really a cache of page table entries, and since there is usually one TLB per processor, the entries must be kept consistent or processes that are sharing virtual memory pages may see incorrect translations or permissions. The most common consistency protocol is called a *TLB Shootdown* [14], and several groups have published their implementations of the protocol [54, 64, 7]. HURRICANE uses a different approach with some interesting trade-offs, and is described in Section 5.6.

The challenges of implementing virtual memory in a parallel environment described so far have been largely functional, although they certainly have performance implications. Another challenge, and one that we address directly, is how to maintain both function and performance as the system scales in size. The parallel operating systems cited from Duke and Rochester teach us little about scalability, since the efforts were focused on policy issues. As well, the work was based on extensions to existing UMA operating systems, and was thus constrained by the existing framework of the underlying UMA system.

The focus on scalability has been more prevalent in distributed systems, where systems like V [23], Mach [2], Sprite [49] and Amoeba [46] have been targeted to run on hundreds of processors. Research in these systems has been primarily concerned with scalable file systems [31, 42] and process scheduling [9, 60].

Scalability issues in distributed shared memory have been explored by the Ivy [43] and Munin [20] systems with a limited number of processors, and we have drawn on their experience where possible. However, in applying distributed system principles to a shared memory multiprocessor, a number of fundamental architectural differences force the re-evaluation of many design decisions:

- The lower communication costs on a shared memory multiprocessor allow much tighter coupling. This is especially important since we expect a finer granularity of sharing by applications in a multiprocessor environment.

- Distributed systems do not have true shared memory, which restricts the data structures and communication protocols that may be used.

- The broadcast based algorithms supported by the network in distributed systems are not generally applicable on shared memory multiprocessors, because most architectures do not support broadcast in hardware.

The remaining chapters of this dissertation discuss issues in scalability and describe how HURRICANE approaches both functional and performance issues in this context.

Chapter 3

Hierarchical Symmetric Multiprocessing

This chapter develops hierarchical symmetric multiprocessing as a structuring technique for the design of scalable operating systems. The discussion begins in Section 3.1 with a review of three basic performance metrics: throughput, utilization, and response time. The behavior of the performance metrics in a large system is examined in Section 3.2, and used to develop a set of properties sufficient for scalability. These properties form the basis of a set of guidelines, presented in Section 3.3, for structuring a scalable operating system. Section 3.4 describes the basic concepts of hierarchical symmetric multiprocessing, and shows how it meets the design guidelines. The chapter concludes with examples of how the principles of HSM are applied in the Hurricane operating system.

3.1 Performance Metrics

This section reviews the definition of several performance metrics and discusses their properties in a scalable system. The terminology and fundamental equations are as presented by Lazowska [38].

The discussion examines performance metrics for servicing the requests of *customers* at *service centers*. We shall denote a particular class of customer as c, and a particular service center, or *resource*, as k. Since a single service center can potentially support more than one class of customer, the index ck is used to denote the customer class c at center k.

The notion of customers and service centers is applied to the operating system at two levels. At one level, applications can be considered customers of the services provided by the operating system. For example, the operating system supports the services of creating a new process, or handling a page fault. The second level of customers and resources is within the operating system itself. At this internal level, the services are operations like acquiring a lock, or allocating a descriptor for a physical page. The resources in these examples are the lock and the list of available page descriptors, respectively. The customers requesting the service of the internal level resources are the servers and fault handlers of the operating system.

Three fundamental metrics of computer performance are throughput, utilization, and response time. The metrics are important because they are observable quantities and have practical interpretations.

Throughput (X) is defined as the number of requests completed per unit time. An example of throughput is the number of page faults that can be handled per second. Throughput can be applied to any service center in the system, where a service center is a resource as simple as a lock, or as complex as the system itself. The throughput for a customer class depends upon the arrival rate of requests, λ_c, and on the number of visits to each service center, v_{ck}, that are required for the request to complete. This dependence is expressed by the *Forced Flow Law*:

$$X_{ck} = v_{ck} \cdot \lambda_c \tag{3.1}$$

For a parallel operating system, we intuitively expect the request rate to increase with the number of processors, either because there are more sequential applications, or because the parallel applications require more resources. However, we also expect the number of service centers to increase as more processors are added, so that the throughput of each resource need not necessarily be higher than for a sequential system. This consideration is important in the development of the scalability criteria that follow.

Utilization (U) is defined as the percentage of time that a resource is busy servicing requests. As an example, a lock that is 50% utilized is actively set half of the time. The utilization of a particular service center is related to throughput by the following equation:

$$U_{ck} = s_{ck} \cdot X_{ck} \tag{3.2}$$

where s_{ck} is the time required to service a request of type c at resource k. Over all customer classes, no resource can be busy more than 100% of the time, so that $1 \geq U_k = \sum_c U_{ck}$. This limit also constrains the throughput at that resource for each customer class.

In any system, the resource with the highest utilization is called the *primary bottleneck*; this is the resource that is limiting further increases in throughput. Throughput can be improved by reducing the utilization of the primary bottleneck, either by reducing its service time or the number of requests it must service. These observations are the basis of the "lock-splitting" approach to scalability described earlier, where the most heavily utilized lock is replaced by several locks, each with (a hopefully) lower utilization.

It is important to recognize that the utilization of a particular resource is workload specific: the primary bottleneck for one class of applications may be different from that of a second class of applications. The classical examples here are CPU bound tasks, for which the processor is the primary bottleneck; and I/O bound tasks, for which a disk may be the most heavily used resource.

Response Time (R) is the amount of time a request must wait for completion. An example of response time is the time it takes to run a sequential program. In parallel environments, speedup is often given in preference to direct response times. Speedup on n processors is given by:

$$S(n) \;=\; R(1)/R(n) \tag{3.3}$$

where $R(1)$ and $R(n)$ are the response times for a given application on 1 and n processors, respectively.

The response time at a service site seems implicitly tied to service time, and indeed they are equal if there is no queuing. Queuing is avoided if the request is serviced as soon as it arrives. Program speed-up (ie. $S(n) > 1$) is usually due not to reduced queuing or service time, but because more service centers are involved, so that the n times increase in demand is serviced simultaneously at n centers, with time proportional to that of a single center.

We conclude this section with a discussion of the terms *granularity* and *coupling* as applied to parallel applications. Granularity is a measurable quantity that has a direct impact on the performance of a given algorithm on a particular architecture. Granularity can be defined as the amount of compute time between communication points; coupling is the dual notion defined as the amount of communication per compute time. Each process in a fine-grained or tightly-coupled program does only a small amount of local computing (tens of instructions) before it must communicate with other processes; a coarse-grained or loosely-coupled application performs thousands of local operations between each communication. The form of communication can be as complex as a message send or barrier synchronization, or as simple as setting a global variable. The important point is that communication involves the interconnection network, and is therefore subject to network latency and contention effects.

The concept of granularity is often applied to architectures, and usually refers to the granularity of applications that can be run efficiently. An architecture can support fine-grained applications if the cost of communication is low, because these applications require low latency and a relatively large communication bandwidth to perform well. Thus, a distributed system is considered coarse grained, because the cost of communicating over the network is high.

3.2 Sufficient Conditions for Scalability

It is surprisingly difficult to give a formal definition that characterizes scalability in a practical sense, especially for operating systems. For example, one of the better known formal definitions is by Nussbaum and Agarwal [48], which states: *The scalability of a machine for a given algorithm and problem size is the ratio of the asymptotic speedup on the real machine and the ideal realization of an EREW PRAM.* While the principles embodied by the definition are important, the definition cannot be applied directly to operating systems for several reasons. First, the operating system is specifically excluded from consideration and is treated as an extension of the hardware.

Second, scalability is expressed in terms of inherent parallelism and asymptotic limits, which is difficult to apply to real systems of a given size and unknown inherent parallelism. Even if the scalability of a system could be determined, a quantitative measure yields no insight about how to design or build a scalable system.

Finally, the definition is based on speedup, which we do not believe to be appropriate for operating systems. For parallel systems, we intuitively do not expect the response time of an operating system call to decrease as more processors are added; rather, we are content if the response time remains constant as processors are added to the system. We believe that throughput and concurrency are the dominant issues in scalable operating system design. Since one can expect the demand on the operating system to increase in proportion to the size of the system, the throughput must also increase proportionally. This is possible only if the operating system has sufficient concurrency to meet the demand.

Instead of attempting to develop a more appropriate definition of our own, we instead identify a set of properties sufficient for an operating system to scale. The discussion assumes that the arrival rate of requests for service, $\lambda_c(p)$, increases proportional to the number of processors in the system, p. In addition, we assume that application service requests are distributed across the processors over time. The assumption of distributed requests is an implicit result of Nussbaum's definition of scalability, because if the inherent parallelism of the application does not match the capabilities of the machine, then the ratio of the speedups will show a lack of scalability. By extending Nussbaum's definition to make the assumption of distributed requests explicit, we recognize that the hardware, operating system, and applications must all work together if the system is to scale, and that an imbalance in any one level can severely degrade performance.

In a parallel environment, the utilization law of Equation 3.2 can be expressed as:

$$1 \;\geq\; U_k(p) \;=\; \sum_c U_{ck}(p) \;=\; \sum_c X_{ck}(p) \cdot s_{ck}(p) \tag{3.4}$$

where the components of the original equation are allowed to become general functions of the number of processors. If the system is to scale in p, then resource k cannot saturate, or it will become a bottleneck that potentially limits performance. This means that the utilization of a resource must be bounded by a constant independent of the number of processors. Similarly, the throughput and service times at resource k can only be functions of p if they are inversely proportional, so that their product is a constant less than 1. In the following, we consider the implications if

i. s_{ck} is independent of p,

ii. s_{ck} increases with p, and

iii. s_{ck} decreases with p.

The discussion considers the class specific terms (s_{ck}, X_{ck}, etc.) because if any one of the class specific products in Equation 3.4 is not constant, it will dominate the sum and cause the resource to saturate.

Constant Service Time

We first consider the case where the operating system service time, s_{ck}, is independent of the number of processors in the system. If service times are constant, then operations such as creating a process or mapping a page, do not take longer to complete as the system grows. Keeping the service time constant as the system grows seems an intuitive goal, albeit challenging to attain. It also means that the throughput, X_{ck}, must be independent of p. From the Forced Flow Law (Equation 3.1), the throughput for customer class c at resource k is proportional to both the system wide arrival rate and the visit count to the resource. For parallel systems, we expect λ_c to increase linearly in the number of processors. Since X_{ck} is to remain constant, v_{ck} must decrease as $1/p$. To see how it is possible to have $v_{ck} \propto 1/p$, define V_c as the total number of visits across the system that are needed to satisfy requests of class c. If there are K_c resources to service this class of customers, then

$$V_c \;=\; \sum_{K_c} v_{ck} \tag{3.5}$$

Now, if all v_{ck} within a group of resources are equal, then the summation of Equation 3.5 can be replaced by a product:

$$V_c \;=\; K_c \cdot v_{ck} \quad \text{or} \quad v_{ck} \;=\; \frac{V_c}{K_c} \tag{3.6}$$

Thus, if K_c grows at the same rate as λ_c, then $v_{ck} \propto 1/p$, assuming V_c remains constant. Equation 3.6 expresses v_{ck} as an *average*; it is likely not true at a given instant for a particular resource, but it must be true on average over time (and resources) if the system is to scale with fixed service times.

In essence, Equation 3.6 requires a system to be balanced in its service capabilities to match the expected workload requirements. Note that to meet this requirement, much of the burden of responsibility falls on the process scheduler to balance the load across the system[1].

Increasing Service Time

We now consider the case where service times grow as processors are added to the system. This situation can occur, for example, if a global unordered list is used that must be searched linearly to service the request. Since the list is global, the number of elements, and therefore the search time, can grow proportionally with p. The increasing service time

[1] Of course, a poorly designed or malicious program can always negate the best design efforts.

forces the throughput at the resource to decrease if saturation is to be avoided, which from Equation 3.6 requires that the number of resources increase faster than p. This requirement is neither reasonable nor desirable, and data structures whose access time depends on p must be avoided if the system is to scale.

A second scenario that may result in service times increasing with p are requests that require a single operation on many resources. These *compound* services include requests to destroy a parallel program containing many processes, or to invalidate data that has been replicated across several processors. Because compound services are requested once on behalf of the n processes in the program, we do not expect their arrival rate to increase linearly in the number of processors, but rather more like $\lambda'_c \propto p/n$. Thus, if the service time increases proportional to n, then the corresponding throughput, X'_{ck}, can decrease by the same amount because, from substitution of Equation 3.6 into Equation 3.1:

$$X'_{ck} \;=\; \lambda'_c \cdot v_{ck} \;\propto\; \frac{pV_c}{nK_c} \tag{3.7}$$

where K_c is still required to grow proportional to p, as before. Optimistically, one might hope to exploit parallelism in servicing compound requests, since there are many resources involved. Parallelism is possible if the operating system can view the request for the service on n different resources as n independent, but simultaneous, requests for service to n different resources. This is only possible if the resources in the compound request are independent. In real systems, however, they are often coupled through shared resources such as locks. This coupling increases the demand on the shared resources, thus limiting the speedup that can be obtained.

Decreasing Service Time

We now consider the case where service times decrease as a function of p. In this scenario, since λ_c is increasing with p, the service time of operating system services, such as creating a process or mapping a page, would have to have a service time $s_{ck} \propto 1/p$. But we argue that this case is unrealistic. Service times can only decrease as $1/p$ if the operation has speedup proportional to p. To obtain speedup proportional to p, the service must be partitioned into p independent operations that can be executed in parallel. However, such a partitioning is not possible unless the resources involved in servicing the request increase with p, which is not true of the service requests under consideration.[2]

Another reason why it is unrealistic to have a service time $s_{ck} \propto 1/p$ is that the assumption of increasing λ_c immediately precludes the possibility of applying multiple processors to the servicing of a particular request, because each processor must be available to service its share of the system-wide load. Even if all processors were involved in servicing each request, so that $s_{ck} \propto 1/p$, each processor is now involved in all λ_c requests, instead of

[2] It is important to note that parallelism in servicing a single request is different from the compound service requests discussed earlier, since compound requests are effectively single operations performed on many similar resources.

λ_c/p requests if the requests are handled independently. This coupling is in fact precluded by Equation 3.6, which assumes that V_c is constant as the system grows. If all the processors are involved in servicing a request, then the visit count would grow as a function of p, which offsets the benefit obtained by increasing K_c. Using parallelism in servicing single requests has the added disadvantage of preventing applications on the other processors from making progress while the request is serviced.

This section has argued against parallelism in the servicing of a single operating system request. Instead, the servicing of requests should be localized, in that no processor should unnecessarily involve other processors in the course of its duties.

Summary

The previous discussion can be summarized as follows. Assuming that the service request rate, λ_c, increases linearly with the number of processors in the system, p, then a system will scale if it possesses the following properties:

1. The time spent servicing request c at resource k, s_{ck}, must be bounded by a constant independent of p, the number of processors in the system.

2. The number of resources available to service a request of class c, K_c, must increase proportional to p.

3. The system must be balanced in its service capabilities, so that $v_{ck} = V_c/K_c$ on average.

4. The servicing of individual requests must be localized and independent, such that $V_c = \sum_k v_{ck}$ is bounded by a constant.

3.3 Design Guidelines

The properties listed above clearly identify the conditions sufficient for a system to scale, but they do not directly specify how the requirements can be met. This section translates the requirements into a set of design guidelines that form the fundamental structuring goals of hierarchical symmetric multiprocessing.

Preserving parallelism: *The operating system must preserve the parallelism afforded by the applications.* Because we do not expect parallelism in servicing a single operating system request, and because an operating system is primarily demand driven, parallelism within the operating system can only come from application demand. If several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, then they must be serviced in parallel. This demand for parallel service can only be met if the number of operating system service points increases with the size of the system, and if the concurrency available in accessing data structures also grows with the size of the system.

Bounded overhead: *The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors* [8]. This guideline follows directly from properties 1 and 4 above. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system-wide ordered queues cannot be used and objects must not be located by linear searches if the queue lengths or search lengths increase with the size of the system. Instead, structures that support search in order constant time, such as static positioning (for example, arrays), must be used.

The principle of bounded overhead is also applied to the space costs of the operating system data structures. While the data structures are required by property 2 to grow proportional to the physical resources of the hardware, the principle of bounded space cost restricts growth to be no more than linear. In particular, the size of memory management data structures cannot depend on the number of virtual resources; their size should depend only on the amount of physical memory [1].

Preserving locality: *The operating system must preserve the locality of the applications.* It is important to consider the memory access locality in large-scale systems, because for example, many large-scale shared memory multiprocessors have non-uniform memory access (NUMA) times, where the cost of accessing memory is a function of the distance between accessing processor and the target memory, and because cache consistency incurs more overhead in large systems. Locality can be increased a) by properly choosing and placing data structures within the operating system, b) by directing requests from the application to nearby service points, and c) by enacting policies that increase locality in the applications' memory accesses and system requests. For example, policies should attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and direct file I/O to devices close by. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors that must access it to handle page faults.

3.4 HSM Architecture

Small-scale multiprocessor operating systems achieve good performance through tightly coupled sharing and fine-grained communication. Tightly-coupled systems can scale if the locks can be kept from saturating, which becomes increasingly difficult as the system grows. In the framework of the design guidelines above, a saturated system fails to preserve the parallelism of the applications it is trying to support. In addition, the shared data structures common in tightly coupled systems, such as the process descriptor table, typically increase in size with the size of the system. While these tables can be distributed across the processors of the system, it is difficult to preserve locality, which results in increased network contention.

Distributed operating systems appear to scale well through replicated services that distribute demand and avoid centralized bottlenecks. Data is also distributed and cached locally to reduce the amount of remote communication needed. The high communication costs in these distributed environments dictate coarse-grained parallelism and loosely coupled synchronization through message passing.

Both structuring approaches are appropriate for their target environment. Both approaches also offer solutions to different problems in the design of an operating system for a large-scale multiprocessor. On large-scale multiprocessors, the relatively low cost of accessing remote memory should encourage fine-grained data sharing between small groups of processors; for the system as a whole, decentralizing resources seems imperative for locality and scalability. Hierarchical symmetric multiprocessing is a hybrid structuring approach that adopts features of both distributed and tightly-coupled systems to achieve scalability while still maintaining good performance.

The basic unit of structuring within HSM is the *cluster*, which consists of a symmetric micro-kernel, memory and device management subsystems, and associated system services such as a scheduler and file server. Data and control structures are shared by all processors within a cluster, giving good performance through fine-grained communication. Thus, the cluster provides the functionality of a small-scale symmetric multiprocessor operating system. On larger systems, multiple clusters are instantiated such that each cluster manages a unique group of "neighboring" processors, where neighboring implies that memory accesses within a cluster are generally less expensive than accesses to another cluster. Clusters cooperate and communicate in a loosely coupled fashion to give applications an integrated and consistent view of the system.

By incorporating structuring principles from both ends of the performance spectrum, hierarchical symmetric multiprocessing realizes the following advantages:

- Hierarchical symmetric multiprocessing provides a framework for managing locality, since system services are replicated on a per cluster basis. For the common case of small-scale parallel programs, all processes are scheduled onto the same cluster. This enhances performance as all interactions are local. Large-scale applications are scheduled across multiple clusters, and can benefit from the concurrency afforded through replicated system services. Support of large applications without penalizing small ones is one of the basic design goals of HURRICANE.

- Hierarchical symmetric multiprocessing enhances portability by allowing performance tuning to different architectures. The appropriate cluster size for a given architecture is affected by several factors, including the local-remote memory access ratio, the hardware cache size and coherence support, and the network topology. On hierarchical multiprocessors such as Hector or Dash, a cluster might correspond to a hardware station; on a local-remote architecture such as the Butterfly, a small cluster size (perhaps even a cluster per processor) might be more appropriate. In this case, HSM can be viewed as an extension of the fully replicated structuring typically used on these machines.

- Finally, hierarchical symmetric multiprocessing may simplify lock structuring issues, which can lead to improved performance and scalability. For example, Chaves [22] reports that the fine-grained locking used in an unclustered system significantly increases the time of the critical path, even when there is no lock contention. As well, deadlock can be a problem when several fine-grained locks must be held simultaneously. Because contention for a lock is limited to the number of processors in a cluster, hierarchical symmetric multiprocessing may allow coarser grained locking.

The implementation of a hierarchical symmetric multiprocessing system presents several challenges. First, clusters should be integrated seamlessly into the system, so that the average application need not be aware of their existence. Of course, sophisticated applications may want to exploit the presence of clusters to enhance performance. Second, it is important that the overhead and complexity introduced by clusters be kept to a minimum. HSM was conceived as a way to reduce overall system complexity and increase performance; these gains may not be realized if the cost of remote communication is too high. Finally, hierarchical symmetric multiprocessing does not by itself guarantee scalability, rather, it provides a framework that allows structuring for scalability. The challenge is to use this framework to reduce system bottlenecks and provide increased system bandwidth as the system grows.

3.5 HSM Applications

In this section, we show how hierarchical symmetric multiprocessing affects the structure of the primary operating system components. While the principles of HSM are applicable to many operating system philosophies, the discussion is presented using examples from the Hurricane operating system. In particular, the structure of four key components of the operating system is described and related back to the design guidelines. The four components are the kernel, the memory manager, the file system, and the scheduler, as shown in Figure 3.1.

3.5.1 The Hurricane Kernel

The kernel is responsible for process management and communication. Processes are represented by process descriptors, which are kept in a hash table for fast access. Most operations on processes involve the queuing and dequeuing of process descriptors to and

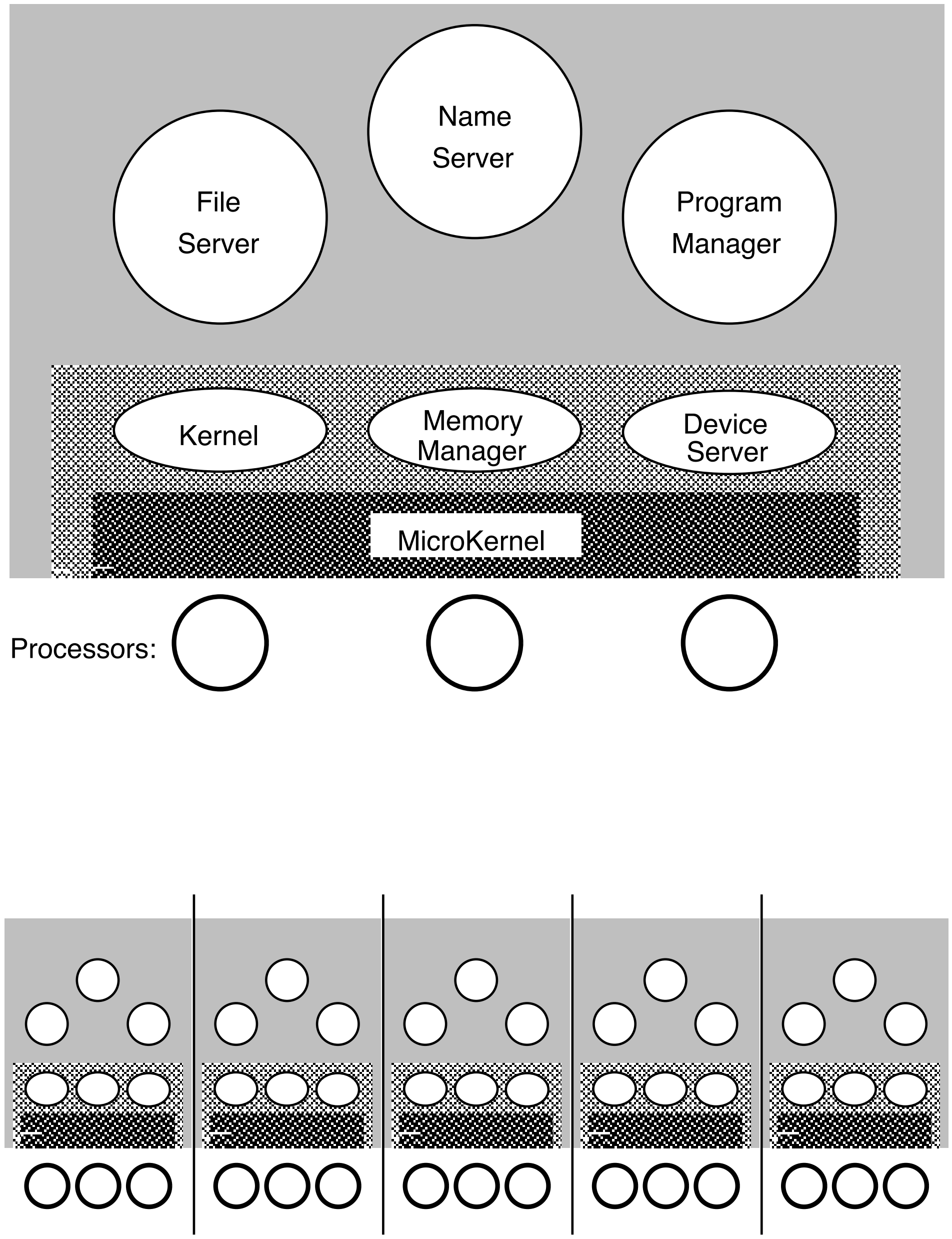

Figure 3.1: Each cluster (top) provides the functionality of a small-scale symmetric multiprocessor operating system. Multiple clusters cooperate and communicate to provide an integrated and consistent system (bottom).

from different queues. For example, when a process sends a message to another process, its descriptor (containing the message) is added to the Message Queue of the target descriptor. Other queues include a Ready Queue for processes that are ready to run, and a Delay Queue for processes waiting for a time-out. The process descriptor table and kernel queues are local to each cluster, which preserves locality for the common case where the request can be handled entirely within the cluster. Moreover, the number of queues increases with the number of clusters, so that the overhead of queue operations remains bounded by a constant.

The kernels of each cluster in the system communicate and cooperate in order to provide the processes and users a consistent view of the system. When a process is created, it is assigned a process identifier within which the id of the *home* cluster is encoded. Usually, a process remains within its home cluster, but if it migrates, then a record of its new location is maintained at the home cluster. As a process migrates from cluster to cluster, its location information at the home cluster is updated each time. This home encoding scheme allows a process to be located anywhere in the system in order constant time, independent of the number of processors. In addition, migrated processes are represented in process descriptors that are local to the new cluster, so that operations on them do not require remote accesses.

Having the size of the clusters smaller than the size of the system has three key advantages:

1. it localizes the kernel data structures that need to be accessed;
2. it reduces the number of process descriptors that must be managed within a cluster, thus reducing the average length of cluster-wide queues; and
3. it limits the amount of searching for ready processes when dispatching.

On the other hand, having a cluster span more than one processor reduces the amount of inter-cluster communication, and makes it easier to balance the load of the processors, leading to better system throughput and reduced application response time.

3.5.2 Memory Management

The memory manager is responsible for the support of virtual resources, including address spaces and their corresponding memory regions. Virtual resources are managed by servers on each cluster, which accept application requests, for example, to allocate and deallocate address spaces and regions. The memory manager is also for supporting physical resources, which include the physical pages of memory and hardware memory components (eg. caches). All memory-related operations at the physical level are on a per page basis, and are primarily demand-driven, which means they are initiated by the hardware as the result of translation or protection faults.

Hierarchical symmetric multiprocessing increases the locality of accesses to the data structures of the memory manager, since each cluster maintains its own set of data structures to manage both the virtual and physical resources of local processes. This improves performance, since the structures that manage private data (such as process stacks) are local to the cluster on which the process executes. When applications share resources across clusters, the data structures that manage these resources can typically be replicated and cached locally, preventing bottlenecks and increasing concurrency, although the replication introduces the issue of consistency.

HURRICANE allows pages to be replicated and uses a simple directory mechanism to maintain the consistency of physical pages across clusters. There is one directory entry for each valid file block currently resident in memory to identify which clusters have copies of the page. The directory is distributed across the processors of the system, allowing concurrent searches and balanced access demand across the system.

Clustering can also provide a framework for enacting paging policies. At this time, the default policy is to share physical pages within a cluster, but to replicate and migrate pages across clusters. Several research groups have shown that page-level replication and migration policies can reduce access latency and contention for some applications [17, 24, 37]. However, the overhead of these policies must be amortized to realize a net gain in performance. On machines where the local-remote access ratio is high, relatively few local accesses are sufficient to recoup the cost of a page migration or replication. However, technology advances have permitted the local-remote access differential to narrow, and have allowed increases in hardware cache size. Both trends mean that more local accesses are required to justify a page movement, which argues for less aggressive placement/replication policies. Moreover, replication lowers the effective utilization of memory by increasing the resident set size of an application, which could lead to increased disk paging when the level of multi-programming is high. The current default policy therefore confines replication and migration operations to pages that are shared across clusters, which reduces overhead but still allows for the reduced latency and increased concurrency of localized accesses to replicated data.

3.5.3 The File System

File system responsibilities can be divided into three levels: name space management; open file state management; and the handling of I/O. In Hurricane, these services are provided on a per cluster basis, and all application requests are directed to local servers.

The single file system name space of HURRICANE is managed by multiple *Name Server* processes. There is one Name Server per cluster in the system, and each server is responsible for searching and consistency of the file names accessed by its local cluster. File names and directories are replicated on demand to those clusters where they are accessed[3]. Consistency of replicated entries is maintained through an updating mechanism (instead of invalidating). This approach localizes name space searches and allows them to proceed in parallel across clusters.

Open files are seldom shared between programs, so open file state is generally maintained on the cluster where the file was opened. In three cases the open state may become

[3] Name space entries accessed only by a single process are therefore confined to a single cluster.

used at other clusters, namely: 1) the process that opened the file passes the file handle as a capability to another program, 2) several processes of a program spanning multiple clusters are accessing the file, 3) the process accessing the file migrates to a new cluster. In these cases the remote cluster replicates the open file state from the home cluster (the id of which is encoded in the file handle).

For those operations that require I/O, we believe clustering can provide a framework for balancing the load across the disks of the system. Although this part of the file system has not yet been implemented, we believe that some files could be replicated across a number of clusters, and other files may be migrated from one cluster to another.

In summary, the file system uses replication at all three levels to increase the number of resources and service points available to handle service requests, and to allow these requests to be serviced concurrently. Since the data structures are replicated local to the cluster, the locality of the applications is preserved.

3.5.4 Scheduling

The primary purpose of the scheduling subsystem is to keep the loads on the processors (and possibly other resources, such as memory) well balanced. As defined in scalability property 3 (Section 3.2), this balance is crucial to scalability, in that it permits the system to function below saturation levels. In a hierarchical symmetric multiprocessing system, the processes of an application are scheduled to run in a single cluster, unless there are performance advantages for a job to span multiple clusters. Hence, for parallel programs with a small number of processes, all of the processes will run on the same cluster. For larger-scale parallel programs that span multiple clusters, the number of clusters spanned is minimized. These policies are motivated by simulation studies [68], which have shown that clustering can noticeably improve overall performance[4].

The scheduling decisions are divided into two levels. Within a cluster, the load between the processors is balanced at a fine granularity through the dispatcher (in the micro-kernel). This fixed scope limits the load on the dispatcher itself and allows local placement decisions to be made concurrently. Cross-cluster scheduling is handled by higher-level scheduling servers, which balance the load by assigning newly created processes to specific clusters, and by migrating existing processes to other clusters. The coarser granularity of these movements permits a low rate of inter-cluster communication between schedulers.

3.6 Super Clusters

A single level of clusters can be expected to effectively support moderately large systems (in the, say, 100–200 processor range). However, for larger systems, additional levels in the hierarchy will probably be necessary. In particular, while each cluster is structured to

[4] A similar structuring mechanism for scheduling has been proposed by Feitelson and Rudolph [28], and by Ahmad and Ghafoor [3].

maximize locality, there is no locality in cross-cluster communication. Examples of this are the home cluster concept for address spaces and the directory for locating pages. The logical next step is to group clusters into super clusters. Processor load balancing is an obvious candidate for this hierarchical clustering. A high-level process manager schedules processes between super clusters, while lower-level managers schedule processes within a super cluster.

The introduction of super clusters should not affect the lower levels of the system. For example, the micro-kernel requires no changes. Super clusters can also be applied to memory management and I/O. For example, in the memory management, the single-level directories could be replaced with a hierarchical one, and multiple copies of the home address descriptor may be necessary, say one per super cluster.

Chapter 4

Requirements

In this chapter, we describe the environment in which the memory manager must execute, and the abstractions the memory manager must provide to the applications. As such, this chapter can be considered as an informal specification of the manager as far as its interfaces are concerned. The choices for both the operating system and the abstractions the memory manager must support are in some sense arbitrary, but represent the state of the art today.

4.1 Application Abstractions

Each HURRICANE application executes within a single address space, regardless of the number of processes it uses. The address space contains all the memory resources that the process of the application can access directly. The address space is defined by a set of non-overlapping *regions*, each of which describes the attributes of a contiguous sequence of virtual *pages*. Besides its virtual extent, each region specifies the attributes of the pages within it. These attributes include initialization characteristics such as initial placement, copy on first write, or zero on first access. Each region also identifies the file region to which it is bound, where a file region is defined as a contiguous sequence of page-sized logical file blocks. The one to one correspondence between a memory region and a file region is called the single-level store, or mapped file, abstraction [39]. The single-level store abstraction is a powerful concept that allows accesses to memory to behave as if they were accesses to the underlying file directly, and is therefore intuitive to apply. Using the single-level store abstraction can also lead to performance improvements, because less data copying is needed relative to the UNIX byte stream abstraction [33].

Figure 4.1 depicts the single-level store abstraction as supported by HURRICANE. In the example, two address spaces have regions bound to files `A` and `B`. The shaded areas of each address space are not bound to any region. This type of address space structuring is called "sparse", because while every address space is capable of providing the addressable limit supported by the hardware, only those virtual addresses that are actively bound are accessible to the application. An address space provides protection in the sense that pro-

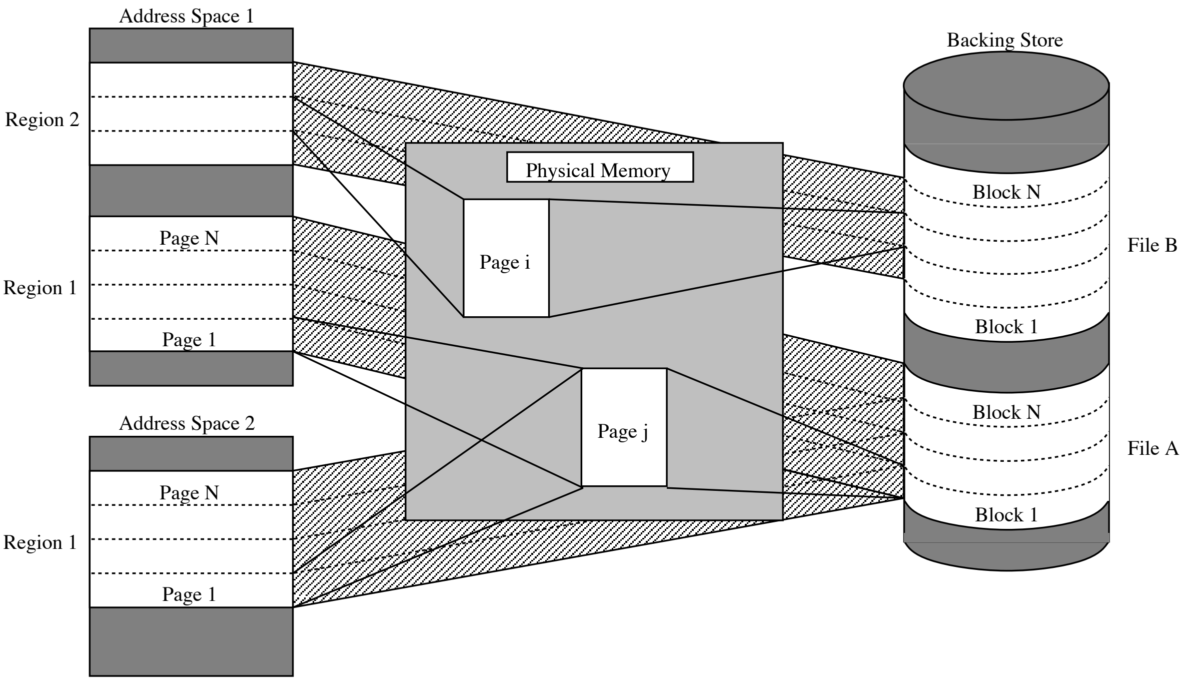

Figure 4.1: Virtual memory abstractions in HURRICANE.

cesses may not directly modify data in any address space but their own. Thus, processes in `Address Space 2` of Figure 4.1 cannot access the memory in `Region 2` of `Address Space 1`. However, because both address spaces have regions bound to `File A`, modifications to this file by either address space are immediately seen in the other.

Figure 4.1 shows that while virtual pages appear to the application as though they are mapped directly to their corresponding file block, they are in fact mapped to physical pages in main memory, which in turn contain the data from their corresponding file blocks on secondary store. In this respect, the main memory of a single-level store system can be considered a cache of secondary store.

The single-level store abstraction can be used to define the interface requirements of the memory manager. Specifically, the memory manager must support application requests to create an address space, to bind or unbind a region, and to manipulate the attributes of an existing region. In the remainder of this section, we briefly review the application interface of the HURRICANE memory manager.

A new address space is created as part of program creation:

```
CreateProgram( filename )
```

where `filename` is the backing file for the program, and is used for page-out of temporary, private regions like the stack and heap. Initially, the address space contains no regions. Regions are created by the call

```
BindRegion( pid, fhandle, freg_start, len, mreg_start, attr )
```

which specifies the virtual extent of the new region as `[mreg_start, mreg_start+len]`.

The file to which the region is bound is specified by a file handle, `fhandle`, and the file region extent is `[freg_start, freg_start+len]`. The address space in which the region is created is specified by the process identifier (`pid`) of any process within the target address space. This explicit address space specification allows a suitably privileged process to bind regions in address spaces other than its own. For example, a Program Manager might bind a stack and executable file for its clients.

The `attr` field of the `BindRegion` call defines the logical and physical attributes of pages within the new region. The physical attributes of the region allow one to specify read-only access, as well as initial placement and coherence policies. Logically, a region may be bound either *shared* or *private*. This classification has no distinction for immutable regions, but changes to a shared region are visible to other programs mapping the same file region, while changes to a private region are seen only by the program modifying the data. Private regions are implemented through a copy-on-write mechanism, described in Section 5.7.

Other calls that pass the application interface include

```
UnbindRegion( pid, address )
ExtendRegion( pid, address, len )
GetRegionAttributes( pid, attr )
SetRegionAttributes( pid, address, attr )
ResetRegion( pid, address )
```

Like `BindRegion`, these calls use `pid` to identify the address space; the region of interest is specified by supplying any address contained by it.

4.2 Hurricane Interprocess Communication

HURRICANE's interprocess communication facility is described here for two reasons. First, HURRICANE message passing is used for communication between loosely coupled servers, and between servers and applications. Second, a generalized facility to transfer arbitrary contiguous virtual segments of data across address space boundaries is available to applications and must be provided by the memory manager.

The basic message passing facility is based on a Send-Receive-Reply (synchronous) message transaction identical to the facility provided by the V system [23]. The sender executes

```
Send( pid, msg )
```

where `pid` is the destination process, and `msg` points to a message of fixed size. The receiver, when executing

```
pid = Receive( msg )
```

blocks until a message is available, at which time the received message is copied to the location specified by `msg` and the process id of the sending process is returned. The sending process is generally referred to as the *client* and the receiving process is generally referred to as the *server*, regardless of what roles these processes play in the system otherwise. The sending process is blocked until the receiver responds with a reply message:

`Reply( pid, msg )`

This causes the message pointed to by `msg` to be copied back to the client, where it overwrites the message that was sent when the transaction was initiated.

The IPC facility is optimized for the synchronous passing of fix-sized messages on the premise that RPC-style communication, where the sender blocks until it receives a reply, will be the dominant form of communication, and is particularly suited for client-server interaction. However, other forms of communication are also supported, including the passing of asynchronous or real-time messages.

A number of primitives provide functionality in addition to the basic message transaction. For example, the function

`Forward( msg, frompid, topid )`

forwards the message pointed to by `msg` (originally received from process `frompid`) to the process specified by `topid`, as though it had been sent by the process `frompid` directly. This function is used to forward a message to a third process, in place of replying to a message. `Forward` does not block.

Processes in different address spaces can share data on a per page basis by binding to a common file. However, for one-time or small transfers, the overhead of setting up and tearing down such a connection will be too expensive. Examples where this may be the case are: passing a file name to the File Server for `open()`; requesting the state of a process from the kernel; or initializing a program's arguments on start-up. These data items are typically smaller than a page and are not normally aligned to any page boundaries, so the page-level protection provided through virtual memory is inconvenient.

The memory manager therefore provides a facility to transfer arbitrary contiguous virtual segments of data across address space boundaries. The application interface to this facility is through the two calls: `CopyTo`, and `CopyFrom`. The transfer is protected in conjunction with the HURRICANE IPC protocol as follows. When process `A` sends a message to a process `B`, the message can specify both the access rights and the location and size of a virtual memory segment in its address space. Consistent with these access rights, process `B` may have data transferred to or from any portion of this segment through the protected copy system calls. The permissions last until `B` replies to the message sent by `A`.

The implementation of Protected Copy is described in Section 5.8.

4.3 Memory Manager Interfaces

The memory manager communicates with applications and the rest of HURRICANE through four cooperating interfaces: the application interface, the kernel interface, the I/O interface, and the hardware. Figure 4.2 shows the memory manager in relation to these entities. Circles represent processes; processes within the same solid box share an address space. The arrows indicate the direction of communication. This section briefly describes each interface, focusing on the requirements implied for the memory manager.

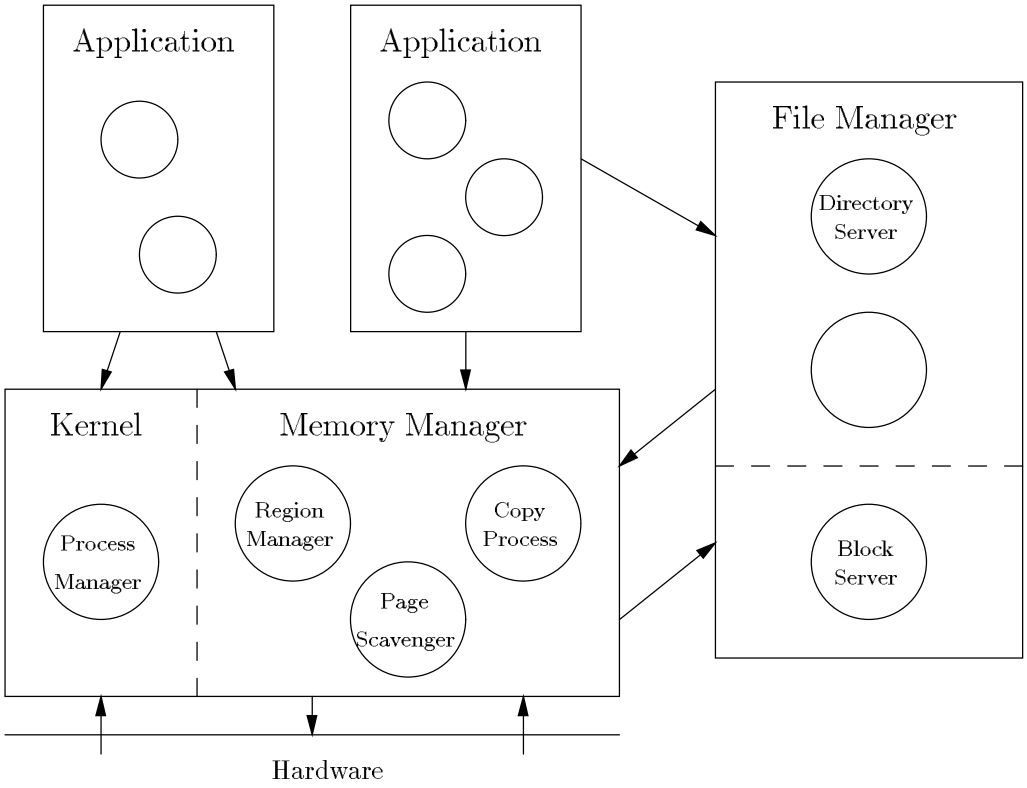

Figure 4.2: Memory Manager interfaces.

The application interface is defined by the single-level store abstraction. Applications create and manipulate regions by directing requests to the *region manager* of Figure 4.2. The *copy process* services application requests to transfer data across address space boundaries. Although not (necessarily) visible to the application programmer, communication with the servers of the memory manager occurs through message passing.

The relationship between the file system and the memory manager is always difficult in a server-based system. Implementing the file system above the virtual memory system leads to an awkward design, because the virtual memory system must use the services of the file system for paging. On the other hand, implementing the virtual memory system above the file system makes the implementation of the virtual memory system more complex and less efficient; for example, it requires the memory manager to verify file access permissions with the file server when the application requests that a file region be bound into its address space. For this reason, we have split the functionality of the file system into two levels, one logically above the memory manager, and one below the memory manager. We are unaware of any similar design, yet in our experience it has worked well.

At the high level, the *directory* server acts in a supervisory capacity, ensuring that processes only bind to files within the permissions granted by the access rights of the file. Hence, a `BindRegion` call is first directed to the directory manager for permission checking; the request is then forwarded to the region manager. At the low level, the memory manager and *Block Server* communicate to transfer physical pages to and from secondary store. The communication interface is synchronous and block-oriented, and is based on the two calls `ReadBlock` and `WriteBlock`. These calls are used to satisfy page-fault and write-back needs, and specify a page-sized file block and physical memory target or source address. The file block is uniquely specified by the tuple <`file_server,token,block`> that is part of all file block identifiers maintained for each physical page. Physical memory addresses are used to specify the source or destination page to allow efficient Direct Memory Access (DMA) by the device servers.

The kernel server and memory manager reside in the same address space, but are kept separate by a procedural interface (the dashed line in Figure 4.2). The kernel server is responsible for servicing requests that change the state of processes. Some examples of the services provided are: `CreateProcess`, which allocates a new process descriptor and readies it for execution; `SetProcessPriority`, which resets the priority of a process, possibly changing its position in the queue of ready processes; and `Delay`, which suspends a process for a specified time interval. Created processes always reside in the same address space as their creator, unless they are created as the result of a `CreateProgram` call. The address space continues to exist until all processes within it have terminated. For this purpose, a process count is kept for each address space on a per cluster basis, which is decremented as each process within it is destroyed. The address space itself is destroyed and its resources released when the count reaches zero.

Finally, the memory manager must interact with the hardware base. The interface between the memory manager and the hardware is bi-directional. The memory manager communicates with the hardware through device-specific instructions or registers, and through the page tables that describe virtual to physical translations and access protections. Device specific instructions include operations that flush a page from the hardware cache, or invalidate a particular TLB entry. The hardware communicates with the memory manager by raising exceptions, such as translation or protection faults. On a fault, the processor saves the state of the faulting process and makes it available to the memory manager. This state information includes, among other things, the process' registers, the virtual address that caused the fault, and the type of access (read or write). The memory manager can then use this state to determine what actions are needed to resolve the fault.

Chapter 5

Per-Cluster Memory Management

In a hierarchical symmetric multiprocessing system, each cluster provides the complete functionality of a small scale symmetric multiprocessor operating system. This chapter describes the data structures that support memory management functions within a Hurricane cluster. The data and control structures used within a cluster are important in attaining good performance for local interactions, and because clusters serve as the base upon which larger systems are built. Although developed independently and concurrently, the structures used within a cluster are similar in many ways to those of other modern memory managers, such as the memory managers of Mach [66] and Chorus [1]. However, our approach differs significantly from that of other systems in a few areas. For example, our copy-on-write and TLB consistency mechanisms are different from those implemented in Mach or Chorus.

The structures used within a cluster are relatively complex, and the interactions between different structures are often subtle. Consequently, we begin with an overview of the primary data structures, and show how they are used to service two key operations: page faults and unmap operations. Subsequent sections examine each data structure in detail.

5.1 Overview

Figure 5.1 shows the primary data structures of the memory manager. The data structures are shared by all processors within the cluster, in order to achieve good performance. The responsibility of virtual resource management is to establish and maintain the logical correlation between virtual pages and file blocks. The virtual resources are maintained on a per address space basis, and include the address space descriptor, the region tree, and the sub-region lists. For each address space, the relation between a virtual page and the file block it is bound to is kept in the region descriptors of the region tree.

The responsibility of physical resource management is to establish and maintain the correlation between file blocks and physical pages. In Figure 5.1, the data structures associated with the management of physical memory are the page cache and page descriptors. There is one page descriptor for each each physical page in the cluster. The page descriptor

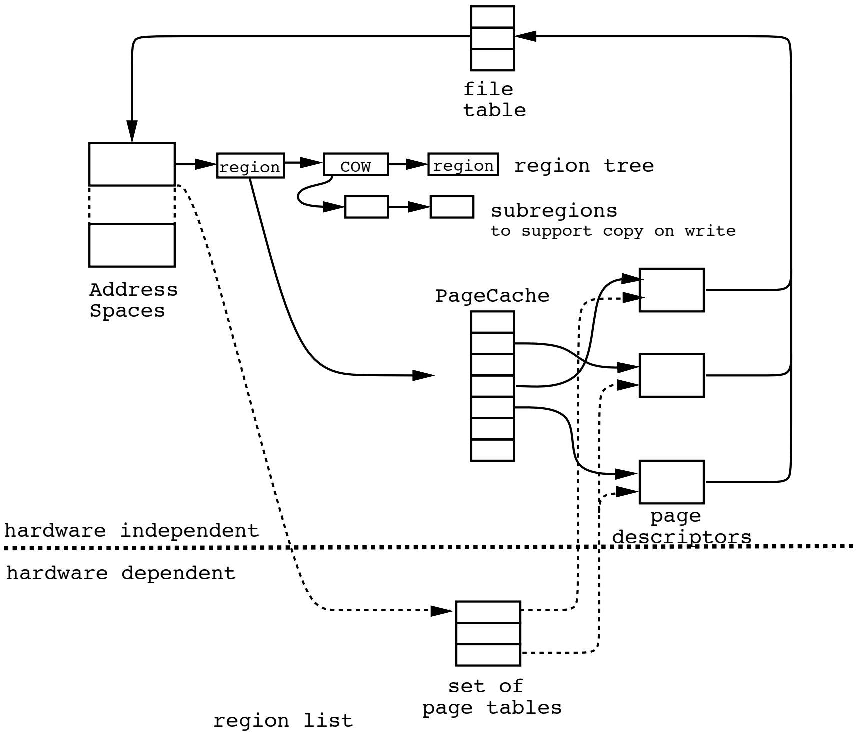

Figure 5.1: Per-cluster data structures.

for a particular physical page identifies the file block cached by the page; the file block is uniquely identified by the tuple <`server,token,block`>.

By separating virtual and physical resource management at the level of file blocks, several advantages are realized. First, the level of indirection introduced permits many-to-one and one-to-many relationships between virtual and physical pages. As an example of a many-to-one relationship, several programs can share a single physical page by binding separate virtual pages to the same file block. This many-to-one mapping can be used to allow sharing across programs, or to improve memory utilization. A one-to-many relationship may be used when a single program spans several clusters. In this case, a single file block can be cached by several different physical pages to reduce contention and improve locality. The decoupling of virtual and physical responsibilities also has the advantage of localizing the effect of changes in each layer, which improves portability and modularity. For example, in moving from one architecture to another, the policies at the physical layer can be tuned for better performance without affecting virtual resource management.

Operations on virtual resources are usually initiated in response to application requests. For example, requests to bind a region or unmap the pages of a region are sent, via message-passing, to the region manager process of the local cluster. Operations on physical resources are usually initiated on demand in response to hardware faults. The three demand-driven mechanisms in Hurricane are paging, cache coherence, and copy-on-write. All three mechanisms are able to defer their actions by relying on the hardware translation and protection support.

The operations performed by the memory manager can also be classified as either maintenance or translation operations. Maintenance operations at one level do not affect the other level because of the decoupling between virtual and physical resources. Maintenance operations on virtual resources include allocation and deallocation of address spaces and regions, as well as the checking and setting of access permissions. Maintenance operations on physical resources include page allocation, page placement and page replacement. To *map* a page[1] is to establish a virtual to physical translation for the page. In practice, this involves setting an entry in the page tables. A page *unmap* removes the virtual to physical translation. In practice, unmap operations are more complex than just removing a page table entry, because cached context may also have to be invalidated from cache and TLB entries. Because translation operations are the most common operations the system performs, the data structuring should be such that the operations can be executed quickly.

Pages are mapped on demand through page-faults. On every memory access, the hardware automatically checks first that the virtual address can be translated to a physical address, and that the type of access does not violate the permissions for that address. If either of these checks fails, an exception is raised on the processor and the memory manager takes over. A translation can fail either because the virtual address is invalid (not within any region of the process' address space), or because the mapping between the virtual and physical pages has never been established. The access check normally fails as processes attempt to write to pages marked read-only. Pages are write protected because the region is immutable, the page is marked for deferred copy, or as part of the memory manager's coherence policy. Illegal accesses are passed to the Hurricane Exception Manager [16]; the algorithm to handle page faults is discussed briefly here.

When a translation fault is recognized, the processor saves the state of the faulting process and passes it on to the memory manager. This state information includes, among other things, the process' registers, the virtual address that caused the fault, and the type of access (read or write). With this information, the memory manager searches the region tree to determine the region containing the address, the access permissions, and the file block associated with the virtual address. Processing then moves to the physical level, where the memory manager searches the page cache to determine if the file block is already memory resident. If the file block is already memory resident, the search identifies the page descriptor of the physical page that is caching the file block. If the file block is not yet memory resident, a new page is allocated to hold the file block data when it arrives from secondary store. In addition, a page descriptor for the new page is added to the page cache. Of course, fault handling in this case must be suspended until the file block transfer is complete. The virtual to physical translation is completed by updating the page tables of the address space that caused the fault, and completing the instruction that caused the fault.

Unmap operations also begin with a virtual address and address space context. Strictly

[1] In contrast, we use the term *bind* to refer to the relation between a virtual page and its corresponding file block. Thus, an application "binds to files" and "maps pages".

speaking, the translation can be removed by manipulating the page tables alone, but the physical page descriptor is also accessed for clean-up purposes, such as freeing the page for possible reallocation. The file table of Figure 5.1 identifies all the address spaces with regions that bind a particular file. This table is used to determine the virtual pages that reference a particular physical page, so that the page table entries can be reset.

The data structures of the memory manager must support the basic page fault and unmap operations efficiently. The most common operation required of the data structures is search. For example, the regions of an address space must be searched to find the file block bound to a given virtual address, and the physical page descriptors must be searched to find which page, if any, caches a particular file block. Besides search, the data structures must support add and delete functions for use by the maintenance class of operations. Since the data structures may be accessed concurrently by several processors needing service on the same or different data, synchronization capabilities must be an integral part of each structure. Synchronization is required at two levels: first to maintain the integrity of the data structures themselves; but also to keep application data consistent across memory and with respect to secondary store. For example, accesses to a file block must be suspended while it is in transit from secondary store, and modifications must be delayed if the block is currently being replicated.

Each data structure could be implemented in any number of different ways, each with its own space/time strengths and weaknesses. The remainder of the chapter describes the choices made in Hurricane, and shows how they meet the basic design guidelines of Chapter 3.

5.2 Virtual Resource Management

Figure 5.2 shows the state kept for each address space: an address space descriptor, or `ASID`; a tree of bound regions; and a set of hardware dependent page tables. All processes within a program share a single ASID record. This makes the ASID a convenient place to keep program-wide information, such as the default backing file and memory usage statistics.

The `regions` field of the address space descriptor points to a balanced binary tree of `Region` records of the form shown in Figure 5.3. The `start` and `end` fields of a `Region` record denote the virtual memory limits of the region; the `file` and `cowfile` structures together define the file to which the region is bound. File regions are identified uniquely by the tuple <`server,token`> and the starting file block of the region. The file region specified by `file` is always the backing store for the region, however, copy-on-write regions may temporarily override this binding to source their data from the `cowfile` file region. The Hurricane copy-on-write mechanism is discussed in more detail in Section 5.7.

The region attributes and the attributes of pages within the region are defined by the `regatt` and `pageatt` fields. `Regatt` maintains the machine independent attributes of a region, such as shared or demand-zeroed; `pageatt` stores the machine dependent page attributes, such as write-back or uncached, and is copied into the page tables as each page

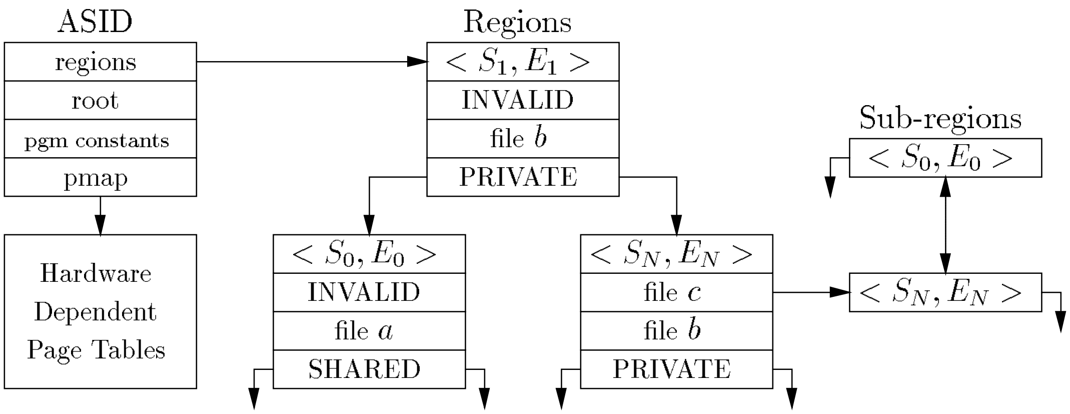

Figure 5.2: The structures associated with each address space.

```
typedef struct subregion {
    struct subregion *left, *right ;
    AddrType start, end ;          // Virtual addresses in this region.
} RegionElement ;

typedef struct {
    ServerType server ;            // File id.
    TokenType token ;
    BlockType start ;              // Starting file block.
} FileRegion ;

typedef struct {
    RegionElement region ;
    FileRegion file ;              // Backing file for this region.
    FileRegion cowfile ;           // Copy-on-Write file.
    RegionElement *subregion ;     // Deferred copy sub-regions.
    Core *preferred ;              // The CPU from which to get memory.
    short regatt ;                 // Machine-independent attributes.
    short pageatt ;                // Machine-dependent attributes.
} Region ;
```

Figure 5.3: The data structures used to describe a virtual memory region.

is faulted in.

The regions are searched on each page fault, so it is imperative that the search overhead be kept low. For this reason, the regions are structured as a balanced binary tree [57]. The tree structure yields $O(\log_2(N))$ worst case search times, and permits searches by several processors to proceed concurrently under the protection of a single multiple-readers/single-writer (MRSW) lock[2]. The main drawback of a balanced tree is the high cost of rebalancing the tree after an insert or delete operation. However, these operations were observed to be relatively infrequent, particularly in comparison with the number of search operations, so the overhead seems justified. The rebalancing after a delete is especially difficult, so a "lazy delete" algorithm was adopted, where a region is deleted by marking the record invalid, and leaving it in the tree. Subsequent searches must step over the invalid record, but their occurrence in practice is relatively rare, and they can also be re-assigned to a new file region if an insert is performed later.

The machine dependent page tables are the third major data structure kept with each address space. These tables record the virtual to physical address translations and access permissions as required by the hardware. Page tables are created on demand as the processes establish their working sets. To support software coherence, each processor has its own independent set of page tables. The trade-offs involved in this decision are discussed in more detail in Section 5.5.

5.3 Physical Resource Management

The structures used to manage the physical memory of each cluster include page descriptors to maintain the state of each physical page, a `Core` structure to manage all the pages on a single memory module, and a `PageCache` hash table to locate physical pages using the file block identifier as a search key. Figure 5.5 shows the organization of these structures.

Each physical memory page on the machine has a page descriptor associated with it, defined as shown in Figure 5.4. The primary purpose of the page descriptor is to identify the file block cached by the physical page at address `addr`. The file block is identified by the tuple <`server,token,block`>. In addition, the page descriptor field `status` is used to maintain the state of the descriptor itself and of the data associated with it. The state has two components: the coherence state consists of several bits used to describe the cacheability of the page; and general attribute information such as `IN_USE` (the page descriptor is not available for re-allocation), `PINNED` (the page must remain resident in memory), or `DOING_IO` (the page is in transit to or from backing store).

[2] A previous design used a doubly-linked list sorted in order of increasing virtual address. The search time of the list was reduced by setting the root of the list (the `regions` pointer) to the target region after each search. Assuming there is some locality of access, there is a good chance that on the next page fault the root already points to the correct region. There are two drawbacks to the list based approach: 1) resetting the root pointer after each search limits concurrency because the root must be changed within a critical section; 2) the assumption of locality of access is not appropriate when several processes, each with a potentially different working set, all want access to the same region list.

```
typedef struct page
  {
    ServerType server ;          // Unique file id.
    TokenType token;
    BlockType block ;            // The file block stored in this page.
    AddrType addr ;              // Physical address of the page.
    unsigned short status ;      // IN_USE, PINNED, etc.
    unsigned short members ;     // Processors caching this page.
    struct page *hnext, *hprev ; // Hash list pointers.
    struct page *fnext, *fprev ; // Free list pointers.
  } Page ;
```

Figure 5.4: The page descriptor record, which is used to describe the state of each physical page in the system.

The page descriptors for all the physical pages of a single memory module are kept in an array called the `pdarray`, which is part of the `Core` structure of Figure 5.5. There is one `Core` structure for each memory module in the cluster, and besides the `pdarray` and some addressing information, each `Core` structure independently maintains a `freelist` of pages that are available for allocation from that memory module. A page is available for allocation if it is not `VALID`, or if it is valid but not `IN_USE`. A page is valid if the file block it caches represents the correct value on secondary store; the page is in use if it is currently mapped into the address space of a program in the cluster. It is important that the free lists of different `Core`s be separate, so that it is possible to quickly allocate a page from a particular memory module. This allows the implementation of various page placement policies.

The three page placement policies currently supported are *first-hit*, *round-robin*, and *fixed*, and are specified on a per region basis:

First-hit: The pages of a first-hit region are allocated from the `Core` of the processor that first accesses them[3]. This policy can improve locality for data accessed by a single process, because the pages are allocated from the processor on which it is executing. Stack regions and regions used for private data are examples that can benefit from the first-hit policy.

Round-Robin: The round-robin policy cycles the pages of a region across the memory modules of a cluster, which can help balance the memory access demand for data that is shared by several processes. For example, code pages and regions that contain common arrays can benefit from a round-robin policy. The `preferred` field of the region descriptor (Figure 5.3) identifies the memory module from which to al-

[3] We assume here that a memory module is associated with each processor, as is the case on Hector.

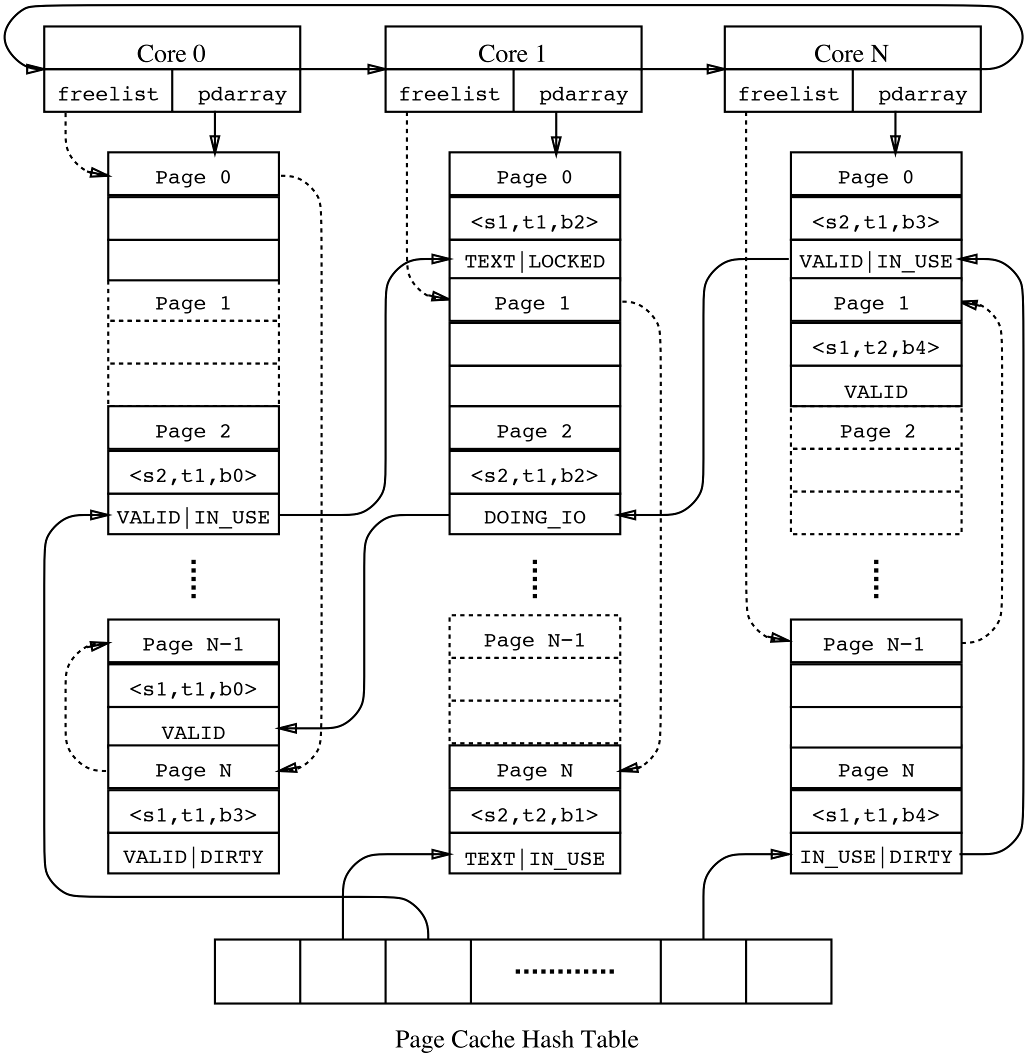

Figure 5.5: The structures used to manage physical memory. The page descriptors of each memory module are grouped together into an array called `pdarray`, which is part of a `Core` structure. There is one `Core` for each memory module in the cluster, and each `Core` has its own LRU `freelist`. There is one `PageCache` hash table in the cluster, which is searched by file block identifier. The chains of the `PageCache` therefore run freely across all the memory modules in the cluster.

locate the next page[4], so that the pages of different round-robin regions are cycled independently.

Fixed: Regions bound with the fixed policy keep a core identifier from which all pages in the region are allocated. This policy is appropriate for applications that know their access patterns in advance, and for system servers that have special placement requirements.

The physical pages of main memory are treated as a cache of file blocks. The relationship between the file block and its location in memory is kept in the page descriptor. The `PageCache` hash table permits a page to be located by searching for its corresponding page descriptor, using the file block identifier as a search key. The basic structure and use of the page cache is identical to the block cache described by Bach [6]: the `PageCache` hash table is keyed by the file block identifier and uses doubly-linked overflow lists to resolve hash conflicts. To support a high degree of concurrency, each bin of the `PageCache` hash table has its own MRSW lock to allow searches to proceed in parallel. Also, each page descriptor is locked separately to allow operations on different pages to proceed concurrently.

All pages in the page cache, except those whose data is in transit from secondary store, are `VALID`, which means that their contents represent the most up-to- date value of the data associated with the file block. Further, all blocks in the page cache are either `IN_USE`, which means they are currently mapped into the address space of at least one program, or they are on a free list, which means they are available for re-allocation to a different file block. By leaving free pages in the `PageCache`, processes are given a "second chance" to reuse the file block before it is reassigned, which saves having to refetch the data from secondary store.

The free lists are maintained in approximate LRU order to increase the probability that frequently used pages will remain in the cache. Freed pages are appended to the end of the free list, while new pages are allocated from its head, so that each page remains in the cache for the time it takes to move to the head of the list before its contents are reassigned. To further enhance this caching, invalid or temporary pages are inserted at the head of the free list so they can be reassigned before the valid pages further down the list. Pages are invalidated as a result of a file delete or truncate; temporary pages include segment tables and stack pages of terminated programs.

The page cache is used as follows. When a page fault must be satisfied from a file, the cache is searched to see if the file block is already in memory. If the block is found (a cache hit), the page containing the block is mapped to the address space of the faulting process, and no I/O is needed. On a cache miss, a new page is first obtained from the free list and inserted in the cache. The placement policy of the region containing the virtual page determines the core from which the physical page is allocated, but otherwise the pages in the cache can be shared by all processors in the cluster. The file block is then fetched

[4] The field is called `preferred` because the memory module identified by this field may not have any pages available for allocation.

from secondary store into the new page. While the data is in transit, the page is marked `DOING_IO` so that other processes faulting on the same page know not to reissue the fetch.

The transfer from secondary store is initiated with a `ReadBlock` call. Because `ReadBlock` is a message send to the Block Server, and because the HURRICANE IPC protocol is process based, a process is needed to send the message. The memory manager cannot issue the send, since it is simply exception handling code and has no associated process descriptor. Instead, the descriptor of the faulting process is used, and is set up to appear as though the process made the `ReadBlock` call directly. This approach has several advantages: it ensures that the memory manager is never blocked waiting for I/O; it allows multiple outstanding page faults (by different processes); and it automatically blocks the faulting process until the server indicates that the transfer is complete.

5.4 The File Table

To this point, the data structures have been described in the context of fault handling: given a virtual address and address space, the region tree identifies the logical file block, which is used to locate the physical page. The reverse information is also needed: given a physical page, the system must determine the virtual mappings to it. This section describes the data structures and control flow associated with the most common operations that require this information: *unmap*, *uncache*, and *invalidate*.

The unmap operation removes the virtual to physical mapping for a given page, so that it can be freed for re-use, or so that its coherence state can be reset. The invalidate operation is similar to unmap, except that the page is removed from the page cache. Unmap and invalidate operations are typically not demand-driven, but are initiated by the region manager in response to program termination, at the request of a user application, or in response to a page scavenger process that frees pages not recently accessed. In contrast, the uncache operation is demand-driven, and is invoked when the HURRICANE default coherence policy determines that a page is write-shared by two different processors (see Section 5.5).

The operations described above begin with a target file block, which is obtained from a region descriptor for unmap, and from the physical page descriptor for uncache. The next step is to update the page table entries that reference the page corresponding to the target file block. For example, if a page is invalidated, then all virtual to physical translations to the page must be removed to ensure correct behavior. Updating the page table entries is complicated by the fact that a physical page could by mapped into the address space of many different programs.

There are several different ways to determine all the virtual to physical mappings that exist for a particular physical page. In general, different approaches can trade search time for space cost. For example:

- Use only the address space and region descriptors that already exist. This approach has no space cost, but requires an exhaustive search of all address spaces and regions

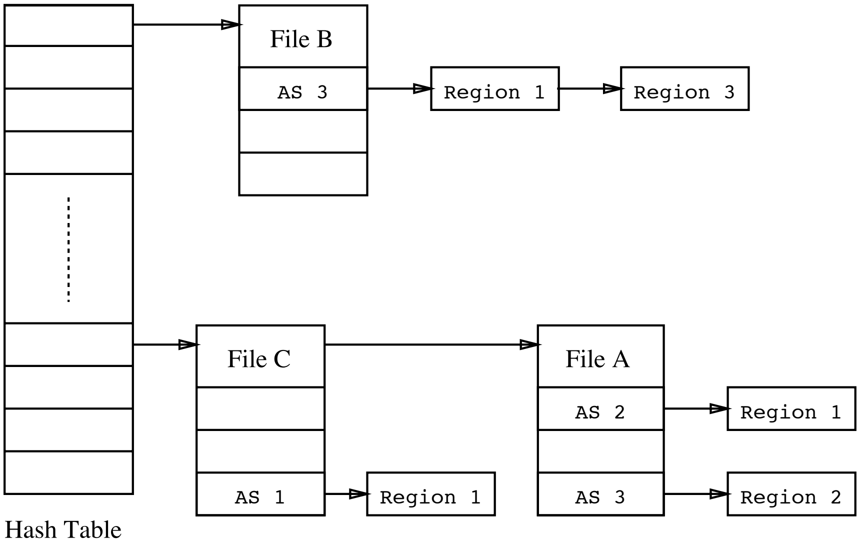

Figure 5.6: The `FileTable`.

to determine the virtual pages that reference the target physical page.

- Record all virtual mappings on a per file block basis. This approach minimizes search time, but has a prohibitive space cost.

- Keep some information, such as a list of address spaces and regions within them that access a particular file. This approach is used in Hurricane. Keeping information on a whole file basis requires less searching than an exhaustive search, and does not have excessive space cost.

Figure 5.6 shows the structure of the `FileTable` used to keep the reverse mapping information. There is one entry in this table for every file that has an *active binding*, which means that some portion of the file is bound to some memory region of an address space. For example, Figure 5.6 shows address spaces with regions bound to some segment of files `A`, `B`, and `C`. The entries of the table are located through a hash table keyed by the <`server,token`> fields of the target file block. Each table entry lists all the address spaces within the cluster that specify a binding to some segment of the file, and each address space in this list has an associated list that identifies the regions within the address space that bind the file. In the figure, file `A` is referenced by region 1 of address space 2, and by region 2 of address space 3.

Given a particular file block, the table is accessed as follows. First the hash key is applied to locate the table entry for the file. Through the address space list and associated region lists, each entry maintains all the regions in the cluster that specify a binding to this file. The extent of the file segment bound to each region is then checked, and if the particular file block lies within one of these extents, the corresponding virtual address is obtained and used to update the page table entries for the address space.

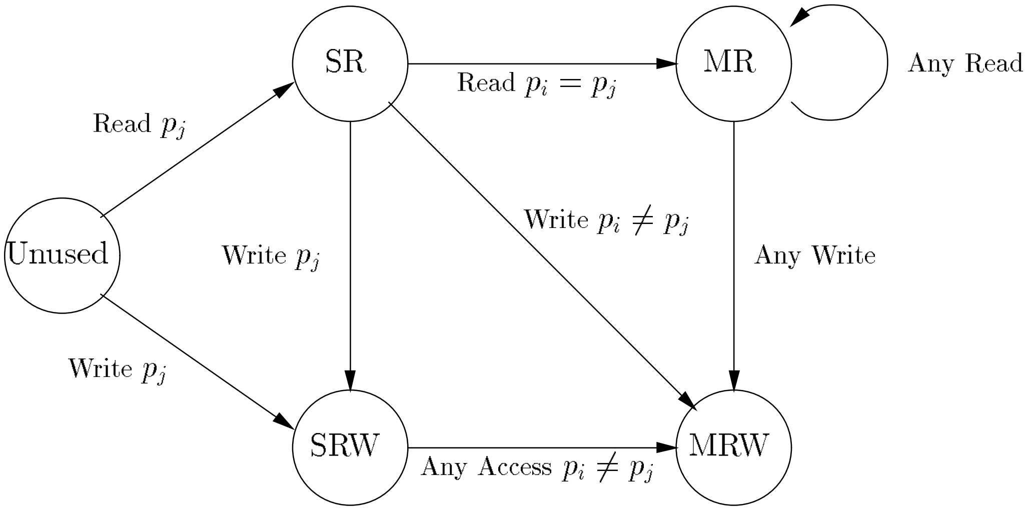

Figure 5.7: The state transition graph for a simple coherence policy.

5.5 Cache Coherence

For those systems that do not support cache coherence in hardware, Hurricane is capable of supporting cache coherence in software. Cache coherence is maintained on a per cluster basis using a full-map directory protocol; the protocol is extended hierarchically to maintain both main memory and cache coherence across clusters in Section 6.4. This section describes the protocol and implementation for each cluster, and examines interesting trade-offs through an example coherence policy.

Recall from Section 2.3 that each "coherence unit" in a full-map directory coherence protocol has a directory entry associated with it, which records the state and processors caching the data. In this case, the coherence unit is a physical page, and the page descriptor serves as the directory entry. The `status` field of the page descriptor records the coherence state, and the `members` field contains a bit for each processor in the cluster (see Figure 5.4). For a given page, the *ith* bit of the `members` field is set if processor p_i might be accessing the contents of the page. While different coherence policies can be implemented from this basic mechanism, this discussion will assume the simple policy represented by the state transition graph of Figure 5.7.

A page is initially *Unused*, which means that it is not mapped into the address space of any process. On first access, the page is mapped into the address space of the faulting process and the coherence state of the page moves to either *Single Reader* (SR) or *Single Reader/Writer* (SRW), depending on whether the access was a read or a write, respectively. The *jth* bit of the `members` field is then set because processor p_j is now referencing the page data. The page can be safely cached in the SR and SRW states because it is referenced by a single processor. However, an SR page must be write-protected to allow a transition to the SRW state if necessary.

Once a page is initialized, other processes executing on the same processor (p_j in Figure 5.7) may access the page without changing its state. For processes within the same address space, this sharing is automatic because they all use the same page tables on a per CPU basis. Processes in other programs may also share access to a single physical page by binding a region to the file block cached by the page. However, while processes in two different address spaces may share a common page, they do not share the same set of page tables. This means that processes in both address spaces must incur a translation fault to map the physical page into their separate address spaces.

Consider the case where another processor, p_i in the figure, references the page. If the access is a read and the state of the page is SR, it remains cacheable and write-protected, and the `members` field is updated to reflect the access by the new processor. This is called the *Multiple Readers* (MR) state, and any number of processors may cache the page as long as no write is attempted.

The final state is called the *Multiple Readers/Writers* (MRW) state, and is entered if i) any processor attempts a write to an MR page, or ii) some processor $p_i \neq p_j$ makes a write access to an SR page, or iii) some processor $p_i \neq p_j$ makes any access to an SRW page with member set p_j.

On a transition to the MRW state, the page data must be invalidated from the caches of all processors in the current member set (and flushed, if the page was SRW), and the mapping protections must be reset to specify uncached access. Uncached access ensures consistency by forcing each processor to go directly to main memory on each access to the page. Once a page has become uncached, it remains in the MRW state until it is no longer referenced or until the page is explicitly invalidated or unmapped. This simple algorithm is sufficient to ensure consistent access to write-shared data while retaining the benefits of caching for non-shared or read-only data.

The basic consistency protocol presents many interesting policy and implementation issues. For example, one disadvantage of the coherence policy of Figure 5.7 is that the decision to uncache a page is never re-evaluated. This can penalize those applications that use one process to initialize a data structure that is then accessed read-only by several other processes. This access pattern will result in the page being uncached, because the initialization places the page in the SRW state, and subsequent accesses by any other processor will move the page to the MRW state. To overcome this problem, Hurricane provides a `ResetRegion` operation, which can be called by an application to reset the coherence state of one or more pages. Besides explicitly resetting the page state, it could be possible to re-evaluate the decision to uncache a page by periodically invalidating it; this returns the page to the *Unused* state of Figure 5.7, giving it a chance to enter a cacheable state on subsequent accesses. Several re-evaluation policies have been proposed by investigators at Duke [36] and Rochester [17].

Another problem is "false sharing", which occurs when data used independently by two different processors is co-located on the same page. Since the coherence checking is restricted to page size granularity, these pages appear to be write-shared, and thus are marked MRW. `ResetRegion` is less effective in addressing this problem, because the page is actively shared. User level coherence and weaker consistency models hold promise for reducing the effects of false sharing [55, 20], but the techniques only try to reduce the effects of false sharing, Often, sources of false sharing can be removed by managing the address space so that pages are allocated to data with similar access patterns. Towards this end, Hurricane provides an "arena" allocator package that allows applications to allocate data structures from separate pools, or arenas, of memory. The pools are typically designated on the basis of expected access patterns. For example, each process might have its own arena for private data, while global read-only data and locks are placed in two other separate arenas.

The directory protocol also presents several interesting implementation issues. In particular, there are several trade-offs in the page table management strategy. Because the processes of a parallel program share a common address space, one set of hardware dependent page tables should be sufficient to provide the necessary mapping information for the entire program. However, if cache coherence is to be supported in software, then a separate set of page tables must be kept for each processor. This is because the hardware support for virtual memory typically provides only two types of faults: translation faults, which are used to move from Unused to the SR/SRW state, or from the SR to the MR state; and protection faults, which are used to detect transitions to SRW or MRW states. Further, the demand driven nature of the memory manager only permits state changes in response to a fault. Consequently, to detect the state changes and maintain the member set correctly, each processor that executes a process on behalf of a parallel program must have its own set of hardware dependent page tables.

It is possible to support cache coherence with a single set of shared page tables, but this approach has several disadvantages. This is because once any process has accessed the page to establish a valid virtual to physical translation, any other process, on any other processor, can access that page without generating a fault, since the page tables are shared. Moreover, with a single page table, it would not be possible to differentiate all five states of Figure 5.7. Instead, the SR and MR states would effectively be merged into one *read-shared* state, and the SRW and MRW states would become a single *write-shared* state. The latter differentiation is particularly important, because a page can be cached in the SRW state but not in the MRW state.

Hence, with a single set of page tables, only one process has to fault to map the page for all the processes sharing the address space at the cost of a loss of member information, which leads to more pessimistic decisions about which pages could safely be cached. In addition, the cost to move from the read-shared to the write-shared state is more expensive, in this case, because the lack of a member set means that all processors that *might* be caching the page must invalidate their caches.

Keeping separate page tables per processor therefore allows precise book-keeping, but also increases the memory requirements of parallel programs because of the multiple page tables, and increases the cost of memory management functions that must now deal with multiple translation entries instead of one. In addition, the number of page faults is increased.

Lastly, the scalability of the two approaches must be considered. Clearly, the page tables for a single program cannot be shared across the entire system, because the demand on this single resource would increase proportional to p, the number of processors in the system. However, the tables could be shared within clusters, provided they are kept consistent across clusters.

In summary, the trade-offs between using one shared set of page tables or one set per processor are interesting and complex. A single shared set of page tables requires less space and results in fewer page faults, but the lack of directory information makes the cache coherence and TLB consistency protocols (Section 5.6) more conservative and more costly. By using a page table per processor, it is possible to differentiate between the SRW and MRW states, and to restrict remote interrupts for page table manipulations to exactly those processors that are mapping the page. We feel the application performance that can be gained by exploiting these features justify the space and complexity overheads of having a page table per processor.

5.6 TLB Consistency

Since Translation Look-Aside Buffers are effectively caches of page table entries, and since each processor typically has one or more TLBs, steps must be taken to keep the TLBs consistent with each other, and to keep the TLB entries consistent with the page table entries in memory. Inconsistent TLBs will allow invalid access as a result of a change to a page table entry that makes the page protection more restrictive (for example, when the protection of a page is changed from read-write to read-only). Invalid access cannot occur if the TLB becomes inconsistent due to a change in the page table entry that makes the access protection to a page less restrictive. At worst, a process that accesses a page with a stale TLB entry will fault, and the memory manager will detect and correct the inconsistency in the course of handling the fault. As a result, immediate action to re-establish TLB consistency is only required when the page protection is made less restrictive.

Several groups have studied TLB consistency and have proposed solutions using a single set of shared page tables [14, 54, 64, 7]. The general approach is typically called "TLB shoot-down" and uses remote interrupts to halt all the processors so that they can invalidate the appropriate TLB entry. The basic TLB shoot-down algorithm is overly excessive in that all processors must be interrupted, and is therefore not scalable. All processors must be interrupted because no information is kept about which processors are sharing the single page table entry.

On Hurricane, the directory information and multiple sets of page tables greatly simplify the solution to the TLB consistency problem. First, a lock is associated with each address space to properly serialize changes to the page table entries. When a page mapping is changed or restricted, only the processors in the member set of the page descriptor need to be interrupted with enough information to update their local TLB and page table entries. The TLBs are invalidated concurrently by using a two-phase protocol. In the first phase, the master initiates the cache operation on all the processors in the page descriptor member set through interrupts, then performs the same operation on its own TLB if necessary. In the second phase, the master waits for the other processors to set a flag signaling their completion.

5.7 Copy-on-Write

The single-level store abstraction of Hurricane requires that each memory region be bound to a corresponding file region. However, certain run-time semantics and optimizations require an extension to this basic abstraction. For example, the initialized data segment of an executable file may be modified by the program but should not change on secondary store. Consequently, each program that binds to the initialized data segment should get its own private copy of the file blocks in the segment. In this way, the copied segment can be modified as needed without changing the original image on secondary store. Also, if several programs are running the same executable file simultaneously, each program receives its own copy of the initialized data segment, so that the modifications made by one program are not visible to the other program instances.

To reduce the amount of copying, Hurricane uses an optimization called copy-on-write [62]. The key to the copy-on-write optimization is to realize that the file blocks of the initialized data segment can be shared by the separate instances of the program as long as they do not try to modify the data. Modifications to the file blocks can be prevented by mapping the corresponding physical pages with read-only permissions. If one of the programs attempts to modify the contents of the page, a protection fault is generated, and the memory manager can make a private copy of the file block for use by the faulting program. The advantage of copy-on-write is that file blocks that are never modified are not copied, which reduces overhead. The disadvantage of copy-on-write is that if the page must be copied, the program must suffer the cost of two page faults: one page fault is needed to map the page read-only, the second (write) fault triggers the copy. If copy-on-write were not used, only one page fault would be needed to make a copy of the page and map the copy into the address space of the faulting program.

Recall from Section 5.2 that each region descriptor contains two `FileRegion` structures that identify the files to which the region is bound. If the region attributes specify that the data is globally shared or immutable, then only the `file` entry is used. On the other hand, if the region attributes specify that the program wants its own private copy of the data, then `file` is set to a temporary file used for page-out, and `cowfile` identifies the file used to source the data. In this case, a sub-region list is created to manage the deferred copy state.

Figure 5.8 shows how the sub-region list is used to shadow the blocks of the source file that have not yet been modified. The figure shows a region descriptor with backing file `A`, and copy-on-write file `B`. Initially, the `subregion` field of the region descriptor points to a single `RegionElement` with the same virtual address range as the parent. Figure 5.8(a) shows this initial state: the region is bound to blocks 1 to 3 of source file `B`, which is recorded in the single sub-region element.

Read faults to the region proceed normally, except that the file block is located through

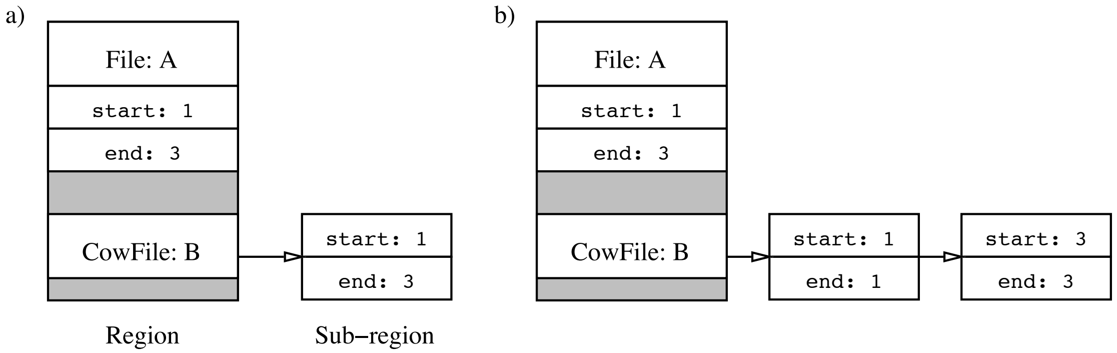

Figure 5.8: a) the initial state of the sub-region list when blocks 1 to 3 of file B are bound for copy-on-write. b) the sub-region list after block 2 of file B has been copied.

the cowfile and its corresponding sub-region list. These pages are mapped read-only to prevent modifications to the source file. On a write access, the page is removed from the sub-region list, splitting single RegionElements into two if necessary. The contents of the source page are copied into a private page mapped in from the temporary file with full access permissions. This scenario is depicted in Figure 5.8(b): block 2 of file B has been copied to its corresponding position in the backing store file, A, so the sub-region list has been split into two single block elements. Modified pages must be removed from the sub-region list because they may be targeted for page-out.

The region tree is then searched as follows. The appropriate region descriptor is located as before, based on the virtual address. If the cowfile descriptor is valid, then the sub-region is searched next. If the file block was never copied, it will be found in the sub-region list, and is still sourced from cowfile. If the page has already been copied, it will not be found in the sub-region list, so the file block is taken from the file descriptor of the region.

Unlike the tree of regions, sub-regions are stored as a doubly-linked list. This data structure was chosen because of the inherently different nature of copy-on-write regions. Although in principle any file may be bound copy-on-write, the overwhelmingly common use is for the initialized data section of executable programs. From the workload we have observed on Hurricane, these regions are relatively small (a few pages on average), and almost all the pages are eventually modified. These findings indicate that the sub-region lists are typically short and short-lived. As a result, the primary operation on sub-regions is to remove a page as the result of a write, which may split a single sub-region into two. A balanced tree is inappropriate if deletes are common, since rebalancing the tree is costly.

The idea of copy-on-write is not new, but our implementation of it is very different from that of Mach [53], Chorus [1], and Sprite [47]. These systems allow programs to inherit regions from their creator on fork() operations. To preserve the semantics of fork() yet attain good performance, the regions are inherited copy-on-write so that if either the parent or the child modify a page, a private copy is derived. If the child then creates a new child, this new child inherits the regions of its parent, which may in turn be copy-on-write regions inherited from some ancestor. Some of the pages in these inherited regions may already be private to the first child, while others may still be shared with the original parent. As the family tree grows, long, complex chains of "shadow" dependencies can arise to keep track of which pages are shared, and by whom.

In contrast, HURRICANE programs do not inherit regions; sharing is accomplished by binding a new region to a common file. This imposes a two-level "master-slave" dependency: the pages of the original file, the master, are never modified; the copy-on-write slaves shadow the master and receive private copies when they attempt to modify the original. Note that the master is not modified even if it is shadowed by only one slave, although the copy could be avoided by simply renaming the modified master block to the corresponding file block of the slave, because this would remove the master from the page cache; if it is then needed again (because, for example, the program is rerun) it must be fetched from secondary store, at a cost that dwarfs the savings gained by avoiding the copy in the first place.

We believe that the two-level approach to copy-on-write is better suited to scalable systems. Since the master is never modified, it may be safely replicated across clusters, which allows faster access and lowers the cost of making a private copy. Although the sub-regions are not replicated like regular regions (see below), this only affects programs that span multiple clusters, and intuitively seems to have lower complexity than trying to maintain a tree of shadow dependencies across clusters.

5.8 Protected Copy

Processes in different address spaces can share data on a per page basis by binding to a common file. However, for one-time or small transfers, the overhead of setting up and tearing down such a connection may not be justified. Examples where this may be the case are: passing a file name to the File Server for `open()`; requesting the state of a process from the kernel; or initializing a program's arguments on start-up. These data items are typically smaller than a page and are not normally aligned to any page boundaries, so the page-level protection provided through virtual memory is inconvenient. The remainder of this section describes HURRICANE's protected copy facility.[5]

The application interface to this facility is through the four calls: `CopyTo`, `CopyFrom`, `GetProcessState` and `SetProcessState`. The first two calls allow the transfer of arbitrary contiguous virtual segments of data across address space boundaries; the latter calls transfer a process specific state structure between the kernel and an application process. The transfer is protected in conjunction with the HURRICANE IPC protocol as follows. A process (`A` in Figure 5.9) grants access to its address space by sending process `B` a message that specifies both the access rights and the location and size of the virtual memory segment. For illustration purposes, process `A` of Figure 5.9 has granted read privileges to the segment

[5] The contents of this section and the next are not central to the thesis — and may be skipped by the casual reader — they are presented for completeness.

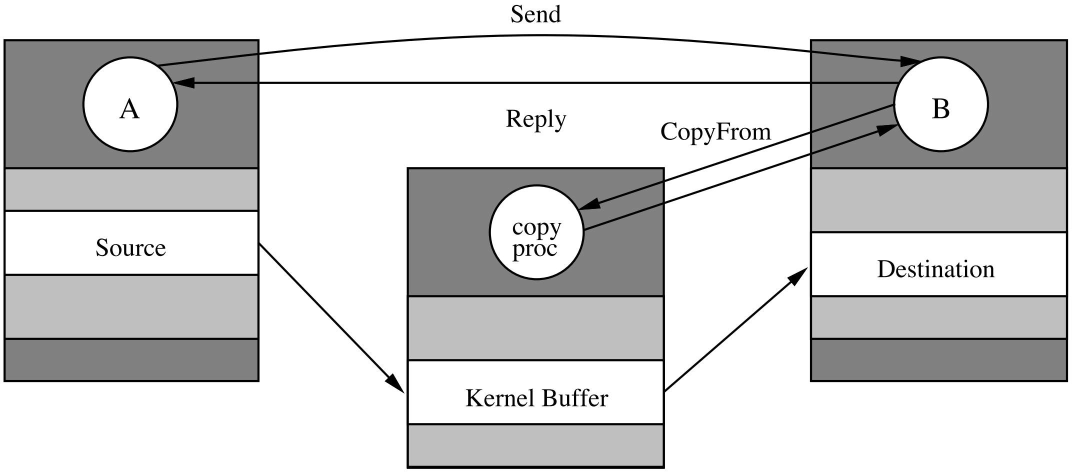

Figure 5.9: Protected Copy in HURRICANE.

labeled `Source` in its address space. Consistent with these permissions, process `B` may have data transferred to or from any portion of this segment through the protected copy system calls. The access permissions last until `B` replies to the message sent by `A`.

The figure shows that the protected copy facility is implemented through a "trusted" server called the copy process, which resides in the kernel address space. This process must be trusted because it must access its client's address spaces and because it must be able to check the permissions before the transfer can begin. The actual transfer is accomplished by first loading from the source segment into a pre-allocated buffer in the kernel address space, and then storing this data into the address space of the destination process.

It is interesting to note that in a prior implementation the copy process transferred the data through mapped regions as shown in Figure 5.10. In this implementation, the copy process binds source and destination regions in its own address space to the corresponding regions of its clients. Once these regions are bound, a simple byte-wise copy by the copy process is sufficient to transfer the data from the address space of `A` to the address space of `B`, because the underlying virtual memory system takes care of all page mapping and coherence concerns. This approach has a number of advantages. First, the data is copied directly from the source to the destination address space, thereby avoiding the intermediate copy into the kernel buffer. Second, the use of shared regions permits copy-on-write optimizations if the virtual segments meet suitable alignment and size restrictions. Finally, the entire implementation is based on standard mechanisms already provided in HURRICANE, so that the copy process could run as a user level (and trusted) server.

In spite of all these benefits, we found that the current implementation, using the intermediate buffer, is up to four times as fast as the mapped implementation for small transfers. The difference in performance is primarily due to the costs, both direct and indirect, of setting up and tearing down the shared regions. The direct costs include binding and unbinding the shared regions; the indirect cost is through additional page

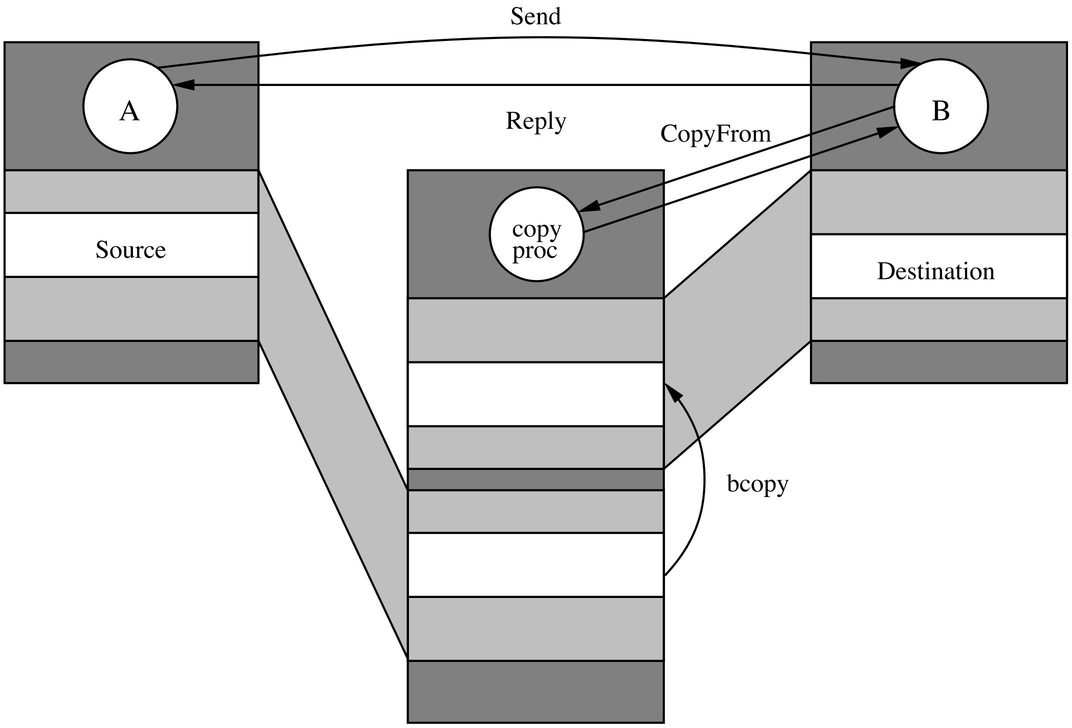

Figure 5.10: An alternative implementation of protected copy.

faults. With shared regions, the copy process must always take at least two page faults: one to map the source page; and one to map the destination page. In contrast, the current implementation uses the address space of each client directly, and usually finds that both the source and destination pages have already been accessed, so that no page faults are generated at all. In the time it takes to service a page fault, the copy process can transfer about 200 data items, giving the current version improved performance on small transfers even with the extra copy.

5.9 Dirty Harry

In a single-level store system, a write to memory is logically a write to the corresponding file block. If this were taken literally, each write to memory would cause an I/O transfer to secondary store. To reduce the amount of I/O, HURRICANE supports a *delayed-write* mechanism [6], which works as follows. A file block that has just been brought to main memory from secondary store is "clean", which means that the data in memory and on secondary store are the same. The first write by an application to the file block sets the `DIRTY` bit of the page descriptor associated with it, but the application continues otherwise uninterrupted. The dirty bit is set to remind the memory manager that the data on secondary store is now inconsistent with the version in memory, and that the page must be written to secondary store (the delayed write) before it can be re-allocated to another file block.

When the dirty page is freed, its page descriptor is appended to the LRU free list of the page cache (as would a free page), where it slowly works its way toward the head of the list. A separate *page scavenger* process is employed to write modified file blocks on behalf of the processes that modified them. This page scavenger has been dubbed "Dirty Harry", because its primary responsibility is to write modified (or dirty) file blocks to secondary store. Like the other processes of the memory manager, there is one page scavenger per cluster, which is responsible for replacement of physical pages local to that cluster.

The blocks targeted for delayed-write are kept on a per-cluster `DirtyList`, which is shared by the memory manager and Dirty Harry. Each element on this list contains a pointer to the page descriptor describing the file block and physical location in memory. This list is used as follows:

1. As the memory manager removes page descriptors from the free list for re-allocation, it will eventually encounter a descriptor marked `DIRTY`. At this time, that entry is appended to the the `DirtyList` and the page descriptor is marked as being on the list.

   Since the memory manager must allocate a page to complete fault handling, it continues searching the free list, repeatedly appending entries to the `DirtyList` until a clean page is found. At this point, the memory manager can complete the page fault and continues with regular fault handling, but also readies Dirty Harry.

   Note that the page descriptors must remain in the page cache while they are being written, because they are valid and represent the correct state of the file data. If the page descriptors were removed from the page cache during the write, processes faulting on the file block would not know that it was already in memory, and would attempt to reallocate the file block from secondary store. Note too that pages can be reclaimed from the dirty list at any time, as long as they are not actively involved in the I/O transfer.

2. Dirty Harry processes the entries on the `DirtyList`. For each each entry on the list, the corresponding page descriptor is marked `DOING_IO`, and a `WriteBlock` request is sent to the appropriate Block Server. By marking the page as doing I/O, processes are prevented from modifying the page contents while they are in transit to secondary store.

3. When the transfer is complete, Dirty Harry will clear the `DIRTY` bit for the file block and remove the page from the `DirtyList`. Since the page remained at the head of the free list during the write, it will be the first to be reallocated (ie. flushing does not "cheat" the LRU ordering of the free list).

On systems with large amounts of physical memory, it may take a long time before a dirty page propagates to the head of the free list. To reduce the amount of data lost should the system crash before the dirty pages are flushed to secondary store, the page scavenger searches the free list periodically, building up the `DirtyList` directly. The dirty pages are then flushed one at a time and at a lowered priority.

Chapter 6

Cross-Cluster Memory Management

6.1 Introduction

The data and control structures that support the management of resources within a cluster are complete in that they provide all the functionality necessary for operations local to the cluster. The structure of a cluster is focused not on scalability, but on attaining good performance for tightly-coupled interactions within the cluster. This chapter focuses on structuring for scalability. Consistent with the structuring philosophy of hierarchical symmetric multiprocessing, the framework for a scalable operating system is laid by instantiating multiple clusters across the processors of the system. Each cluster instance retains its basic service and control structuring, which includes the per-cluster servers and the four primary data structures: address spaces and region trees; the page cache; and the file table.

Having multiple clusters increases the service capability of the operating system, while still maintaining the benefits of locality for operations local to the cluster. However, the clusters must cooperate and communicate to provide applications with an integrated and consistent view of the system. For example, independent programs on different clusters may share data through a common file region; the clusters must then communicate to maintain the consistency of the shared data. Similarly, the processes of a large-scale parallel application may span multiple clusters; in this case the memory manager must support a consistent view of the single shared address space.

These examples imply that some of the resources within a cluster must be shared with other clusters. The way in which the shared resources are accessed determines how they are best managed. There are four strategies for managing shared resources:

1. The resource exists as a single copy in a well-known, but statically determined, location in the system. Static placement permits the resources to be located quickly, although the accesses are typically remote. This strategy is appropriate for data structures that are changed frequently, since maintaining the consistency of multiple copies would otherwise be costly.

2. The resource exists as a single copy, but the location changes over time. Migrating the resource to where it will be accessed can reduce access latency and network contention, but makes it more difficult to locate the resource. Thus, the fast search allowed by static placement is lost. Also, the migration of actively shared resources can lead to thrashing.
3. The resource is always replicated to every cluster, and the replicas are kept consistent across clusters. This strategy allows local and concurrent accesses to the replicated copies, but maintaining consistency is expensive if the structure changes often.
4. The resource is replicated on demand, and the replicas are then kept consistent. The concurrency-consistency trade-off for demand replication is similar to that of full replication above, but demand replication can reduce the cost of replication and consistency if the resource is shared by a subset of the clusters in the system.

In all four approaches to managing shared data, data in remote clusters will need to be accessed, for example: i) to keep replicated copies consistent ii) to fetch a copy of a data item that is replicated on demand iii) when the only copy is remote. Hence, communication with remote clusters is necessary to access remote data. As discussed by Chaves [22] there are three primary methods of remote communication:

Shared Memory is still a viable option, particularly for lightweight operations that make only a few references. While remote shared memory accesses have no direct overhead, they contribute to bus contention and have longer latency than local accesses. In addition, each cluster in HURRICANE resides in a separate address space, which means that data in a remote cluster can only be accessed by mapping the appropriate portion of the remote cluster's address space into the address space of the local cluster. Pointers are especially difficult to handle, and must be managed carefully to avoid aliasing across the local and remote address spaces.

Remote Interrupts or *bottom-half* invocations, are appropriate for heavier weight operations. To perform a remote interrupt, the requesting processor interrupts the target processor so that it can perform the operation. Because the target processor performs the operation, data accesses are local to the interrupted cluster. However, processing the interrupt steals cycles from the process that was executing on the processor, impeding its progress. Light weight operations are not suitable candidates for bottom-half invocation because they cannot amortize the cost of the interrupt (saving and restoring the registers, etc.).

Using remote interrupts as a base, we have built a Remote Procedure Call [13] facility which allows operations performed on remote clusters to appear like procedure calls to the local cluster. The RPC facility uses remote shared memory to pass the arguments of the call.

Message-passing or *top-half invocation*, is best suited to high level operations, where a server process on the target cluster handles the request on behalf of the remote cluster. Communication through top-half invocation should be used for operations that are relatively infrequent or which require significant processing, so that the extra overhead imposed by the message passing system is not significant. In our system, message passing across clusters is implemented using remote procedure calls.

The rest of the chapter is structured as follows. Section 6.2 explains the sharing and communication strategies used to support the management of virtual resources across clusters; Section 6.3 describes the strategies used for global file block management. In Section 6.4, we show how the data structures can be applied to support the coherence of main memory across clusters. Finally, the last two sections review how the cluster and cross-cluster structures interact, by considering in detail the two most important operations: paging (Section 6.5) and unmapping (Section 6.6).

6.2 Managing Cluster Data Structures

The address space and region descriptors are referenced on every page fault, so it is important to keep the cost of accessing this information to a minimum. Consequently, each cluster keeps a local "working set" of per address space data structures used by the processes on that cluster. For programs that span multiple clusters, the shared resources are replicated on demand to the clusters that reference them. The data structures that are replicated in this fashion are the address space descriptors (`ASID`s), and the region descriptors (`Regions`). The shared state kept in these structures seldom changes, (except for copy-on-write regions, which are treated differently, as discussed below).

Because regions may be replicated across clusters, the memory manager must ensure the consistency of the replicated state. For example, operations that reset the length of a region or remove a region must be supported correctly. Consistency is maintained by directing all modification requests to the *home cluster* of the address space, where the global updates are properly synchronized and disseminated to the other clusters.

Currently, the home cluster is designated as the cluster on which the address space was created. This decision was made in preference to other placement schemes because at the time the address space is created it is not known how many processes it will eventually support. Placing the home cluster anywhere but the creation site would thus penalize sequential or small-scale parallel programs that do not span multiple clusters. Since the home clusters are designated on a per program basis, they are naturally distributed across the system as the result of process load balancing.

The per address space data structures are replicated on demand from the home cluster. When a process first migrates to a new cluster that does not yet have any address space state, the `ASID` is replicated from the home cluster to the new cluster using remote procedure calls. The region tree of the replicated descriptor is initially empty — the address space descriptor only contains information about the home cluster and program-wide constants, such as the default backing file.

The region tree is built up in similar fashion as the program executes. When a search for a region fails on the local cluster, the home cluster is asked for a copy of the region, which is then inserted into the local tree. Note that this makes searches for regions that do not exist more expensive than if the regions were not replicated, since the search will fail first on the local cluster and then fail again on the home cluster.

The sub-region lists, which are used by copy-on-write regions, are an example of a data structure that is not replicated across clusters. Instead, sub-region lists reside on the home cluster only, and all operations involving sub-regions are directed to the home cluster. Sub-region lists are not replicated because copy-on-write regions are primarily used for the initialized data section of programs; the lists therefore tend to contain only a few elements that change rapidly as the program starts-up. By not replicating the sub-region lists, the number of remote searches is increased. However, the cost due to the increased number of searches is far lower than the overhead of keeping multiple copies of a frequently changing data structure consistent. Moreover, in the HURRICANE environment, copy-on-write regions are typically small, so that few remote searches are necessary in the common case. As an additional optimization, the region descriptor is globally updated to eliminate the `cowfile` reference, once the entire region has been copied.

Application requests to the memory manager are always sent by message to the local region manager process (see Figure 4.2). If the local cluster is not the home cluster, then the request is forwarded to the region manager on the home cluster. All modifications to address space resources are serialized properly at the home cluster, and the home cluster is responsible for propagating the modification to the clusters that share the data.

The clusters that have a local copy of an address space or region descriptor are kept in a *cluster set* bit vector that is part of the respective descriptor on the home cluster. When a region or address space is modified, the home cluster uses RPC calls to pass the new state of the descriptor to each cluster in the cluster set. Hence, the cluster set is effectively a full-map directory of the clusters that contain copies of a particular descriptor. Because update requests could arrive at any time, each address space on a cluster has a multiple reader single writer (MRSW) lock to synchronize concurrent accesses.

Demand replication of address spaces and regions is important in realizing the performance goals of hierarchical symmetric multiprocessing. The replication itself permits higher concurrency and localized access to the descriptors. Because the regions are replicated only on demand, the number of replication operations is reduced, compared to a full replication strategy. For example, regions private to a particular process of a parallel application are copied only to the cluster where that process executes. Consequently, the number of data structures local to each cluster is minimal, and these data structures are accessed only by processors local to that cluster, which allows the service times in each cluster to remain low. For regions that are shared by several processes within a cluster, the first process on each cluster to access the region replicates it once, for use by all processes in the cluster. The sharing within clusters further reduces the number of replication operations. Thus, having multiple processors per cluster can reduce the amount of cross-cluster communication, compared to a fully-distributed system (or equivalently, having one processor per cluster).

This section has focused on the address space and region descriptor management across multiple cluster instances. The other primary data structures, the page cache, file table, and page tables, also exist on a per-cluster basis. The file table and page tables do not directly share their data with other clusters. Because all the state in the file and page tables is local to the cluster, these structures are maintained independently across clusters. The page descriptors, however, can contain remote file block information, their management is the subject of the next two sections.

6.3 Directories

The most important operations at the physical resource level are those of locating pages, and identifying processors that have references to a page. The first is used to handle a page fault, and the latter is used for page unmappings and invalidations. In the single cluster case, the page cache hash table was used to search for the corresponding page descriptor. For the multiple cluster case, the search must be expanded to include other clusters. In fact, because individual pages may be replicated, the search may be required to return the "closest" copy.

Rather than try to support global operations using the physical resources of the clusters directly, we have introduced a higher level directory whose entries identify the state and location of resident file blocks in a global sense. The directories are also used for synchronization purposes. These directories have the following advantages: 1) they support fast search operations, which is important for scalability 2) besides search, the directory entries can be used as a basis for cross-cluster coherence mechanisms, and 3) synchronization can be easily applied at the level of a directory entry, which permits highly concurrent access. Because accesses to directories are light-weight, in that they typically involve checking a field or setting a bit, directory entries are accessed through shared memory.

The following discussion assumes a directory entry of the following form:

File Block - which is the <`server,token,block`> identifier used to identify the file block cached by a physical page. These fields are used to search the directory as described below.

Member Set - a bit vector with one bit per cluster in the system. If a bit is set, then the corresponding cluster has valid references to the file block, where a valid reference means that the page cache of the cluster has a valid page descriptor for the file block. The member set, together with the state field below, is used to maintain memory coherence across clusters.

State - this field consists of three elements: a dirty bit to indicate whether or not the page has been modified; a busy bit to synchronize accesses to both the page and the directory entry itself; and an I/O bit to denote pages that are in-transit from secondary store.

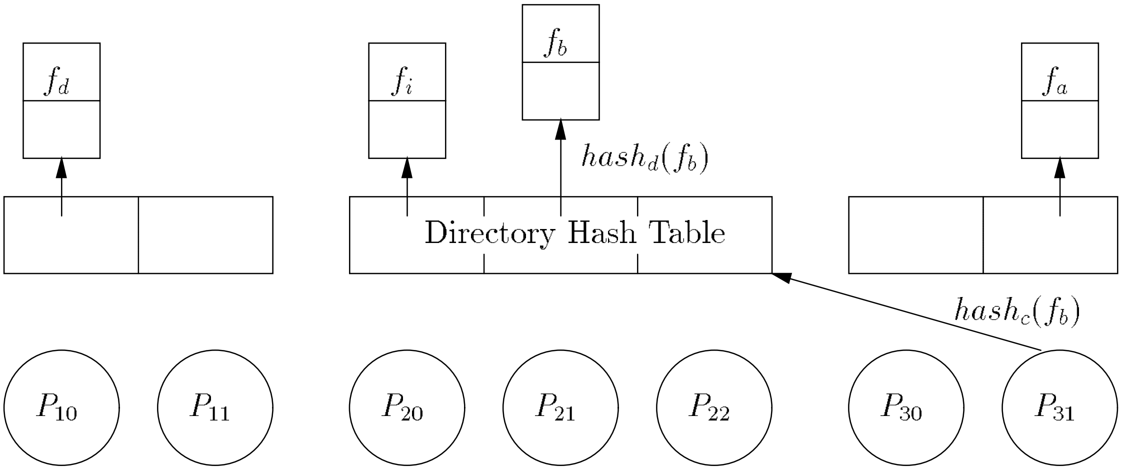

Figure 6.1: A two-level hash function is used to search the global directory table. The first level, $hash_c$, identifies the cluster; the second level, $hash_d$, identifies the per cluster hash bin in which a directory entry resides.

Because there is potentially one such entry per physical page, the size of the directory is proportional to the size of physical memory. The directory entries must therefore be distributed across the system to balance the load and maximize concurrency. We refer to the way in which these entries are distributed as the *placement strategy*. This section presents several placement strategies and discusses their strengths and weaknesses.

6.3.1 Single-Level Directories

Currently, Hurricane supports cross-cluster page operations through a single-level directory, using a distributed two-level hash function to distribute the directory entries evenly across the system. Figure 6.1 illustrates this placement strategy for a 7 processor system configured as 3 clusters. The figure shows the search for a particular entry as a two-step process: first the cluster hash function, $hash_c(server, token, block)$, is used to identify the cluster in which an entry resides; from here the directory hash, $hash_d(server, token, block)$, identifies the bin of the hash table within the cluster that stores the entry.

This placement strategy allows a physical page to be located from anywhere in the system in at most two remote operations: a search of the directory to determine the cluster containing the page; and a search of the remote page cache to find the page itself. By distributing the directory entries in this fashion, a high degree of concurrency can be supported: the per cluster directories each have their own lock, which only needs to be held during a search or update, so that directories on separate clusters may be searched in parallel. As well, the $hash_c$ and $hash_d$ hash functions distribute the entries evenly across the system, which helps to balance the search load. Unfortunately, this load balancing comes at the expense of locality. On a C cluster system, any page has only a $1/C$ chance of being in the local directory. This implies that most directory searches will be remote, adding traffic to the intercommunication network and interfering with processing on the remote cluster.

We had considered putting the directory entries for each file on the cluster of the file server responsible for that file. Because main memory in a single-level store system is simply a cache of secondary store, this approach would be analogous to directory-based cache coherence schemes implemented in hardware, which place the directory entries for cache lines near the memory they are caching. The rationale behind this placement strategy is that the file server must be contacted for page-level I/O anyway, so that the number of remote clusters involved is reduced by placing the directory entries local to the server. This approach also provides order constant time access to the directory entries since the server is encoded as part of the logical file descriptor. One concern, however, is that the cluster containing the I/O server could become a bottleneck, since the searches for the directory entries are directed there, in addition to the I/O requests themselves. A second concern is that the responsibility for balancing the load across the clusters now rests with the I/O sub-system, because directory entries could only be distributed evenly if the physical file blocks they map are distributed evenly. Finally, this scheme only reduces remote accesses when a page needs to be read from or written to secondary store; the directory entry is not necessarily near where the page will be used. Since the per cluster page caches are extremely effective, most accesses to a directory entry are in fact not related to I/O, but instead to unmapping or consistency operations during the course of execution.

To improve locality one could place the directory entry on the cluster that first accesses the file block. This would require an extra level of indirection to the basic directory distribution scheme. The top level of the directory would still be distributed across the clusters and would still be accessed through the $hash_c$ and $hash_d$ hash functions. However, the top level directory entries would no longer maintain the full state of the file blocks, but would instead keep a pointer to the cluster that maintains the complete directory information for the file block. Thus, the first cluster to access a file block sets itself as the *owner* at the top level (by setting the pointer in the top-level directory entry to itself), and then places the directory information in its own cluster. All subsequent accesses by the owner cluster are thus local references. Other clusters that must access the directory information can find it in two searches: the top level search determines the owner cluster; this cluster is then searched to find the entry itself. It may even be possible to avoid the top level search by keeping the owners of frequently referenced directory entries in a local hint table. Because the hint table is local, the number of remote searches can be reduced, which conserves communication bandwidth and reduces access latency. Of course, the search of the hint table can only be justified if the hit rate is high enough to amortize the extra top-level search that results if no hint is present.

This first-hit placement strategy is optimal for sequential tasks or for small scale parallel tasks that run entirely within a single cluster. In association with the hint table, this scheme can exhibit good locality for multi-cluster applications, assuming the processes are scheduled onto clusters near the owner cluster. However, these locality advantages can break down under a number of scenarios, in which case the overhead is far greater than if the top level directory were used alone. Consider, for example, the case where a number of instances of the same execution are simultaneously executed in different clusters across the system, as is common for many system utilities in a multi-programmed environment. Since only one cluster can be the owner, the remaining clusters could wind up needing three look-ups to access the directory entries instead of one if the top level directory is used alone[1].

The break-down of locality stems from the lack of distribution control within the placement strategy itself, which is using the first access to a file block to place its directory entry for all subsequent accesses. This one-time placement decision is even less attractive when one considers the lifetime accesses to a directory entry. Since the page caches are effective, the lifetime of a cached file block is typically far greater than the lifetime of the program that first referenced it. Thus, while the example above specified simultaneous execution of independent programs (a lack of spatial locality), the exact same behavior is seen if the multiple instances of the same program runs on different clusters over time (a lack of temporal locality). It may be possible to reduce this effect by migrating directory entries away from owner clusters that no longer reference the page[2]. The migration is completed by changing the owner identifier in the top-level directory. If hint tables are being used, their entries are now stale, but will be updated when the change of ownership is noticed on the next access.

The problems with the first-hit directory entry placement strategy can be even worse for large-scale parallel applications, particularly those that share regions across clusters. For example, consider a parallel matrix multiply program that must calculate the matrix product $C = AB$. Each process computes the solution for a subset of the rows of the C matrix, which requires access to the corresponding rows of the A matrix and *all* of B. If the program spans multiple clusters, the directory entries corresponding to the independently accessed rows of C and A will be evenly distributed across the clusters, but all of the directory entries for B could reside on a single cluster. This happens because the first process to access the first page of B becomes the owner of its directory entry, and is also the first process to complete its fault processing. Consequently, the same process will also be the first to access the second page of B, and so on, until all the directory entries reside on the same cluster. This behavior was actually observed in HURRICANE in the context of page placement: using a first-hit policy all the pages of B were placed on a single processor. It was these observations that led to the development of the round-robin placement policy.

6.3.2 Multi-level Directories

The single-level directories described in the last section force a flat distribution of directory entries, so that access locality suffers. In very large systems, a lack of locality can degrade performance significantly. Locality and concurrency can both be improved if the directories

[1] The three searches performed by the remote clusters are: 1) search the local hint table, which results in a miss; 2) search the top-level directory entry to find the owner cluster; 3) search the directory entries of the owner cluster to find the state of the file block.

[2] This situation is detected when the owner cluster is not a part of the member set.

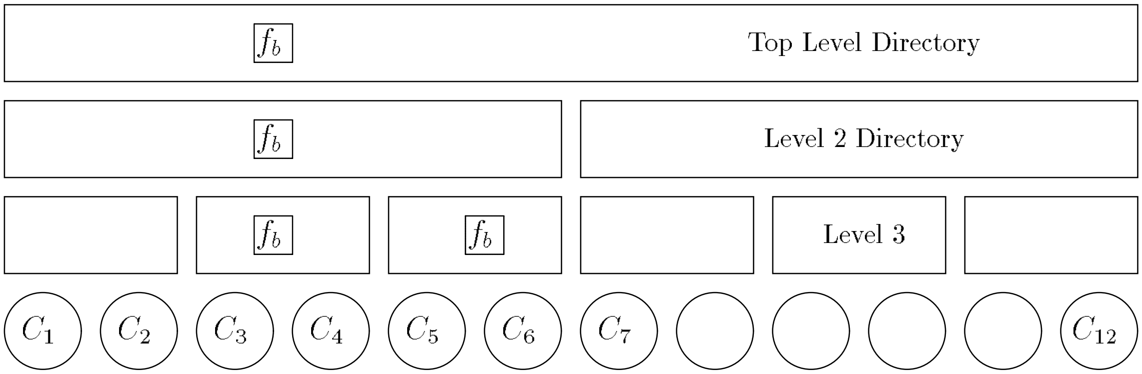

Figure 6.2: A hierarchical directory configured into three levels across twelve clusters.

are structured hierarchically, as shown in Figure 6.2. The circles in the figure are clusters, which form the leaves of the tree. The boxes at each of the L levels are the nodes of the tree, and represent directories with the same entry structures and hash tables as for single-level directories, except that the member sets at level l now identify the children at level $l+1$ that contain information about the particular file block. The entries within a node are still distributed uniformly across the clusters spanned by the node, and are still located through a hashing function.

With hierarchical directories, searches for file blocks not found in the local cluster proceed up the levels of the tree. Locality of access is improved because the entries searched are initially confined to nearby clusters. For example, clusters C_3, C_4 and C_5, C_6 of Figure 6.2 can locate the entry for file block f_b in one Level-3 directory search. Concurrency is enhanced because searches to different nodes at the same level are completely independent. For example, clusters C_4 and C_5 of Figure 6.2 are covered by two different Level-3 nodes, so searches by these clusters can proceed in parallel.

If a particular file block is not found in either the local cluster, or the Level-3 directory, the search continues up the tree until a directory entry for the block is found or the root of the tree is reached. Thus, the first level search for f_b fails for cluster C_1 in the figure, but succeeds at Level-2. The information obtained at this level indicates that either of the Level-3 nodes covering clusters C_3 and C_4, or C_5 and C_6 can be searched to identify a cluster that contains a physical page caching the file block. Similarly, for clusters C_7 through C_{12} the search for file block f_b fails at the first two levels but succeeds in the Level-1 directory, so that in general, any page can be located in at worst $2L$ searches.

It is interesting to note that if the search was always started at the root of the tree, only L searches would be required. However, these L searches would *always* be required, and would negate the locality benefits of the hierarchical structuring. Also, the $2L$ accesses for a bottom-up search is the worst case; the number of accesses in practice depends on the system workload. In any event, a non-trivial hierarchical directory will always require more searches than a single-level directory. This extra overhead is amortized by the locality and concurrency benefits, particularly in large systems where network bandwidth must be conserved, and where access latencies to remote clusters can be high.

Hierarchical directories can be viewed as a placement strategy that replicates the state of a directory entry to provide better locality of access. Because the state is replicated, the consistency of this state across entries becomes an issue. For example, if file block f_b of Figure 6.2 is modified in one of the clusters, the coherence policy in effect may require that the dirty bits be set in all the directory entries for the file block. Other operations that may potentially modify multiple directory entries are unmap and invalidate. When such global state changes are required, the operation should appear atomic, or else some entries will contain temporarily inconsistent state. The integrity of the directory can be preserved by introducing locks at each node and imposing a protocol on their acquisition. The locking protocol is required to prevent deadlock if several locks must be held simultaneously. Deadlock can be avoided by requiring that multiple locks in different levels be acquired by progressing from the top of the tree down towards the leaves, and that locks within a level be acquired in order of increasing cluster number.

This locking protocol dictates that modifications to a file block entry must start at the root of the tree and propagate down to the leaves. Searches can still proceed up the tree, provided they can prevent the node from being modified while it is being searched. If the search of the node as level l fails, the node must be unlocked before the node of its parent (at level $l-1$) can be searched. A race condition can result here if the tree is modified between the time the lock is released at level l and acquired at level $l-1$. This condition is easily detected by checking the member set of the parent node: if the search of the child failed then it was not a member at the time of the search; if the child is a member when the parent lock is obtained, then the tree was modified and the search is retried in the child. The restriction that modifications proceed down the tree does not impact concurrency severely, because the locks at one level only need be held long enough to make the change and acquire the locks at the next level down. Thus, multiple updates can proceed simultaneously in pipeline fashion.

6.4 Main Memory Coherence

Main memory coherence is required to support policies that replicate physical pages. For example, the default page-placement policy in HURRICANE replicates shared pages across clusters to reduce contention and increase locality. This section discusses mechanisms and issues related to extending the full-map directory protocol used for cache coherence within clusters to the next level of the storage hierarchy: that of main memory coherence across clusters.

The basic concepts of main memory coherence are similar to those of cache-coherence, but there are some differences in mechanism because the caches are implemented in hardware, while main memory consistency is implemented entirely in software. For example, caches automatically replicate the data of a cacheable page on a line by line basis. To achieve analogous locality benefits at the main memory level, the system must provide a primitive to replicate data pages. Moreover, since the cost of replicating an entire page is much higher than the cost of loading a cache line, the policy decisions that govern replication may be different at the two levels. Specifically, the cost to replicate a page can only be justified if the number of memory accesses to the new copy can amortize the initial overhead. Otherwise, the line level replication already supported in hardware will result in a lower total access cost. Bolosky has shown that knowledge of future accesses is required to make an optimal decision about when to replicate a page [18]. Consistent with the hierarchical symmetric multiprocessing structuring philosophy, the heuristic used in Hurricane is to share main memory pages within clusters, and to replicate them across clusters. Of course, this default policy can be over-ridden by applications that have prior knowledge about their memory access patterns.

Our design requires that even remotely accessed pages have a local descriptor to hold coherence and housekeeping state needed by the cluster. One reason for this is that page descriptors in remote clusters cannot be accessed directly, because they are in a different address space. While remote shared memory could still be used to access page descriptors across clusters (even if this access is somewhat complicated), manipulations to a page descriptor are usually accompanied by other operations, such as flushing a cached page, or resetting a page table entry. The heavier-weight of these operations makes RPC the appropriate communication medium; the page descriptor manipulations are bundled as part of the remote procedure call handling.

Currently, the state for remotely referenced pages is held in a special page descriptor called a *representative*. Representatives have the same fields as normal page descriptors, so that they can be inter-mixed in the page cache. However, representatives have their own `Core` structure for free list management, and the `addr` field contains the physical address of a remote page.

In the worst case, all mappings within a cluster could be to remote pages, which implies that there could be one representative per physical page in the system. If this worst case were common, the design would not be scalable or even practical, but we have found empirically that a relatively small number of representatives is sufficient for most applications. To handle the remaining cases, the techniques of virtual page replacement [40] can be applied to reclaim the least recently used representatives for re-assignment. The limited number of representatives could lead to performance degradation if all the representatives are a subset of the working set of a single program, since a type of thrashing can occur as the representatives are repeatedly reclaimed and re-assigned a short time later. However, program referencing this many remote pages has extremely poor locality, and would not perform well under any circumstances.

The remainder of this section shows how the coherence policy represented by the state transition diagram of Figure 6.3 can be applied to support both main memory and cache coherence. The policy is similar to the MRSW coherence policy of Section 5.5, with two important differences. First, file blocks that are read shared across clusters are replicated onto each cluster that accesses it. This replicated state is called *Multiple Clusters and Readers*, or MCR. Second, when a file block moves from the MCR to the MRW state, all but one of the replicas is removed from memory. Processors remote to the cluster containing the single copy of the file block must then access the data remotely.

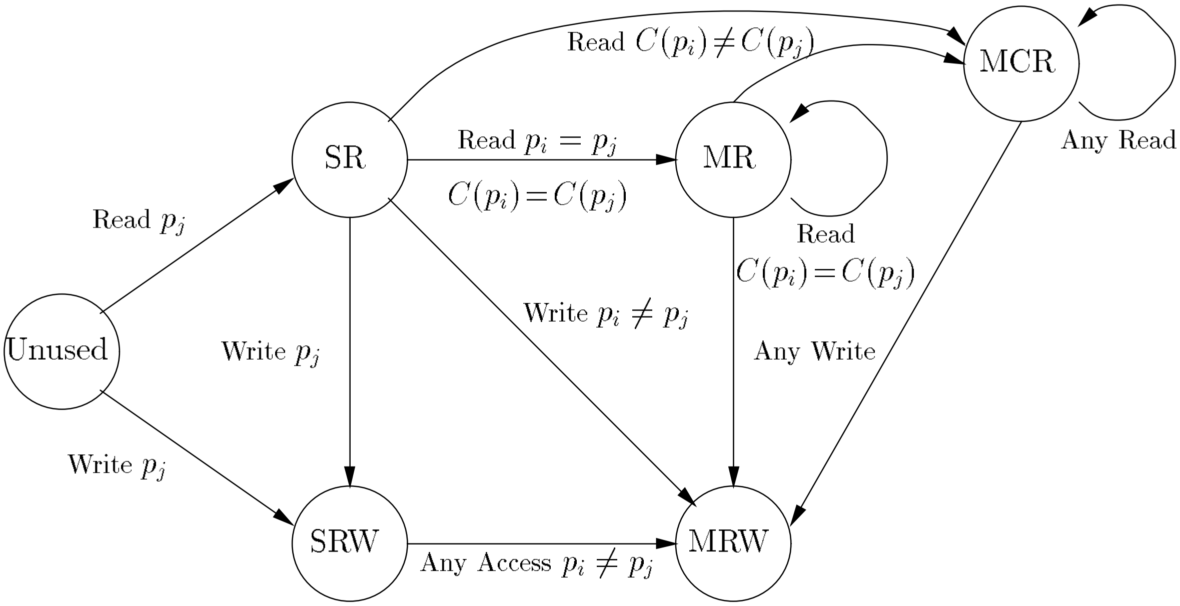

Figure 6.3: The state transition graph for a coherence policy that replicates pages shared for reading across clusters.

In this hierarchical configuration, cache coherence is still maintained within clusters through the mechanisms of Section 5.5. As a result, a single cross-cluster uncache page operation will uncache the page on all processors in the member set of the cluster. Because memory consistency is not supported directly in hardware, the coherence state transitions are driven indirectly when a processor in a cluster determines (through the directory) that a global operation is required. In the same way, operations such as uncache are applied to entire clusters.

To illustrate the cross-cluster coherence protocol, we now step through the actions that result as a page is initialized and stepped through the SR, MCR, and MRW states in sequence. The discussion assumes that single-level directories are used. When the page is first accessed, the local page cache search and subsequent directory search reveal that the page is not yet resident in memory. At this time, the page is requested from secondary store, page descriptor and directory entries are allocated for the page, and the member sets at both levels are initialized to identify the processor and cluster that is requesting the data. In addition, the `DOING_IO` bits are set to inform the other processors that the page is in transit, but that the data is not yet valid. Assuming the initial access was a read, the page moves to the SR state when it arrives from secondary store.

If a processor in a different cluster attempts to read the page, the local page cache search will fail because the page has never been accessed on that cluster. However, the directory search will find that the page is in memory, that it has not yet been accessed for a write, and will also indicate the cluster in which the data resides. The faulting processor can then obtain a local replica of the page and add itself to the member set of both the page and directory structures. Note that the directory entry must be locked during the replicate operation to prevent other clusters from attempting to write the page while it is being copied.

At this point, two identical copies of the page exist, and the state of both is SR within their respective clusters and MCR across clusters. Further reads by different processors within these clusters will move the local state to multiple-readers, but will not affect the global state. Indeed, the directory is not even searched for these accesses, because the local page descriptor state contains all the information needed. These accesses may also proceed concurrently, because the structures and pages are independent across clusters. If a replicated page is unmapped, it is unmapped in all clusters that have an active mapping, but is left in the local page cache of the clusters to avoid the cost of replication should the page be accessed again in the near future.

To complete the illustration, assume that one of the processors in either of the two clusters attempts to write the page. Since this is the first write access, the `MODIFIED` bits must be set at both levels of the consistency hierarchy. Even if the page is in the SR state within the faulting cluster, the system recognizes that a replica exists and moves the page to the MRW state. On the remote cluster, the replica page and the caches containing data from the page must then be invalidated, and the page descriptor must be replaced by a representative. The physical address in the representative page descriptor identifies the remote page, which can be used in the page tables to access the data directly.

This example has assumed that the initial write was attempted by a member of the existing cluster member set. If the write originated from a cluster outside the member set, the local page cache search would fail and the directory entry would indicate that the page is replicated across two clusters. The state in all three clusters must therefore be moved to the MRW state, but two courses of action are possible. Because the faulting cluster does not yet have a mapping to the page, it could either migrate a copy of the page locally before invalidating the remote copies and replace the remote page descriptors with representatives pointing to the local copy, or it could invalidate one of the copies and create a representative locally that identifies the remaining (single) copy of the page. The justification for the former decision is that the processor that caused the fault will likely access the page again in the near future, so that making the page local will reduce the latency of these accesses; Hurricane has adopted the latter policy as the default to avoid the cost of replicating a page that is moving to the uncached state.

6.5 Demand Paging

The Hurricane memory manager is primarily demand-driven, which means its mechanisms are implemented by delaying actions until necessary, rather than trying to predict application requirements. When a translation fault is recognized, the processor saves the state of the faulting process and passes it on the memory manager. This state information includes, among other things, the process' registers, the virtual address that caused the fault, and the type of access (read or write). The memory manager uses this state and the algorithm of Figure 6.4 to determine the actions needed to resolve the fault.

```
ALGORITHM HandlePageFault :

    find the region containing the faulting address
    determine the file block containing the faulting address

    search the local page cache for the file block

    IF the file block is not locally resident THEN
        allocate a page and page descriptor for the file block

        search the directory for the file block

        IF the file block is not globally resident THEN
            initiate the I/O transfer
            RETURN
        ELSE
            % the file block is resident in a different cluster
            determine the global state of the file block

            CASE action( global state)
                Replicate:
                    copy the remote page to the local cluster
                Remote Access:
                    free the page and descriptor
                    create a representative for the remote block
            ENDCASE
        ENDIF
    ENDIF

    IF the faulting access was a write THEN
        IF the file block is marked copy-on-write THEN
            allocate a backing store page to the process
            copy the source page to the backing store page
        ENDIF
    ENDIF

    check the coherence policy to determine the mapping permission
    map the page into the address space of the faulting process

    RETURN
```

Figure 6.4: The algorithm used to handle translation faults.

The first step is to search the tree of regions bound into the process' address space for the one containing the faulting address. As discussed in Section 5.2, the region descriptor maintains the file to which the region is bound and the mapping attributes of pages in the region. The file region and virtual address together identify the file block, which is used as the key to search the local page cache of the cluster. If the file block is found to be locally resident, the next step is to determine the mapping permissions. The mapping permissions are a combination of the region attributes (for example, read-only), the coherence state of the page (SR, MRW, etc.), and whether or not the page is marked for copy-on-write.

If the file block is not found in the local cluster, a new page and page descriptor is allocated for the file block. It is possible that the page and descriptor will not be needed, since later processing may determine that the page data must be accessed through a representative. However, the page descriptor is allocated immediately because it is needed for synchronization during the fault handling. If the file block was not found in the local page cache, the directory is queried next to see if the block is resident anywhere in the system. If the block is resident in a remote cluster, the algorithm proceeds to the coherence check phase, which will identify whether the page should be replicated to the local cluster or accessed through a representative.

The final possibility is that the file block is not yet initialized anywhere in the system, either because this is the first access, or because it is already in transit from secondary store. In both cases, the local page descriptor is marked `DOING_IO` to prevent other processes from accessing it prematurely. From this point the fault handling algorithms for the two cases diverge, as described below.

On first access, a directory entry is created for the file block with its initial state set to `DOING_IO`. If the check of the directory entry reveals that the file block is in transit, but to a different cluster, the faulting process is blocked by placing the descriptor on a waiting queue rooted in the `FileTable` entry for the file. In addition, the cluster is added to the member set of the directory entry, to inform the initiating cluster that a process is waiting. Once the directory and page descriptors have been marked as `DOING_IO`, other processes that fault on the page will find from their local page caches that the page is already on its way, and are blocked in their local `FileTable`s.

Once the file block transfer is complete, some clean-up must be performed before the faulting process can be restarted. First, the `DOING_IO` bit is cleared in both the page descriptor and directory entry. Second, other processes queued in the local `FileTable` waiting for the file block to arrive must be signaled and readied. Third, if the directory indicates that other clusters have waiting processes, a remote procedure call is used to inform the memory managers on these clusters that their processes can be readied. Since the Block Server transferred a single copy of the file block, the clean-up on these remote clusters will involve a replicate to make the page local, or the clean-up will involve the creation of a representative to allow a remote mapping. Finally, the physical page is mapped to the address space of the faulting process and the process is restarted.

6.6 Unmap

This section describes the algorithm for unmap, which removes the virtual to physical translations for a possibly replicated file block. This operation is important for several reasons. First, it is used frequently, since every page that is mapped must eventually be unmapped. Second, the algorithm is representative of other operations that modify page table entries, such as page invalidate or uncache. Finally, the unmap algorithm is an example of a compound request, because the possibility of replicated pages means that multiple resources are involved. A compound request permits concurrency to be exploited in its servicing.

The algorithm for unmap is given in Figure 6.5. The immediate observation is that the algorithm is split in two: `UnmapCluster` locks the directory entry for the file block and calls `UnmapPage` on each of the clusters in the member set. On each cluster called, `UnmapPage` searches the local page cache to lock the page descriptor of the file block. The page descriptor is required for three reasons. First, the `IN_USE` bit is checked to see if the page is actively mapped by some process in the cluster. If the page is not in use, no further action is necessary. Second, the member set of the page descriptor is used for TLB consistency, and to invalidate the caches of local processors accessing the page. The cache invalidate is necessary on all systems with physically addressed caches, even if the system supports hardware cache coherence. This is because the page, once freed, could be assigned to a different file block. If the caches are not invalidated, the processors could still be caching the contents of the original file block, and see incorrect data.

Assuming the page is in use, the next step of the algorithm is to remove the virtual to physical translation from all the active mappings in the cluster. The address spaces and regions that could be referencing the file block are determined by searching the `FileTable`. For each region found, the virtual range and file offset of the region gives the corresponding virtual address in the address space that must be invalidated.

Once all the mappings in the cluster have been removed, the page descriptor is checked to see if it is a representative. For these remotely accessed pages, the representative is discarded. If the file block was replicated to the local cluster, the descriptor for the replica is placed on the free list but remains valid and in the page cache. This avoids the overhead of replication should the file block be accessed again in the near future.

The hierarchical decomposition of the unmap algorithm permits parallelism to be exploited both within and across clusters. Within a cluster, the caches and TLBs are invalidated concurrently by using a two-phase protocol. In the first phase, the master initiates the cache operation on all the processors in the page descriptor member set through interrupts, then performs the same operation on its own cache if necessary. In the second phase, the master waits for the other processors to set a flag signaling their completion. An extension of this asynchronous RPC is used to exploit parallelism across clusters as well. The master cluster first initiates the unmap operation on all remote clusters in the member set of the directory entry. The processors on the remote clusters can search their page caches and file tables concurrently to perform the unmap on the local cluster. The

```
ALGORITHM UnmapCluster( virtual address, address space ) :

    find the region containing the virtual address in address space
    determine the file block containing the virtual address

    lock the directory entry for the file block

    DO for each cluster in the member set of the directory entry
        CALL UnmapPage( file block )

        IF the page descriptor was a representative THEN
            remove the cluster from the member set
        ENDIF
    OD

    unlock the directory entry for the file block
END UnmapCluster

ALGORITHM UnmapPage( file block ) :

    lock the page descriptor for the file block

    IF the page is IN_USE by some process THEN
        search the file table for the file

        DO for each address space that binds the file block
            remove the virtual to physical translation
            invalidate the page from the caches if necessary
        OD

        IF the page descriptor is a representative THEN
            invalidate the representative descriptor
        ELSE
            free the page for possible reallocation
        ENDIF
    ENDIF

    unlock the page descriptor for the file block

    RETURN
END UnmapPage
```

Figure 6.5: The algorithm used to unmap a page.

operation is complete when all the remote clusters have responded to the initiating cluster. In this discussion, “master” refers to the cluster that initiated the `UnmapCluster` call, or the processor that fielded the `UnmapPage` request. Because all clusters and processors are symmetric in their capabilities, HURRICANE does not support the notion of a pre-defined master.

The asynchronous RPCs within and across clusters are combined to form a two-level spanning tree of the system, which increases concurrency and helps minimize the service time of the operation. However, other optimizations are also possible. For example, the algorithm as given in Figure 6.5 operates on one file block at a time. Thus, the costs of the RPC and cache invalidate operations are incurred on a per page basis. Operations that deal with sequences of file blocks, such as unbinding a region of deleting a file, could amortize the overhead by operating on the entire sequence of blocks.

The unmap operation as described is global — it unmaps all the pages from all the address spaces in all the clusters that access a file block. The global semantics are necessary for consistency when pages are invalidated because a file is destroyed, or when uncache is called as part of a MRSW coherence policy. However, unmap is often invoked as part of a single `DestroyAddressSpace` call, and need not have global repercussions. In particular, if several instances of a program are executing simultaneously, the code pages may be replicated across clusters, and certainly exist in several sets of page tables. When one instance of the program completes and its resources are released, the current algorithm also unmaps the code pages of all the other program instances, forcing unnecessary page faults. To reduce the amount of interference, a version of unmap could be supported that works on a single cluster of address space, provided that the member set information at the page and directory levels is maintained correctly.

6.7 Summary

This chapter has introduced a number of data structures and mechanisms. Although the data structures are applied to different purposes, there are a number of common themes as a result of the hierarchical symmetric multiprocessing structuring. Here we summarize these themes by relating them back to the design guidelines of Section 3.3.

Preserving Parallelism: Data structures at both the virtual and physical levels are replicated on demand across the clusters of the system. Replication preserves the parallelism of application requests because the number of resources, and the number of service points, grow to meet the service demand.

Bounded Overhead: One of the primary approaches to bounding overhead is to choose search strategies that are independent of the size of the system. This approach is illustrated at the physical level by directories, which permit the location of any file block to be determined in order constant time. At the virtual level, the home cluster concept permits any region to be located in at most one remote search.

The replication of resources also helps to keep service times constant as the system grows, because contention and access latency to the replicated data structures is reduced.

The physical data structures in HURRICANE grow proportional to the size of main memory, which meets the bounded space guideline of Section 3.3. The exception to this rule are representative page descriptors. Although these data structures could grow proportional to the size of virtual memory, scalability is preserved by limiting their number to a constant that is proportional to the size of main memory.

Preserving Locality: The replication of resources preserves locality by placing the copies close to where they are accessed. The data structures also preserve the locality of applications by supporting the implementation of higher level policies that allow replication and caching of application data. Because replication is on demand, only globally shared data structures are actually replicated. Data structures that are not actively shared are always placed local to where they are accessed, and many structures are not shared at all. For example, the `FileTable` and page tables are maintained independently on a per cluster basis.

The data structures are flexible in that they allow performance tuning to different architectures by adjusting the cluster size. Choosing cluster sizes smaller than the number of processors in the system lessens contention in accessing a particular virtual resource, because there are fewer processors accessing each resource and because these accesses are local to the cluster. On the other hand, choosing a cluster size greater than one results in fewer remote operations, because resources are replicated once per cluster. Thus, by choosing the cluster size appropriately, both contention and communication costs can be minimized.

Chapter 7

Experimental Results

The primary goal of HURRICANE is to investigate scalability by using hierarchical symmetric multiprocessing to exploit locality and improve concurrency. Previous chapters have discussed the issues involved and presented a design that we believe meets the research objectives. This chapter attempts to evaluate our implementation to determine how well the system scales, and the factors that affect its performance.

In many ways, the task of evaluating a system is as challenging as designing the system itself. First, to measure the memory management sub-system, the experiments should be designed to exclude effects due to the hardware, other parts of the operating system, and the test program itself. Second, each experiment should exercise a single aspect of the memory manager, so that the results can be interpreted unambiguously. This task is made more difficult because scalability, which we are seeking to investigate, is itself difficult to define quantitatively. Finally, the experiments must be performed on the available hardware platform, which had only 16 processors at the time the measurements were obtained. Thus, the experiments should attempt to demonstrate scalability effects even on a relatively small system, so that extrapolations to larger systems are possible.

Considerations of experimental design raise the issue of software architecture and its separation from implementation. Some of the designs discussed, particularly for directory structuring, may allow the system to scale further but generally have a higher base overhead, which makes them difficult to evaluate on a small platform. As well, HURRICANE is an evolving system; it is only partly optimized, and many issues related to scalability have yet to be explored. Although these challenges appear daunting, we believe the experiments presented in this chapter meet many of the design criteria, and yield valuable insight into the behavior of the system.

The general reasoning behind the experiments is as follows. Hierarchical symmetric multiprocessing was developed from the assumption that while tightly-coupled systems demonstrate high performance, they cannot scale because the shared resources must eventually saturate as the demand on them grows. Conversely, distributed systems appear to scale well through replicated service sites and data structures, but typically have higher communication overhead due to this replication. These trade-offs are illustrated concep-

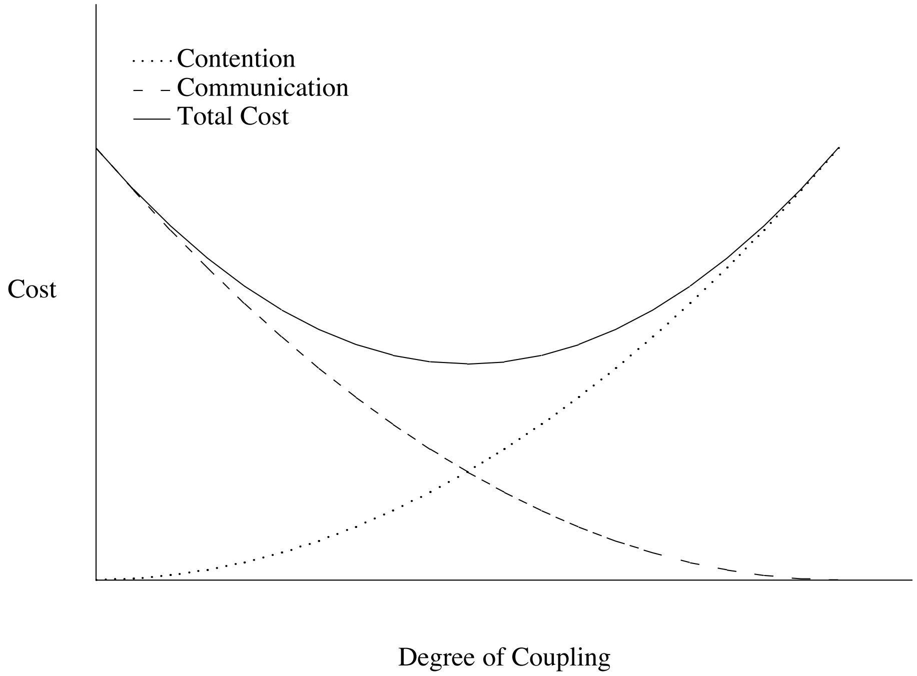

Figure 7.1: A conceptual illustration of the trade-offs in contention and communication costs as a function of coupling.

tually by the graph of Figure 7.1. The graph plots cost, measured in response time or number of operations, against the degree of coupling, or sharing, for some fixed number of processors. The dotted curve in the figure represents contention, which increases as the coupling becomes tighter because of the increased demand on shared resources. The dashed curve represents the cost of remote communication, which decreases with increased sharing because, in general, fewer accesses are remote. Neither curve is linear, which accounts for second order effects such as memory and bus contention. The solid curve is the sum of the contention and communication costs, and suggests that there is some degree of coupling for which the overall cost is minimized.

Hurricane is uniquely suited to exploring the trade-offs in contention versus communication as a function of coupling. For a given number of processors, say p, configuring the system into p clusters containing one processor each effectively mimics the behavior of a fully distributed system. Similarly, if the system is configured into one cluster with p processors, then all data and data structures are shared. If the cost trade-offs can be observed by changing the degree of coupling (the cluster size of the system), and in particular if a minimum cost is observed for some intermediate cluster size, then the fundamental thesis of hierarchical symmetric multiprocessing is proven.

The experiments begin by measuring the performance of the basic memory management primitives: page mapping and unmapping. These primitives are examined first for a single process, and then under increasing levels of contention in an attempt to establish bounds on system performance. The last section of this chapter investigates the performance of three parallel applications to see how their response times are affected by hierarchical symmetric multiprocessing.

The hardware platform used for the measurements is a 16 processor Hector multiprocessor, configured as a local ring with 4 stations of 4 processor modules each (see Section 2.2). Although the experiments configure the operating system into different cluster sizes, the hardware configuration never changes. Each processor module contains 4 MBytes of local memory, a Motorola 88100 CPU, and 2 Motorola 88200 CMMUs [45]. The entire system is clocked at 16 MHz.

7.1 Basic Primitives

This section examines the performance of page mapping and unmapping in an uncontended environment. Good performance of these primitives is crucial to the performance of more complex operations that build on them, including application programs. Although page mapping and unmapping are referred to as primitives, these operations are of course composed of even lower level primitives, primarily data structure manipulations and hardware dependent housekeeping. Consequently, the experiments in this section measure both the aggregate and component time of each primitive, so that their behavior can be better understood.

All experiments in this section were run with the operating system configured into 4 clusters of 4 processors each, with all servers running. However, the system was otherwise idle. The times were measured using the system's microsecond timers, and the numbers reported are the average over several thousand operations. To remove start-up effects, the pages are all pre-touched so that I/O and initialization is excluded.

7.1.1 Read Faults

The first experiment measures the time it takes to demand map a page for read access. The experiment was performed using a simple program that binds a region of 256 pages into its address space, and then repeatedly accesses a single word on each page to cause a page fault. The pages are unmapped between successive iterations of the loop so that subsequent accesses always have to remap each page.

The average time per page fault was measured to be 160 μsec. This time can be broken into three primary components: the hardware dependent fault overhead; the region search; and the page cache search. Note that since the page is always found in the local page cache, and since the access is a read, the directory need not be consulted. The hardware dependent overhead includes the time to recognize the fault, save and restore the registers, set the page table entry, and emulate the access. The cumulative time for these operations was measured to be about 36 μsec. The region tree is searched to translate the virtual address into a logical file block. This search time depends on the number of regions currently bound to the address space, and on the contention for the MRSW lock that protects it. The program for this test had 5 active regions and no lock contention, resulting in a search time of 43 μsec. This figure reported includes the time to obtain and release the MRSW lock for the address space.

The page cache search time was measured to be 33 μsec, including the lock overhead. Like the region tree, this search time depends on the contention for the lock protecting the page, and on the number of pages in the cache. Although the page cache is implemented by a hash table, hash conflicts are resolved by linking the pages into an overflow list. The probability of having to search multiple entries increases with the number of valid physical pages in the cluster. The search time reported is for a newly booted system, so there are relatively few valid pages.

7.1.2 Write Faults

The previous section measured the time required to map an unshared page for reading. After this operation, the page is in the SR state, so it is write-protected. If the first access to the page is for a write, or if the page state later moves to SRW, the same fault sequence is followed, except that the page must also be registered as dirty in the directory. The additional time for this directory lookup raises the SRW fault time to 230 μsec. The directory search time, 43 μsec, is somewhat higher than the page cache search time of 33 μsec, even though the directory and page caches have similar implementation structures (a hash table with overflow chains). However, the page cache search is completely local to the cluster, while three out of every four directory look-ups are to remote clusters, because of the uniform placement strategy of directory entries.

In addition to the directory search itself, the directory locking protocol requires the lock on the page descriptor to be released during the directory search. Thus, a second page cache search is required after the directory check to reacquire the lock on the page descriptor. The directory search and extra page cache search together account for the difference between the read fault time (160 μsec) and the write fault time (230 μsec).

A more interesting case is when a write fault causes the page state to move to the MRW state. To maintain cache coherence on HECTOR, this operation requires a flush/invalidate of the processors caching the page, and is accomplished through remote interrupts. To measure the time for a MRW fault, the simple program of the previous section was extended to include a second process running on a different processor. The first process still runs through the region of 256 pages, initializing each to the SR state, but then blocks while the second process writes a word to each page in the region, causing an MRW fault for each page. The pages of the region are then unmapped and the above sequence is repeated. The time for this type of fault was measured to be 506 μsec when the two processes are on the same cluster. The contribution due to the remote interrupt, including the cache/TLB invalidate and page table update, was approximately 90 μsec.

When the writing process is on a different cluster, the time per fault is 828 μsec. There are several reasons for this substantial increase in time. First, the search of the page cache on the writer's cluster results in a miss[1], because the page was mapped on the reader's

[1] A miss is typically more expensive than a hit, because all the entries in the overflow chain must be

cluster. The subsequent directory query shows that the page is already mapped for reading on the other cluster. To move to the MRW state, a Representative must be allocated on the writer's cluster, and an RPC is needed to initialize its state from the reader's cluster. This also means that a second page cache search is needed, this time on the reader's cluster, to determine the physical address of the shared page. The last step of the fault is to invalidate the reader's processor cache, but since it is in a different cluster, an RPC call is used to perform the operation.

An RPC is similar in many ways to the remote interrupt used to flush another processor's cache, except that a lock is acquired to protect the operation. The measured time for the null RPC was 58 μsec. The times to acquire the Representative page and perform the remote cache flush, including the RPC overhead, was measured as 175 and 250 μsec, respectively.

7.1.3 Unmap

Like write faults, the cost of unmapping depends on which processors are caching the page and where they are located in the system. The simplest case is when the page is cached only by the processor performing the unmap, since no remote operations are required. If the page is cached by a different processor within the same cluster, a remote interrupt is needed to flush/invalidate its cache. Finally, if the page is cached by a processor on a different cluster, the remote cache operation is handled by an RPC call to that cluster.

To measure the cost of these operations, the write fault program of the previous section was modified so that instead of writing to each page, the second process unmapped the pages of the region. Having the second process do the unmap permits more freedom in controlling which processor initiates the operation.

The time to unmap a page cached only by the local processor was measured to be 323 μsec[2]. Like the page fault primitives, this time contains contributions from both the hardware and data structure maintenance; however, the specific operations are very different. The hardware dependent times include the cache and TLB invalidations and the resetting of the page table entry (about 62 μsec). The data structures that need to be searched for this operation are: the directory to determine which clusters map the page; the page cache to obtain the page descriptor, which identifies the processors within the cluster that cache the page; and the File Table, which identifies the address spaces and regions that bind the file containing the page. The directory and page cache search times are as reported earlier; the time to search the File Table was measured as 120 μsec.

If the page is cached by a different processor within the same cluster, a remote interrupt is required to have the processor invalidate the corresponding cache/TLB entries and remove the page table entry. The measured time for this operation was 336 μsec. The

searched to determine that the page is not present.

[2] An application unmap request is a system call whose latency is highly dependent on the current system state. Consequently, the times reported were obtained around the unmap call inside the kernel, which removes the variability of results due to system call handling.

component breakdown for the additional overhead is the same as for the write fault case above.

Finally, if the page is cached by a processor in a remote cluster, an RPC call is used to perform the unmap remotely. If the page descriptor is not a representative it is left in the remote page cache, otherwise the representative is freed and the corresponding directory entry is cleared. The measured time for a cross-cluster unmap with no representative is 362 μsec.

7.2 Performance Bounds

The previous section determined a lower bound on the response times of page mapping and unmapping operations by examining the performance of these primitives in the absence of contention. This section builds on these results by adding contention to the point where the system starts to saturate. This type of bounds analysis is important for two reasons. First, it allows the determination of maximum throughput, which is an indicator of the scalability of the system. Second, performance bounds can be applied to the study of application performance, since applications cannot achieve more throughput than is obtainable from the synthetic stress tests developed in this section.

Contention relates to the demand placed on a resource by its clients, and manifests itself in many ways, such as bus or memory contention at the hardware level, or lock contention in software. The tests in this section are primarily concerned with the contention for memory management resources, and will focus on two types: the contention caused by simultaneous requests for different resources; and the contention caused by simultaneous requests for the same resource. Although the resource demand is applied in a controlled manner, the response times are measured from the application level, and therefore include memory and bus contention caused by the test.

The experimental setup used for this section is similar to that of the previous section: the system is fully configured with all servers running, but no other users were active. As before, start-up costs such as process migration, region replication, and page initialization are not included. All processes were scheduled onto different processors in a way that minimizes the number of clusters spanned. That is, new processes are added to one cluster until each processor within it has a process, before any processes are added to any other cluster. This allocation algorithm corresponds to a scheduling policy that attempts to assign related processes to the same cluster to maximize locality.

7.2.1 Independent Page Faults

This section measures the throughput of simultaneous mapping faults to independent pages. To obtain these values, the basic read fault program of Section 7.1.1 was modified to allow any number of processes to run simultaneously. Each process binds its own private region of 256 pages, and repeatedly reads the first byte of each page to cause a page fault. The pages of each region are unmapped at each iteration of the loop. Because each region is bound to a different file segment, the faults are all to different physical pages. To maximize contention, the processes barrier before and after the fault intensive portion of each iteration. The degree of contention is controlled by varying the number of processes running simultaneously.

Figure 7.2 shows the average response time for mapping faults for cluster sizes ranging from 1 to 16 processors. The fault times vary from 142 μsec when one process is running on a single cluster of size 1, to 512 μsec per fault when 16 processes are simultaneously faulting on 16 processors configured as a single cluster.

The most noticeable feature of these curves is the dip in response times for cluster size 8, and the smaller dips for cluster size 4. These dips are a direct consequence of the scheduling policy used in the experiment, which places the processes in a way that minimizes the number of clusters spanned. Consequently, contention within a cluster increases as processes are added until the cluster is full, at which point contention is maximized. The next process is added to a new cluster that is otherwise idle, so the response time of the new process is a minimum. Since the response times plotted are averages across all processors, the lone process has the effect of pulling the average down, thus causing the dip seen in the figures. As further processes are added to the new cluster, contention effects begin to increase the individual response times, but there are more processes to weight the average. This is why the dip for cluster size 8 has a minimum at 12 processors. As the number of processors is further increased, contention increases until at 16 processors, when both clusters are full, the response time returns to the saturated value obtained for the single cluster of 8 processors. These same arguments apply to the dips seen at 6, 10, and 14 processors for the cluster size 4 curve, and clearly shows how bandwidth increases as more clusters are added.

One will note that the single processor page fault times of Figure 7.2 steadily increase as the cluster size increases — from 142 μsec for cluster size 1 to 179 μsec for cluster size 16. The increase is due to NUMA effects within the larger clusters. The data structures of a cluster are shared by all processors within the cluster, and are distributed evenly across the memories of the cluster to balance the average memory demand. As a result the probability of having to access data from a remote processor (or station) increases with the cluster size, and consequently the average access time to memory increases.

It is interesting to consider what would happen if a different scheduling policy were used, for example, scheduling first across clusters and then within them. Under this policy, response times remain constant (at their minimum value) as new processes are added to idle clusters, until all processes have been assigned a process. When the number of processes exceeds the number of clusters, those clusters with multiple processes start to become contended, and the weighted averaging starts to bring the response times up. For extreme cases, the response times for the schedule-across policy are the same as for the schedule-within policy discussed earlier. These cases include sequential tasks and parallel programs that require all the processors, and operating system configurations where there is one cluster per processor or where a single cluster spans all the processors in the system.

To investigate the scalability of independent page faults, Figure 7.3(b) plots "Normal-

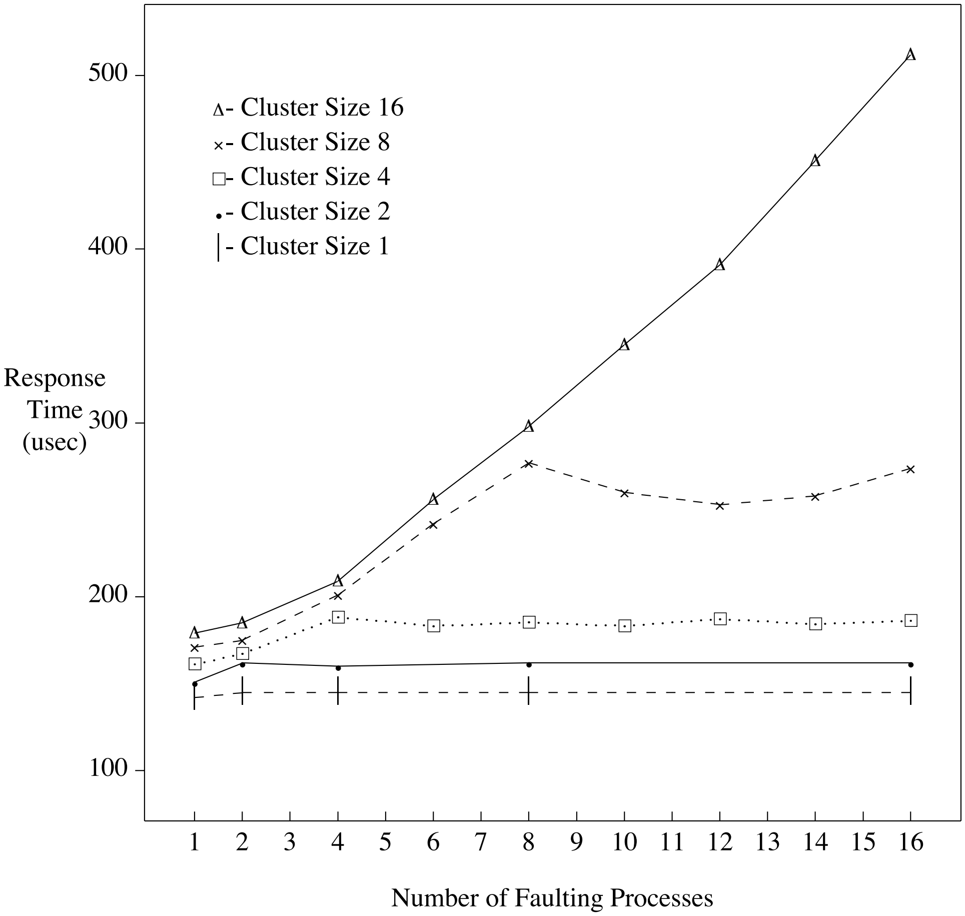

Figure 7.2: The response time versus number of faulting processes for simultaneous independent page faults. Each curve is for a different cluster configuration.

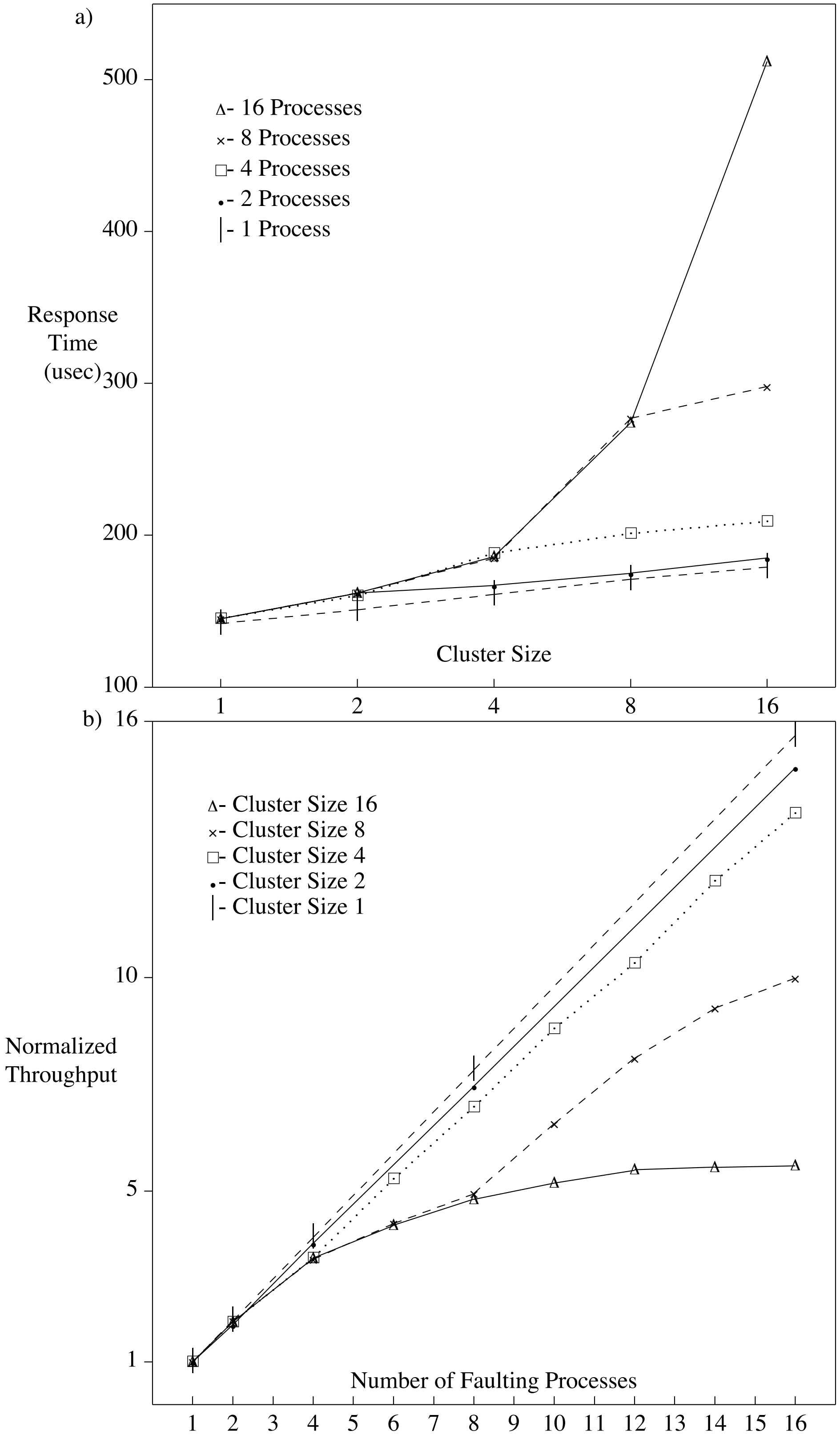

Figure 7.3: Performance of simultaneous independent page faults. a) Response time versus cluster size for different numbers of faulting processes. b) Normalized throughput versus number of faulting processes for different cluster configurations.

ized Throughput", which is derived from the response times of Figure 7.2 by the following relation:

$$X(n) = R(1)/(R(n)/n) = nR(1)/R(n) \tag{7.1}$$

where $R(1)$ is the response time for a single process, n is the number of processes, and $R(n)$ is the response time per page fault when n processes are running simultaneously. The figure clearly shows that this test favors small cluster sizes, which exhibit close to linear throughput increases (15.7 for 16 processes on cluster size 1). This behavior can be anticipated because the faults are to independent pages. The small cluster sizes benefit from several factors. First, the lazy replication mechanism for regions means that each cluster only contains the regions it is actively referencing, which reduces the search time. Second, the page caches only contain the pages accessed locally by the cluster, which reduces hash table conflicts and search times for these structures as well. Finally, but perhaps most important, lock contention is reduced on small cluster sizes. This is because equivalent data structures in different clusters are locked separately, effectively increasing the lock bandwidth. In addition, the reduced search times due to the automatic distribution of data structures when many clusters are used means that the length of critical sections is reduced.

7.2.2 Concurrent Faults to Shared Pages

The previous section measured the performance of simultaneous faults to independent pages. This section investigates the opposite end of the workload spectrum: the performance of simultaneous faults to shared pages. The program used to measure this performance is an extended version of the basic write fault program of Section 7.1.2. The program creates any number of children, or writer processes, and schedules each onto a different processor. A master process then reads a small number of pages (4 in the examples used here) to initialize them into the SR state, and then enters a barrier with the children. All the writer processes simultaneously write to these shared pages, causing a flood of page faults to the memory manager. The children then barrier again with the master, so that it can unmap the pages and start the next iteration.

The response time per fault is plotted against cluster size in Figure 7.4(a). Each curve in the figure joins the response times of equivalent numbers of writer processes for different cluster configurations. The values are plotted this way to show the performance trade-offs between contention and communication.

Intuitively, one expects this test to favor large cluster sizes, since a cross-cluster write fault involves two RPC calls and the overhead of creating a representative page. For the 16 process workload, Figure 7.4(a) shows that response times improve with increasing cluster size up to 8 processors per cluster, but then contention on the page cache causes a degradation in the performance when all 16 processors are configured as a single cluster. For the less extreme workloads, the minimum response time is found at the point where the cluster size matches the number of processes in the test. From this minimum, the curve slopes up sharply to the left because smaller cluster sizes must use remote communication

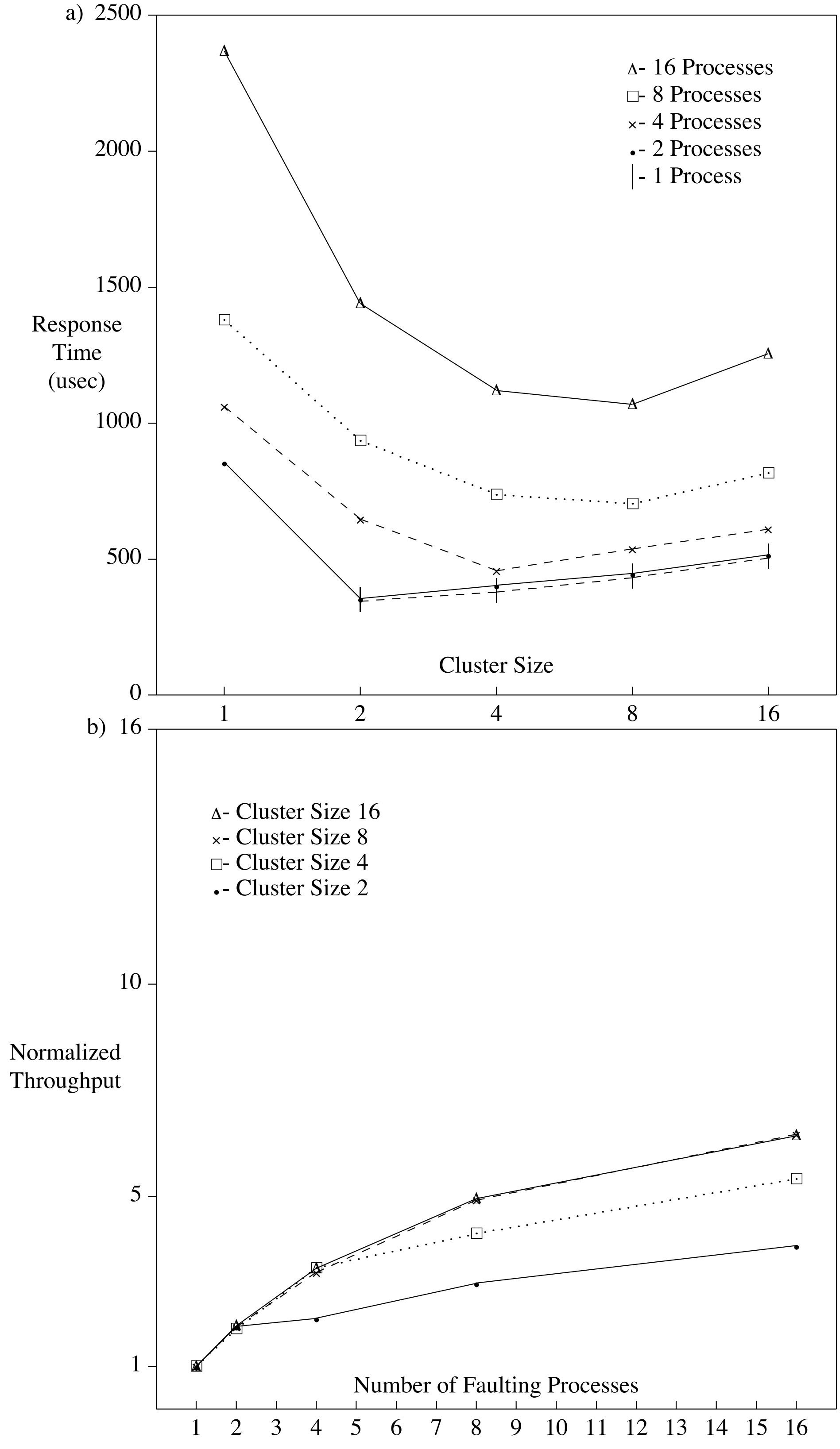

Figure 7.4: Performance of simultaneous faults to shared pages. a) Response time versus cluster size for different numbers of faulting processes. b) Normalized throughput versus number of faulting processes for different cluster configurations.

to complete the fault. To the right of the minimum, the slope of the curve is more gradual, and is due to the same NUMA effects that were observed for the independent page fault experiment of the previous section. It is certain that these are NUMA effects because all the data structures, including locks, are identical within all cluster sizes larger than or equal to the number of processes used in the test.

The normalized throughput for this example is given in Figure 7.4(b). The figure shows that throughput for faults to shared pages is much less than for faults to independent pages, with a peak of 6.5 for 16 processes on 2 clusters of 8 processors each. This result is not surprising, since the object of the test is to saturate a single resource: the shared pages that are the targets of the simultaneous faults. These findings seem to support the intuition that HECTOR/HURRICANE is best suited to medium or coarse-grained parallelism.

It is again interesting to consider the behavior of the system if the schedule-across policy is used instead of the schedule-within policy used for the experiments. As before, both policies yield identical response times for the four extremes: a single process or cluster size 1; and 16 processes or cluster size 16. For intermediate cluster sizes, the response time for 2 processes is significantly higher than the response time for the single process case, because the second process is scheduled to a different cluster and must therefore incur the cost of a cross-cluster write fault. As more processes are added to new clusters, the response times increase slightly due to RPC contention on the cluster containing the shared page. When the number of processes first exceeds the number of clusters, there is little change in response time, primarily because the overhead is dominated by the cost of the remote communication already present. As the clusters become full, the local page cache contention begins to dominate.

These observations, together with the consideration of scheduling effects on independent faults discussed in the previous section, suggest that the appropriate system-wide scheduling policy is to use the schedule-across policy for sequential tasks, and the schedule-within policy for the processes of parallel tasks. This hybrid policy minimizes the response times for the independent faults of separate sequential tasks, and minimizes the communication overhead for the shared faults of parallel tasks.

Multiprocessors like HECTOR are sometimes termed *weakly* NUMA, because the access ratio between local and remote memory is relatively small. For example, in the absence of contention, memory local to a HECTOR processor can be accessed in 10 cycles, while accesses to memory on other processors in the same station take only 4 cycles longer. In our particular configuration, accesses to memory on a different station take only 8 cycles longer than accesses to local memory. To investigate the effects of remote access latency, an artificial delay of 16 cycles was added to the ring in the HECTOR prototype. As a result accesses to a different station take 24 cycles longer than accesses to local memory. Figure 7.5 shows the results of the concurrent faults experiment with this ring delay. The experiment was run with 16 processes, and the cluster size was varied from 1 processor per cluster, to 16 processors per cluster.

The figure shows that the increased access latency does have an effect, particularly on the large and small cluster configurations. When the system is configured with cluster

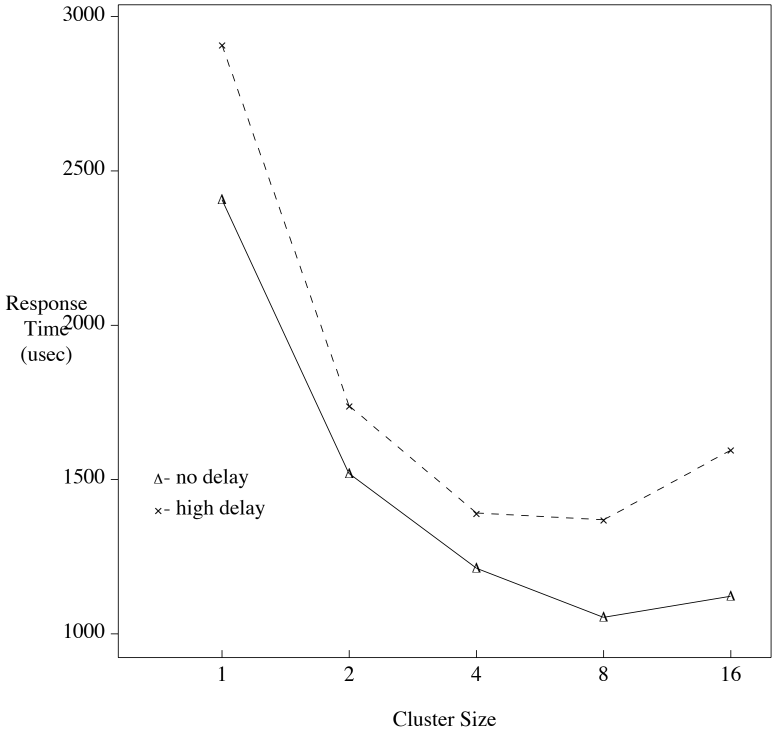

Figure 7.5: Performance of simultaneous faults to shared pages when the access latency to remote memory is changed.

size 16, all processors access the same data structures in a tightly-coupled fashion. The response times for this case degrade because the shared data structures have a relatively high probability of being on a remote station, and are therefore more costly to access. The response times for cluster size 1 also degrade more quickly than for intermediate cluster sizes. This is because there is more cross-cluster communication, so that the probability that the communication is to a cluster on a remote station is higher.

7.2.3 Unmap

This section examines the behavior of the unmap operation for compound requests. The example request removes the translations to a single physical page that is mapped by multiple processes. We began with the simple unmap program of Section 7.1.3, and extended it to allow any number of processes to execute on different processors. A master process first binds a shared region of 256 virtual pages. Each process then reads the first byte of each page in the region, so that all the pages become read shared by all the processes. The master then unmaps the pages of the region and the sequence is repeated.

Figure 7.6(a) shows the response times obtained for varying numbers of processes and cluster configurations. Since there is no contention from other processes while the region is unmapped, the experiment measures the parallelism of the unmap operation itself. To perform the unmap, asynchronous RPCs are used within and across clusters to form a

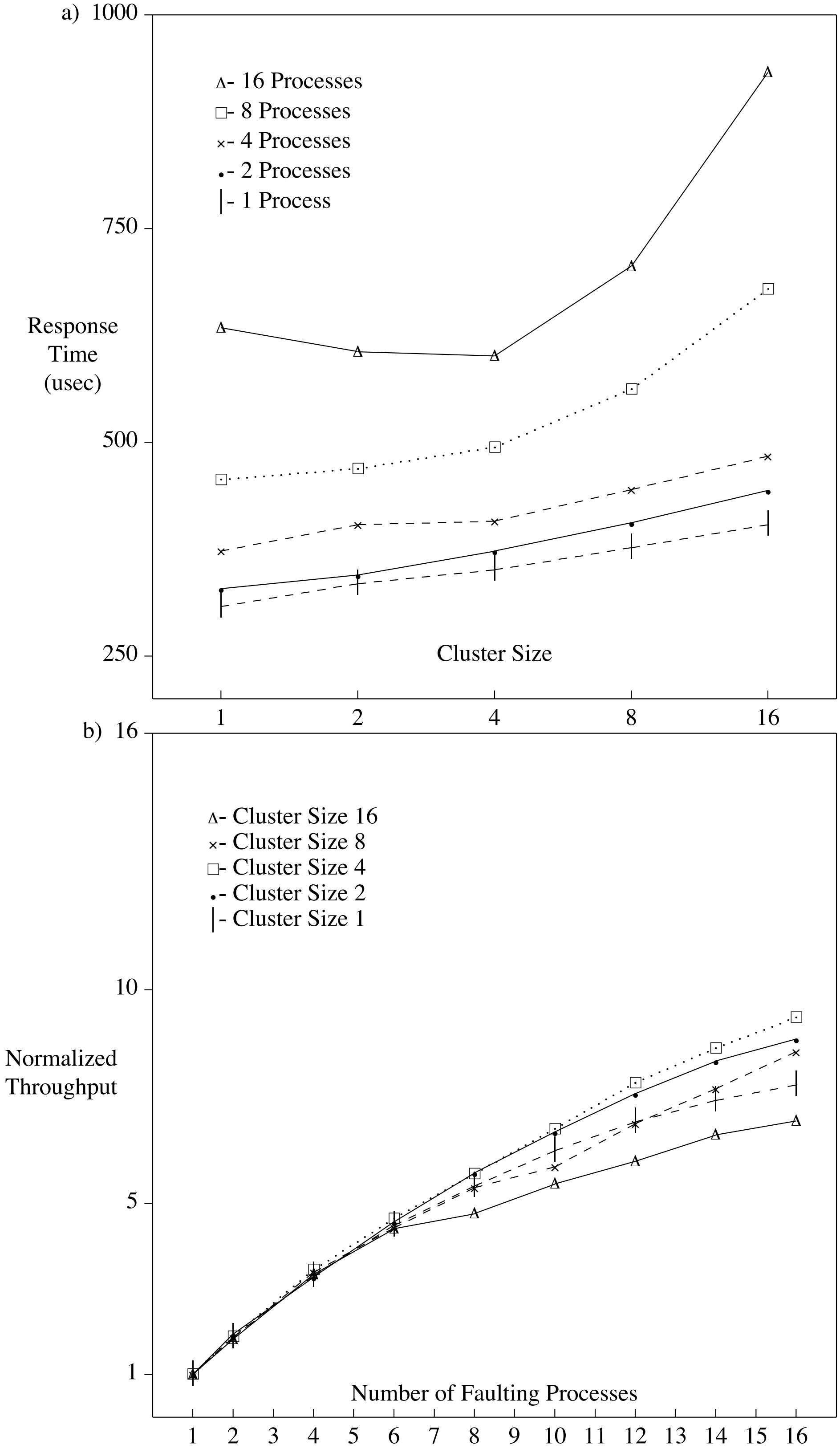

Figure 7.6: Performance of unmap for compound requests. a) Response time versus cluster size for different numbers of faulting processes. b) Normalized throughput versus number of faulting processes for different cluster configurations.

two-level spanning tree of the system (see Section 6.6).

Figure 7.6(a) shows the effectiveness of this approach, particularly for the 16 process workload. For this workload, concurrency is maximized for cluster size 4, which exhibits the lowest response time. The slopes of the curves in the figure indicate that the cross-cluster parallelism has a larger benefit than the parallelism obtained within the clusters. This is because the cross-cluster asynchronous RPCs permit concurrent searches of local data structures, while large cluster sizes can only benefit from parallel cache operations, which are a much smaller part of the total unmap time.

Figure 7.6(b) shows that the unmap operation appears to scale reasonably well, achieving a peak relative throughput of 9.34 for 16 processes on a 4 cluster configuration. These values indicate that the bandwidth of the unmap operation is increasing with system size.

7.3 Application Performance

Previous sections have examined the performance bounds of the system by using synthetic stress tests to establish throughput limits. These synthetic programs are useful because they allow control of experimental parameters, but they do not necessarily represent the behavior of real programs. This section examines the performance of three parallel applications as a function of cluster size. The applications are: SOR, a partial differential equation solver; matrix multiply; and 2D-FFT, which calculates the fourier transform of a two-dimensional array of data. All three programs are of the data-parallel, or SPMD[3], class of applications, which means that each process executes the same computational kernel on a different portion of the data space. However, the data access patterns of the applications are very different, so that each stresses a different aspect of the memory management sub-system.

All the tests are run with 16 processors; only the data set sizes and cluster configurations are varied. As for the rest of the experiments, the system is booted in full multi-user mode, so that all system servers are active but the system is otherwise idle. Unlike previous experiments, the response times reported in this section are for the entire program, which includes the time to create, schedule and destroy the worker processes, as well as the time to initialize any data structures. As a result, the times reported represent more than the overhead due to memory management alone; in particular, they include the overhead of the HURRICANE kernel. However, it is difficult to separate memory management from kernel overhead for real applications, and the response times still demonstrate the effects of clustering. In addition, measuring the time for the entire program permits a fair speed-up comparison, which will be reported for each application.

[3] SPMD stands for Single Program Multiple Data, and refers to the class of programs where all processors execute identical computational kernels, but on different parts of the data set.

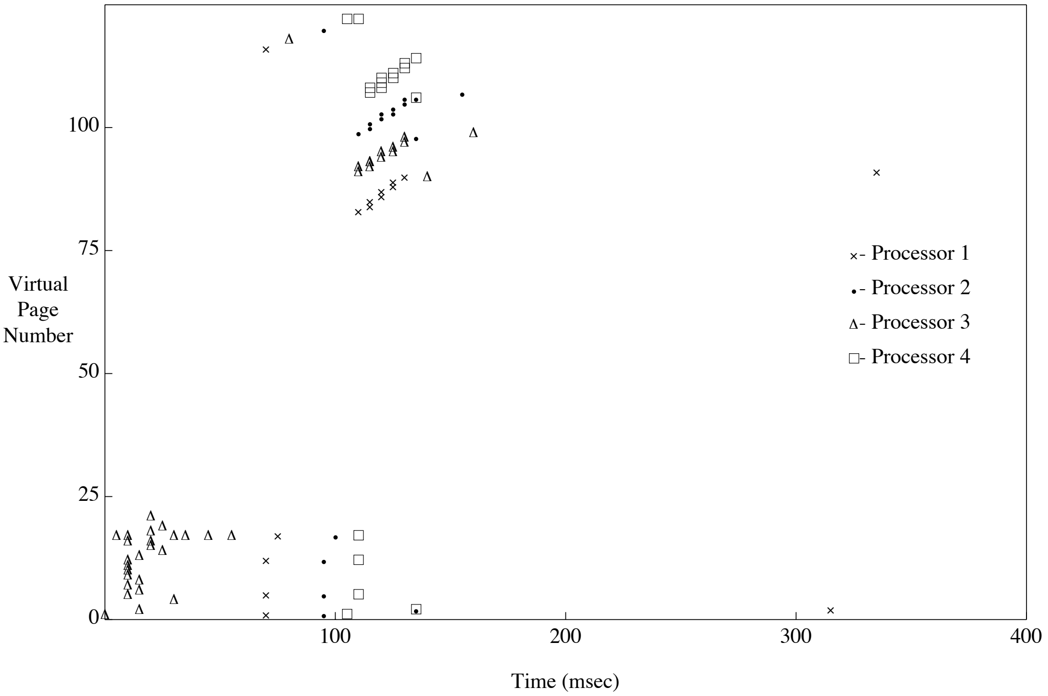

Figure 7.7: The page access and coherence faults of the SOR program, as recorded on the Hector multiprocessor.

7.3.1 Partial Differential Equations

The SOR application uses the method of simultaneous over-relaxation [52] to solve Laplace's equation for a two-dimensional data set. This method starts from a set of initial, or boundary conditions and iterates over the data space until the solution converges to a steady-state. The particular example used finds the temperature distribution in an infinitely long square rod whose edges are held at a fixed value. This example was chosen because it has a closed form solution, which allows the verification of the solution obtained. To solve this problem in parallel, one worker process is assigned to a separate processor, and each worker process iterates over an equal sized strip of the two-dimensional array. The data in each strip is accessed by a single process except along the common edges, where the data is shared by the processes in adjacent strips.

Figure 7.7 shows the page faults generated by a run of the program using 4 workers on a double-precision data set size of 128×128. The vertical axis shows the virtual page number (virtual address divided by the page size) of the page that was accessed to cause the fault. The horizontal axis shows the time at which each page fault was serviced, measured to the nearest 5 millisecond interval. Because the figure plots faults as a function of the process that caused them, there may be several entries for a given page number. Multiple faults can occur if the page changes state, for example from SRW to MRW, or if the page is unmapped and then reaccessed.

The scatter plot shown can be interpreted as follows. The faults in page numbers

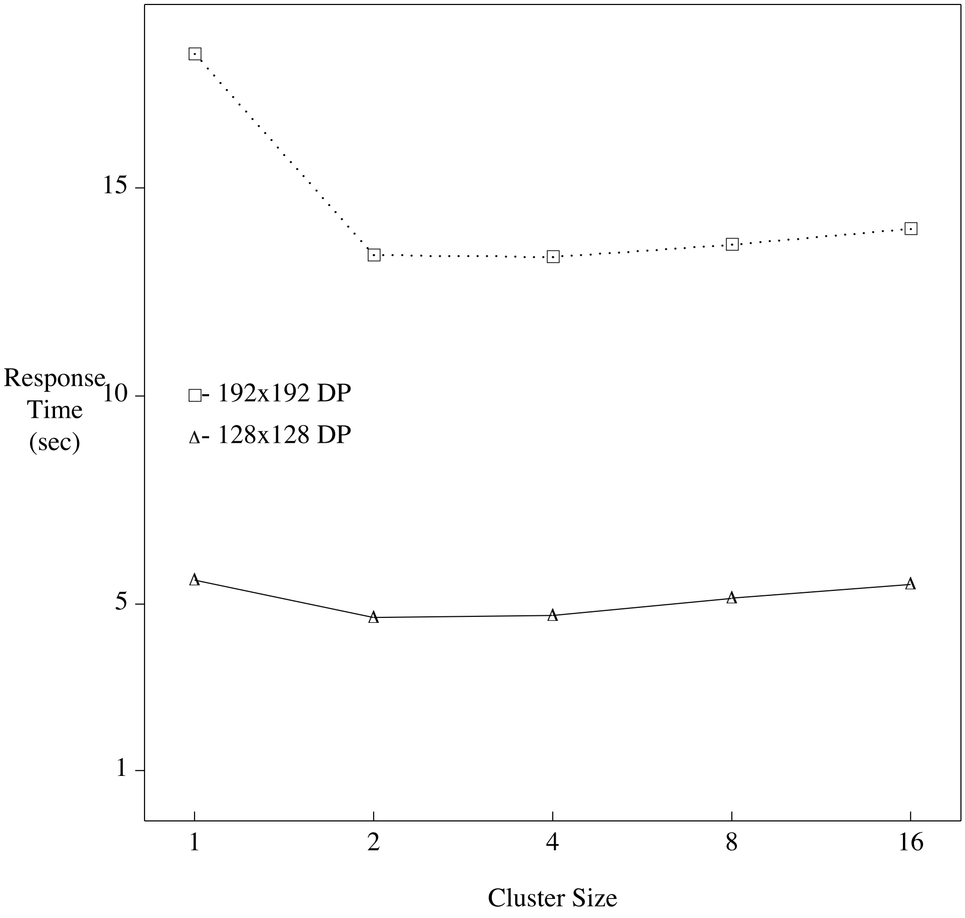

Figure 7.8: The response time of the SOR program for different data set sizes and cluster configurations.

below 25 are in the text and initialized data sections of the program. The program begins with a master (on processor 3) reading command line arguments and initializing the data for the forthcoming computation. At time 70 msec the children start, and immediately fault on their respective code and stack pages. The worker's stack pages are located at the highest page numbers in the figure. At time 110 msec the computation begins, as each worker sequentially accesses the pages in its designated strip of data. There are two faults on each page: the first is the initial access as the workers read the boundary values of the data set; the second follows almost immediately as the first computed value is written back to the array. At time 135 msec , the workers have accessed all the pages in their allocated data strip, and try to complete the first iteration by accessing the data of the adjacent worker. Since this adjacent page has already been initialized to the SRW state on a different processor, a coherence fault is generated to move the page into the MRW state, which allows the page to be shared safely by both workers. At this time, all processes have completely established their working sets, so there are no more faults until the computation completes. There are two key observations to be made about the data access patterns exhibited by the program: simultaneous page faults are primarily to the independent pages of each strip; but the pages along common edges are shared.

Figure 7.8 shows the response time as a function of cluster size for the SOR program using 16 processors on data set sizes of 128 × 128 and 192 × 192. The execution time for the large data set was 13.3 seconds when the system was configured with cluster size 4; this is a speed-up of about 13 over the execution time for a single worker. Both curves exhibit decreasing response times as the size of a cluster is reduced from 16 to 4 processors. At cluster size 2 the response times level out, and then increase when the cluster size is further reduced to 1 processor per cluster. Note that the increase in response time is more pronounced for the large data set, although the virtual resource and process management overhead is the same. One must therefore conclude that the increase is primarily due to memory effects.

Because the page faults are primarily to independent pages, one would expect that performance would improve as the cluster size is decreased (see Figure 7.3(a)). However, the observed decerease in response time as the cluster size decreases is much less than if the faults were to independent pages alone. To examine the behavior more closely, consider the data distribution of the array. For 16 processors, the 128 × 128 double-precision data is decomposed into strips of 2 pages each; for the 192 × 192 array, each worker is allotted a strip of 4.5 pages. In both cases, edge pages are shared by two workers, which means that two pages per strip are in the MRW state. MRW faults are always more expensive that SRW faults, and cross-cluster faults are even more expensive. As the cluster size decreases, the proportion of cross-cluster faults increases. Thus, for the small data set and 1 processor per cluster, each cluster has 2 pages for the single worker, plus 2 representative pages for the shared edge pages. For cluster size 2, there are still 2 representatives per cluster, but there are now 4 local pages because there are 2 workers on the cluster. The increase in cross-cluster faults that results as the cluster size is reduced is the primary reason for the flattening of the curve.

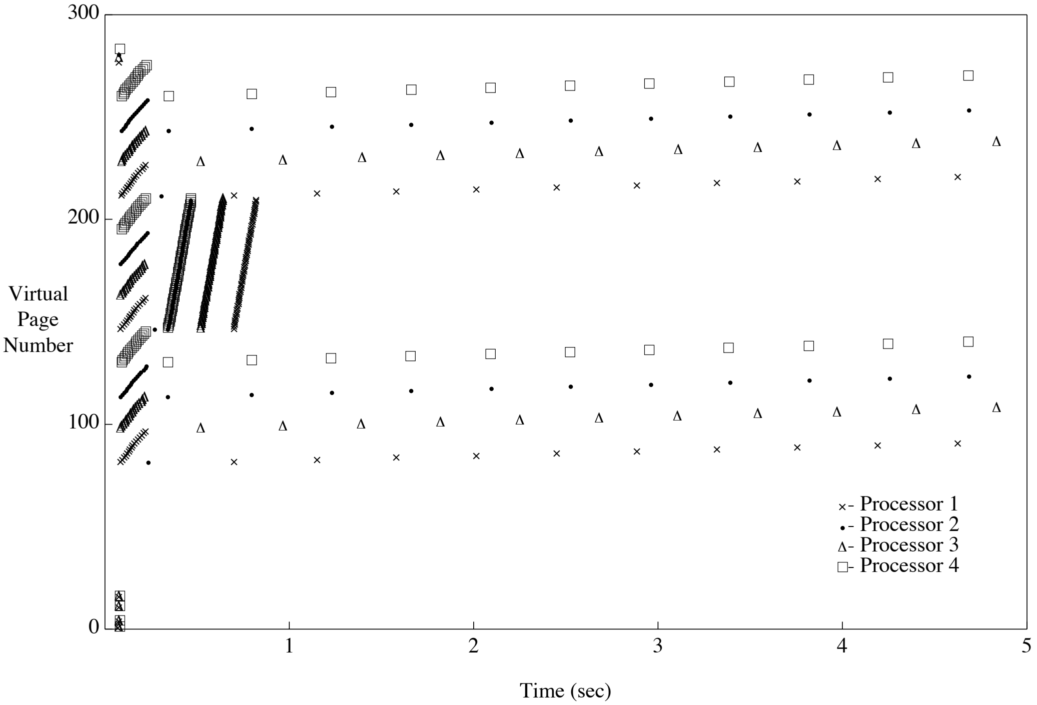

Figure 7.9: The page access and coherence faults of the Matrix Multiply program, as recorded on the HECTOR multiprocessor.

7.3.2 Matrix Multiply

The matrix multiply program solves the matrix equation $C = AB$, which has the following computational kernel:

```
do i=1 to N
  do j=1 to N
    do k=1 to N
      c[i,k] = c[i,k] + a[i,j]*b[j,k]
```

To solve this problem in parallel, each worker process executes the above kernel for an equal sized strip of the C matrix. The A and B matrices are read but not written, so they can be completely cached. The elements of C are written by a single process, so they should be cacheable as well. However, if the elements of two adjacent strips happen to be co-located on a common page, then that page will appear write-shared to the system, and is therefore uncached. This is an example of false sharing.

Page placement policies are very important in improving access locality for this program. To calculate the results for a strip of C, each process must access all the elements of C contained in the strip, the elements of the same strip of A, and the entire B matrix. To maximize locality for this access pattern, the A and C arrays are bound with a first-hit policy, which means the physical pages are allocated on the processor that first accesses them. Since B is accessed by all processors, the region containing this array is bound round-robin, which means the physical pages are cycled across the memories of the processors to balance the access load.

Figure 7.9 shows the page faults generated by a run of the matrix multiply program with four processes and a single precision array size of 256 × 256. The program begins by initializing the three arrays in parallel. This phase is seen as the short diagonal series of page faults by each processor. Besides initialization, this phase serves the important function of placing the data pages as discussed above. Since the pages of A and B are written to, their coherence state is be reset by the master to allow the processors to cache the data during the actual computation.

The computation starts at time 250 msec, with each worker moving through its strip of C and corresponding strip of A. The figure shows the faults along the rows of the A and C matrices as slowly increasing diagonals for each worker. In the figure, the C matrix is bound at the lowest virtual address, followed by B and then A. Since the entire B matrix is needed by all processes for each row of C, the faults to this array are concentrated at the beginning of the calculation. This aspect of the program stresses the capability of the memory manager to handle multiple simultaneous read faults to shared pages.

Figure 7.10 shows the response times for the matrix multiply program on different cluster configurations and with different data set sizes. Although the response time curves show U-shapes similar to the SOR program, the difference in the access patterns of the two programs indicate that the underlying cost contributions must also be different.

The default policy for read-only data is to share pages within clusters and replicate them across clusters. This policy applies to the B matrix, which is accessed in its entirety by all processors, and therefore replicated across all the clusters. The result is a strong improvement in locality and subsequent decrease in remote accesses, which benefits intermediate sized clusters the most. For large cluster sizes, there is not enough replication to improve the locality significantly, and for small clusters, the advantages are offset by the overhead of the replication itself.

The reader will notice that the data point is missing for the double-precision 384×384 data set on cluster 1. This is because the system attempts to replicate B onto every processor, which causes the system to run out of memory. However, even if page-out or a remote mapping policy were employed to allow the program to complete, the overhead of the policy would degrade the response times substantially.

7.3.3 Two-Dimensional FFT

The 2D-FFT program computes the forward and reverse transform of a two-dimensional array of data. The algorithm proceeds by performing a one-dimensional transform first along each row of the array, and then along each of its columns. The one-dimensional transform is computed with a fast fourier transform (FFT) algorithm, which requires $O(logN)$ time, where N is the number of points in the data set. This algorithm exploits symmetry in the calculation, which is dominated by sine and cosine operations. To improve the performance of the two-dimensional calculation, the sine and cosine values are pre-computed and saved

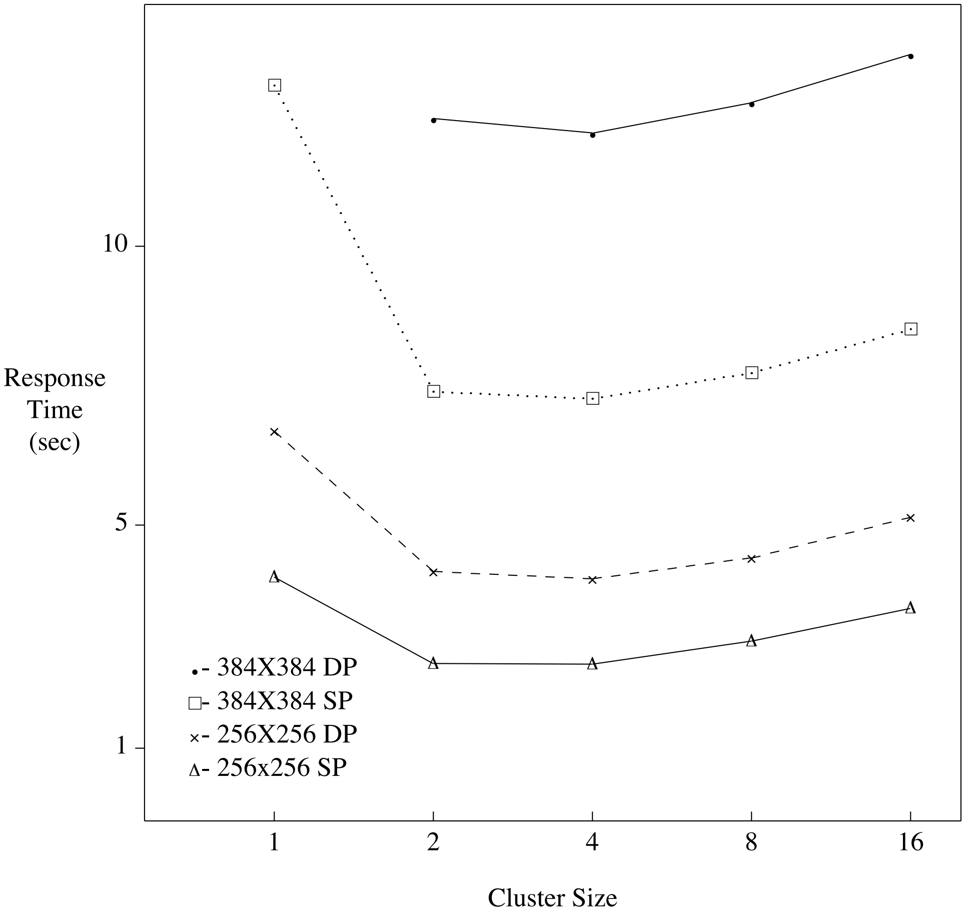

Figure 7.10: The response time of the Matrix Multiply program for different data set sizes and cluster configurations.

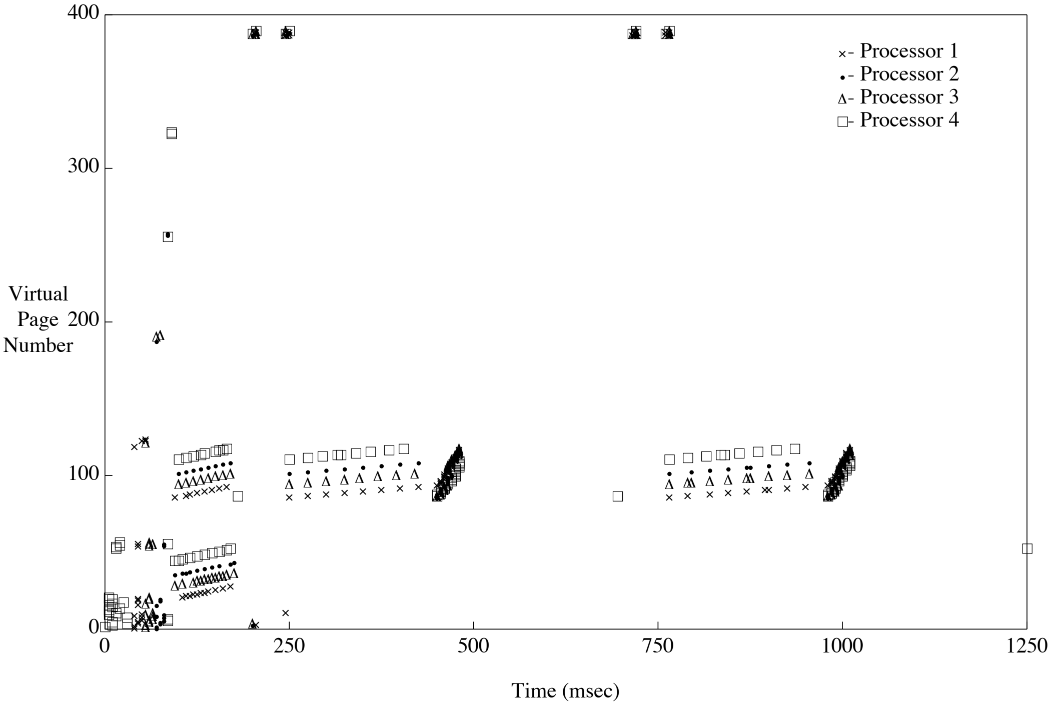

Figure 7.11: The page access and coherence faults of the 2D-FFT program, as recorded on the HECTOR multiprocessor.

in a look-up table.

Like the previous two applications, the parallel solution to this problem assigns equal sized strips of rows and columns to each worker process. The workers compute the row transforms in parallel, then barrier before computing the column transforms. Since the C language stores arrays in row-major order, the columns are loaded into local vectors before transforming them; this allows a single 1D-FFT routine to suffice for all calculations.

Figure 7.11 shows the page faults generated by a run of the 2D-FFT program with four workers on a single-precision data set size of 128 × 128. The program starts with the usual cluster of text and stack faults as the workers are created and initialized. At time 100 msec, the array is initialized, in parallel, to random values. A copy of the data set is also made at this time so that the results of the calculation can be checked. The forward transform begins at time 200 msec with the workers initializing the sine and cosine look-up tables. These tables occupy the highest page numbers in the figure. Since the tables have been written for initialization but are used in a read-only manner, the coherence state of the tables is reset so that they can be cached for the remainder of the computation. At time 250 msec the row transforms begin, and can be identified by the series of page faults at gradually increasing virtual page numbers. The workers barrier at time 450 msec to begin the column transform. Since the pages of the array had been initialized to the SRW state, the load of the columns causes an intense flurry of coherence faults as the pages move to the MRW state. This aspect of the program stresses the memory managers ability

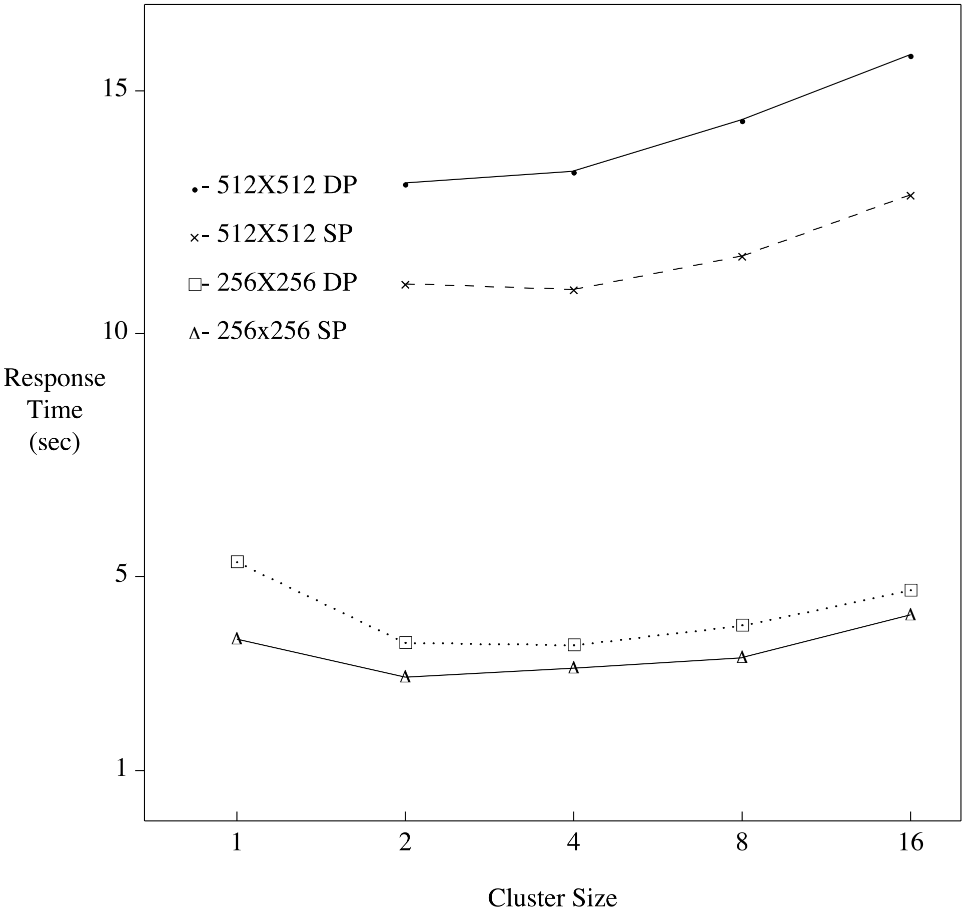

Figure 7.12: The response time of the 2D FFT program for different data set sizes and cluster configurations.

to handle simultaneous faults to shared pages. There are no more faults for the remainder of the forward transform, since the entire working set is by now well established. The transform completes at time 700 msec, when the workers barrier with the master. The remainder of the faults in the figure are the results of the reverse transform, which shows the same access pattern as exhibited by the forward part of the program.

Figure 7.12 shows the response times for the 2D-FFT program for different data set sizes and cluster configurations. Again, intermediate cluster sizes show the best performance, with cluster size 2 achieving a speed-up of 9.3 on the 512×512 double-precision data set. The complex access patterns of this application make it difficult to attribute the performance difference to any one aspect of the system. However, in tuning the system for better performance, we found that times on intermediate cluster sizes were improved by the addition of asynchronous unmap operations (they used to be sequential), and that large cluster sizes benefited from finer grained locking in the page cache.

Chapter 8

Conclusion

8.1 Overview of the Dissertation

This dissertation has focused on the structuring of memory management for scalable performance. We began by considering the properties that an operating system should possess if the throughput is to increase with the number of processors. These considerations led to a set of design guidelines and a new framework for structuring systems, called Hierarchical Symmetric Multiprocessing. The HSM architecture promotes scalability by distributing and replicating service capabilities and data structures across tightly-coupled groups of processors called clusters.

Chapters 5 and 6 described how the principles of hierarchical symmetric multiprocessing are applied to the Hurricane memory manager. The data structures of the memory manager are directly accessible by all processors within a cluster and are shared as needed across clusters. The integrity of the data structures is maintained by synchronization protocols within and across clusters. Cross-cluster consistency of virtual resources is maintained by directing all requests through the home cluster of the address space. A directory structure is used as the synchronization point for cross-cluster operations on physical pages. Clusters cooperate through three types of communication mechanisms: shared memory is used to access directory entries; RPC is used for demand-driven operations; and message-passing is used by the server processes of the memory manager.

The effectiveness of the HSM architecture was illustrated in Chapter 7. The experiments showed that the trade-offs between contention and communication could be balanced by choosing the cluster size correctly. In particular, several applications with different memory access behaviors were shown to perform better using an intermediate cluster size than using either fully distributed or fully shared configurations. For example, for the SOR, 2D-FFT, and Matrix Multiply applications, cluster size 4 and 2 always resulted in better performance than cluster size 1.

8.2 Summary of Contributions

The effort to build a scalable memory manager has resulted in three main contributions: a set of sufficient conditions and guidelines for a scalable operating system; an architectural framework for designing scalable systems, called hierarchical symmetric multiprocessing; and a prototype implementation of the architecture, called HURRICANE, which demonstrates the viability of the approach. This section reviews these contributions.

8.2.1 Conditions for Scalability

From a consideration of the properties of the fundamental metrics of computer performance, we have been able to develop the following set of conditions sufficient for an operating system to scale:

1. The time at a particular resource devoted to servicing a particular request must be bounded by a constant independent of p, the number of processors in the system.

2. The number of resources available to service a particular class of request must increase proportional to p.

3. The system must be balanced in its service capabilities.

4. The servicing of individual requests must be localized and independent.

These criteria can be translated into a set of design guidelines that can be used as a basis for system structuring:

1. The operating system must preserve the parallelism of the applications it supports. This is achieved by increasing the operating system resources as a function of p, and by providing balanced and distributed service capabilities.

2. The operating system must bound its service time and space overhead. Bounded service times means the system must use data structures and search techniques that are independent of p. Bounded space restricts the system to use data structures whose space costs grow proportional to the physical resources of the underlying machine.

3. The operating system must preserve the locality of the applications. Locality can be preserved a) by properly choosing and placing data structures within the operating system, b) by directing application requests to nearby service points, and c) by enacting policies that preserve locality in the applications' memory accesses.

8.2.2 Hierarchical Symmetric Multiprocessing

Hierarchical symmetric multiprocessing is a new architecture for operating system structuring. The goal of HSM is to provide an integrated and formal approach to operating system structuring without compromising performance. The architecture is based on observations of the characteristics of existing systems. Tightly-coupled systems can achieve good performance through shared data structures and fine-grained communication, but cannot scale. Loosely-coupled systems can scale through distributed service capability and replicated data structures, but the cost of remote communication is typically high.

A hierarchical symmetric multiprocessing system is based on the notion of clusters, which are composed of groups of neighboring processors that share data structures in a tightly-coupled way. Each cluster supports the complete functionality of a small-scale symmetric multiprocessing operating system. Multiple clusters cooperate and communicate in a loosely-coupled fashion to give applications an integrated and consistent view of a single large system. The advantages of hierarchical symmetric multiprocessing can be summarized as follows:

- Hierarchical symmetric multiprocessing provides a framework for exploiting locality, since system service points and data structures are replicated across the system. Performance is maintained because data structures are local to where they are accessed, and because data structures are shared primarily by processors within a cluster. Large-scale applications are scheduled across multiple clusters, and can benefit from the concurrency afforded through replicated system services.

- Hierarchical symmetric multiprocessing enhances portability by allowing performance tuning to different architectures. The appropriate cluster size for a given architecture is affected by several factors, including the local-remote memory access ratio, the hardware cache size and coherence support, and the network topology. HSM allows the cluster size to be chosen to match these architectural parameters.

  This same reasoning can be applied to tuning for different workloads on a single system. By choosing the cluster size correctly, the trade-offs between contention and communication overhead can be balanced to maximize performance.

- Finally, hierarchical symmetric multiprocessing simplifies lock structuring issues, which can lead to improved performance and scalability. Because contention for a lock is limited to the number of processors in a cluster, locks within a cluster can be relatively coarse-grained and still achieve good performance. Further, the locks do not have to be restructured as the system grows, because the locks and the structures they protect are instantiated separately on each cluster of a large system.

8.2.3 Hurricane

Hurricane is a new operating system built to evaluate the viability of a hierarchical symmetric multiprocessing architecture. The memory manager was built from scratch so that it is free of the the bias and constraints imposed by an established environment. Through the demand replication and distribution of data and control structures, we were able to meet the design criteria of a HSM architecture. In particular, our focus on structure and mechanism has resulted in a flexible base that can be used to experiment with trade-offs in coupling, and which can be used as a platform for the development of higher level policies.

In the course of developing the HURRICANE memory manager, we have gained valuable experience in designing large systems, which has led to several novel structuring approaches. First, the copy-on-write mechanism of HURRICANE is not based on the traditional shadow trees of Mach and Chorus. The sub-region approach we use arises partly as a result of the different environment and workload of HURRICANE, but we also believe the approach is more scalable because the dependence on global data structures is reduced. Second, we have developed an asynchronous RPC protocol that permits parallelism to be exploited in the servicing of compound requests, such as unmapping multiple replicas of a physical page. Third, HURRICANE maintains page tables on a per processor basis. This approach permits exact book-keeping for advanced coherence policies, and simplifies the problem of TLB-consistency. HURRICANE is the first operating system we know of to maintain both cache and memory coherence as part of the memory manager. Our experience with directories has yielded interesting insights into structuring of directories for both locality and scalability. All four areas have been developed in the context of hierarchical symmetric multiprocessing on HURRICANE. It will be interesting to explore the applicability of these approaches on other systems and architectures.

8.3 Future Work

The implications of hierarchical symmetric multiprocessing structuring are far-reaching, and present many challenges. This section enumerates several such challenges within the memory manager itself, and across the system as a whole.

Development so far has focused on the structuring and mechanisms of the HURRICANE memory manager to allow scalability, and the current system provides a flexible test-bed for further experimentation. In particular, several of our current mechanisms warrant further attention. First, our current approach to supporting remote accesses through representative page descriptors must be re-evaluated. The number of representatives per cluster is currently a constant, and the constant can increase with cluster size because there is more memory available for the descriptors. Unfortunately, the probability of accessing pages remotely increases as the cluster size decreases, and the small cluster sizes have the fewest number of available representatives. However, we are hesitant to share page descriptors directly across clusters, because locality is lost and contention is increased.

A second interesting area for study is in the structuring and handling of RPC calls. To support clustering, we have found that each memory manager function has three internal interfaces: one for operations local to the cluster; another to serve as the RPC handler for operations remote to the cluster; and the third to invoke operations remote to the cluster. We feel that having three interfaces for each call is counter to good software engineering principles, and would like to investigate other structuring approaches.

The use of RPC for cross-cluster communication also raises issues in how the servicing of remote requests is balanced across the system. Currently, the RPC service load is balanced by having processor i of each cluster direct its RPC requests to processor i of the target cluster. If the target processor is busy at the time of the request (for example, it may be servicing a local page fault), the requesting processor waits until the target processor is free. We would like to investigate more dynamic load balancing approaches; an example would be to direct an RPC request to any non-busy processor in the target cluster.

While development so far has focused on the structuring and mechanism, there are a number of other area of memory management that should be addressed:

Memory Management Policies: The suitability of HURRICANE as a platform for higher level policies has been demonstrated through the example coherence policy used in the dissertation, but other, more sophisticated policies are possible and could perform better. The policies that govern the placement and replication/migration of pages must consider both the application access patterns and the architectural parameters [37]. Several groups have done initial policy studies [36, 17, 24]. However, their results tend to be specific to a particular architecture, so the generalization of their findings to a clustered environment, and the analysis of the scalability of the policies, is an interesting area of study [18].

Application-Level Control: Because the memory manager must support a broad mix of applications, its policies are general and are tuned to perform well for the common case. In addition, the mechanisms and policies are constrained to the protection granularity supported by the hardware. Applications may be able to improve their performance by overriding these defaults with their own memory management policies. Some possibilities for study in this area include:

- Application level cache coherence [50] - to allow finer control than the page size granularity supported by the memory manager.
- Weakly consistent systems [20] - to reduce the effects of false sharing.
- Application level page fault handling [66, 5] - to allow application control of page placement and mapping.

Page-out: Few researchers have investigated page replacement policies for NUMA architectures, although this topic raises several important issues. For example, instead of replacing or paging-out valid pages on clusters where available memory is low, pages can be "borrowed" from neighboring clusters with more available memory [35]. The opposite approach is also possible: instead of writing pages targeted for replacement to disk immediately, they could be migrated away to clusters with more available memory. Another possible solution is to balance memory demand by migrating processes to clusters with more available memory.

Besides memory management, the principles of hierarchical symmetric multiprocessing have also been applied to the HURRICANE kernel implementation. The next challenge is the issue of scalability within the I/O subsystem. We believe it will be beneficial to replicate and distribute the file blocks of individual files across a number of disks, which are themselves distributed across the system. One of the important differences between memory and files is that data in the latter is persistent, which means that a block placement decision has repercussions far beyond the lifetime of the program that wrote the data. Although the file blocks are cached by the memory manager, the persistence issue complicates the management of locality, because remotely accessed file blocks consume bus bandwidth during their transfer. The eventual solution to the problem could even involve scheduling decisions that place a program near to where its data is located.

While the dissertation has considered the application of HSM to operating systems, we believe the architecture is also well suited to the design of large-scale parallel programs. The cost trade-offs between contention and communication are largely the same for applications and operating systems, and the management of locality is key to the performance of both. An interesting question in application design, which also affects the operating system, is what to do when the degree of coupling that optimizes application performance is different from the cluster size of the system.

In the longer term, we would like to test our ideas on larger systems, and are particularly interested in extending the hierarchy of the system. For example, the single-level directory of the memory manager could be replaced with a hierarchical directory, to permit better locality of access and more concurrency. Also, clusters could themselves be grouped into super clusters. Processor load balancing is an obvious candidate for this hierarchical clustering. A high-level process manager schedules processes between super clusters, while lower-level managers schedule processes within a super cluster.

In conclusion, we are encouraged by the results obtained through the application of hierarchical symmetric multiprocessing to HURRICANE so far, and look forward to the challenges of the future.

Bibliography

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. "Generic Virtual Memory Management for Operating System Kernels". In *Proc. 12th ACM Symposium on Operating System Principles*, pages 123–136, Litchfield Park, Arizona, December 1989.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for Unix Development". In *Proc. 1986 Summer USENIX Conference*, pages 93–112, July 1986.

[3] I. Ahmad and A. Ghafoor. "Semi-distributed Load Balancing for Massively Parallel Multicomputer Systems". *IEEE Transactions on Software Engineering*, 17(10):987–1004, October 1991.

[4] George S. Almasi and Alan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Co., 390 Bridge Parkway, Redwood City, CA 94065, 1989.

[5] Andrew W. Appel and Kai Li. "Virtual Memory Primitives for User Programs". In *ASPLOS-IV Proceedings*, pages 96–107, Santa Clara, California, April 8–11 1991.

[6] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1986.

[7] Ramesh Balan and Kurt Gollhardt. "A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines". In *Summer '92 USENIX*, pages 107–115, San Antonio, TX, June 1992.

[8] Amnon Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. Computer Science RC 13220 (#59114), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1987.

[9] Amnon Barak and On G. Paradise. "MOS - Scaling up UNIX". In *Proc. USENIX Conference*, pages 414–418, December 1986.

[10] BBN Advanced Computers, Cambridge, MA. *Inside the Butterfly-Plus*, October 1987.

[11] John Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. Technical Report COMP TR89-98, Rice University, P.O. Box 1892, Houston, Texas 77251-1892, November 1989.

[12] Richard R. Billig, Stephen S. Corbin, and Russel L. Moore. "A fast backplane cluster heralds a 1000-MIPS computer". *Electronic Design*, July 1987.

[13] A. D. Birrell and B. J. Nelson. "Implementing Remote Procedure Calls". *ACM Transactions on Computer Systems*, 2(1), February 1984.

[14] David Black. "Translation Lookaside Buffer Consistency: A Software Approach". In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, 1989.

[15] David Black, Anoop Gupta, and Wolf-Dietrich Weber. "Competitive Management of Distributed Shared Memory". In *Spring COMPCON 89 Digest of Papers*, pages 184–190, 1989.

[16] David R. Blythe, Michael Stumm, and Ron Unrau. Hurricane Exception Architecture — Source Specification. Technical report, University of Toronto, Toronto, Ontario, Canada, August 1989.

[17] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. "Simple But Effective Techniques for NUMA Memory Management". *Operating Systems Review*, 23(5):19–31, 1989. Reprinted from *Proc. 12th ACM SOSP*, Dec. 1989, Litchfield Park, Arizona.

[18] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. "NUMA Policies and Their Relation to Memory Architecture". In *ASPLOS-IV Proceedings*, pages 212–221, Santa Clara, California, April 8–11 1991.

[19] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendell Square Research, Boston, February 1992.

[20] John B. Carter, John K. Bennett, and Willy Zwaenepoel. "Implementation and Performance of Munin". In *Proc. 13th ACM SOSP*, October 1991.

[21] David Chaiken, John Kubiatowics, and Anant Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme". In *ASPLOS-IV Proceedings*, pages 224–234, Santa Clara, California, April 8-11 1991. ACM.

[22] E. Chaves, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-Kernel Communication in a Shared-Memory Multiprocessor,. In *Second Symposium on Distributed and Multiprocessor Systems*, pages 105–116, Atlanta, Georgia, March 1991. Usenix. submitted to Computing SystemsP, April 1991.

[23] David R. Cheriton. “The V Distributed System”. *Communications of the ACM*, 31(3):314–333, March 1988.

[24] Alan L. Cox and Robert J. Fowler. “The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM”. *Operating Systems Review*, 23(5):32–44, 1989. Reprinted from *Proc. 12th ACM SOSP*, Dec. 1989, Litchfield Park, Arizona.

[25] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, Reading, Mass., 1984.

[26] P. J. Denning. “Thrashing: Its Causes and Prevention”. In *AFIPS Conf. Proc.*, pages 915–922, 1968.

[27] Peter J. Denning. “Working Sets Past and Present”. *IEEE Transactions on Software Engineering*, January 1980.

[28] Dror .G. Feitelson and Larry Rudolph. “Distributed Hierarchical Control for Parallel Processing”. *Computer*, 23(5):65–81, May 1990.

[29] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. *IEEE Transactions on Computers*, C-29(9):948–960, September 1972.

[30] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. “Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors”. In *ASPLOS-IV*, pages 245–257, Santa Clara, California, April 8-11 1991. ACM.

[31] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. “Scale and Performance in a Distributed File System”. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[32] David V. James, Anthony T Laudrie, Stein Gjessing, and Gurindar S. Sohi. “Distributed-Directory Scheme: Scalable Coherent Interface”. *Computer*, 23(6):74–77, June 1990.

[33] Orran Krieger, Michael Stumm, and Ron Unrau. “Exploiting the Advantages of Mapped Files for Stream I/O”. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 27–42, San Francisco, January 1992.

[34] Leslie Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[35] Richard P. LaRowe Jr. and Carla Schlatter Ellis. Dynamic page placement in a NUMA multiprocessor virtual memory system. Technical Report CS-1989-21, Dept. Computer Science, Duke University, Durham, NC 27706, October 1989.

[36] Richard P. LaRowe Jr. and Carla Schlatter Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical Report CS-1990-10, Dept. Computer Science, Duke University, Durham, NC 27706, April 1990.

[37] Richard P. LaRowe Jr., Carla Schlatter Ellis, and Laurence S. Kaplan. "Tuning NUMA Memory Management for Applications and Architectures". In *Proc. 13th ACM SOSP*, October 1991.

[38] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1984.

[39] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. "The Architecture of an Integrated Local Network". *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–856, November 1983.

[40] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[41] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. "The Stanford Dash Multiprocessor". *Computer*, 25(3):63–79, March 1992.

[42] Eliezer Levy and Abraham Silberschatz. "Distributed File Systems: Concepts and Examples". *ACM Computing Surveys*, 22(4):323–373, December 1990.

[43] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University Department of Computer Science, Cambridge, Massachusetts, September 1986.

[44] Tom Lovett and Shreekant Thakkar. "The Symmetry Multiprocessor System". In *Proc. 1988 International Conference on Parallel Processing*, pages 303–310. CRC Press, Inc., August 1988.

[45] Motorola Inc. *MC88100 RISC Microprocessor User's Manual*, 1988.

[46] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. "Amoeba: A Distributed Operating System for the 1990s". *Computer*, 23(5):44–53, May 1990.

[47] Michael N. Nelson and John K. Ousterhout. "Copy-on-write for Sprite". In *Proc. Summer USENIX '88 Conference*, pages 187–201, San Fransisco, California, June 1988.

[48] Daniel Nussbaum and Anant Agarwal. "Scalability of Parallel Machines". *Communications of the ACM*, 34(3):56–61, March 1991.

[49] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglas, Michael N. Nelson, and Brent B. Welch. "The Sprite Network Operating System". *Computer*, 21(2):23–36, February 1988.

[50] Susan Owicki and Anant Agarwal. "Evaluating the Performance of Software Cache Coherence". *ACM*, pages 230–242, 1989.

[51] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. "The IBM Research Parallel Processor Prototype". In *Proc. 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[52] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 510 North Avenue, New Rochelle, NY 10801, 1988.

[53] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures". In *Proc. 11th ACM Symposium on Operating System Principles*, pages 63–76, Austin, Texas, November 1987.

[54] Bryan S. Rosenburg. "Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors". *Operating Systems Review*, 23(5):137–146, 1989. Reprinted from *Proc. 12th ACM SOSP*, Dec. 1989, Litchfield Park, Arizona.

[55] Harjinder S. Sandhu, Benjamin Gamsa, and Songian Zhou. Region-oriented memory management in shared-memory multiprocessors. Technical Report CSRI-269, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1, April 1992.

[56] C. Scheurich and M. Dubois. "Dependency and Hazard Resolution in Multiprocessors". In *Proc. 14th International Symposium on Computer Architecture*, pages 234–243, Los Alamitos, California, 1987. IEEE CS Press. Order No. 776.

[57] Robert Sedgewick. *Algorithms in C.* Addison-Wesley Publishing Company, Inc., Reading, Mass, 1990.

[58] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples.* McGraw-Hill Book Company, 1982.

[59] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1987.

[60] M. Stumm. The design and implementation of a decentralized scheduling facility for a workstation cluster. In *Proc. IEEE Conf. on Computer Workstations*, pages 12–22, March 1988.

[61] Michael Stumm and Songian Zhou. "Algorithms Implementing Distributed Shared Memory". *Computer*, 23(5):54–64, May 1990.

[62] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, N.J., 07632, 1992.

[63] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr. "Firefly: A Multiprocessor Workstation". *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[64] Michael Y. Thompson, J.M. Barton, T.A. Jermoluk, and J.C. Wagner. "Translation Lookaside Buffer Synchronization in a Multiprocessor System". In *USENIX Winter Conference*, pages 297–302, Dallas, Texas, February 1988.

[65] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *Computer*, 24(1), January 1991.

[66] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System". In *Proc. 11th ACM Symposium on Operating Systems Principles*, Austin, Texas, November 1987.

[67] S. Zhou, M. Stumm, M. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):540–554, 1991.

[68] Songian Zhou and Tim Brecht. "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors". In *Sigmetrics 1991 Proceedings*, June 1991.