HIVE: OPERATING SYSTEM FAULT CONTAINMENT FOR SHARED-MEMORY MULTIPROCESSORS

John Chapin

Technical Report No. CSL-TR-97-712

July 1997

This research has been supported by DARPA contract DABT63-94-C-0054.
Author also acknowledges support from the Fannie and John Hertz Foundation.

Hive: Operating System Fault Containment for Shared-Memory Multiprocessors

John Chapin

Technical Report No. CSL-TR-97-712

July 1997

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, A-408
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

Reliability and scalability are major concerns when designing general-purpose operating systems for large-scale shared-memory multiprocessors. This dissertation describes Hive, an operating system with a novel kernel architecture that addresses these issues. Hive is structured as an internal distributed system of independent kernels called *cells*. This architecture improves reliability because a hardware or software error damages only one cell rather than the whole system. The architecture improves scalability because few kernel resources are shared by processes running on different cells. The Hive prototype is a complete implementation of UNIX SVR4 and is targeted to run on the Stanford FLASH multiprocessor.

The research described in the dissertation makes three primary contributions: (1) it demonstrates that distributed system mechanisms can be used to provide fault containment inside a shared-memory multiprocessor; (2) it provides a specification for a set of hardware features, implemented in the Stanford FLASH, that are sufficient to support fault containment; and (3) it demonstrates how to take advantage of shared-memory hardware across cell boundaries at both application and kernel levels while preserving fault containment. The dissertation also analyzes the architectural and performance tradeoffs of multicellular kernels.

Fault injection experiments conducted using the SimOS machine simulator demonstrate the reliability of the Hive prototype. Studies using both general-purpose and scientific workloads illustrate the performance tradeoffs of the multicellular kernel architecture.

Key words and phrases: Hive, Stanford FLASH, UNIX SVR4, multicellular architecture, operating system reliability, operating system scalability, fault containment, fault tolerance, shared-memory multiprocessors, CC-NUMA multiprocessors, distributed systems.

To my parents.

Acknowledgments

Building an operating system is an immense task. The challenge is even more daunting when the system architecture and the machine on which it is to run are being developed at the same time. Hive would not be operational today without the hard work of many people.

First I would like to thank my advisor, Mendel Rosenblum, for his inspiration and guidance. The other faculty members of the FLASH project, Anoop Gupta, Mark Horowitz, and John Hennessy, contributed valuable advice as the system developed. My committee, consisting of Mendel Rosenblum, Anoop Gupta, and Mary Baker, spent substantial time commenting on the dissertation and greatly improved its quality.

I would also like to acknowledge the huge design and implementation effort made by the other members of the Hive team. Dan Teodosiu, Scott Devine, Kinshuk Govil, Tirthankar Lahiri, and Yasuichiro Izumi have collectively spent about eight man-years on the system. Compared to industrial operating system development efforts we are a small team; the level of functionality reached by Hive testifies to the superb job done by everyone involved.

No experiments on Hive would have been possible without the SimOS machine simulator. Steve Herrod, Emmett Witchel, Ed Bugnion, Robert Bosch, Scott Devine, and Ben Verghese developed the simulator and were always willing to add the unique features required for Hive.

The FLASH hardware team responded energetically to the opportunity to add new fault containment functionality to their design. I thank them all for their help; in particular, Joel Baxter and John Heinlein built custom features to support Hive, while Jeff Kuskin, Dave Ofelt, Mark Heinrich, Hema Kapadia, and others modified their designs where needed.

I thank the Fannie and John Hertz foundation for the graduate research fellowship that supported me at Stanford, ARPA for funding the FLASH project in contract DABT63-94-C-0054, and Silicon Graphics for both development machines and the IRIX source code on which Hive is based.

Finally, very special thanks to Earin Pinkerton—constant companion, hiking buddy, racquetball fiend and gourmet chef—for her friendship and love throughout my Ph.D. studies, and to my parents Paul and Susan Chapin for love and support, decades of education, and detailed proofreading that has significantly helped this dissertation.

Preface

Hive is a joint project. Four Ph.D. students and one visitor at Stanford have contributed significantly to the operating system:

- Dan Teodosiu — Process management, RPC subsystem, anonymous memory and failure recovery.
- Scott Devine — Virtual memory system and heap corruption experiments.
- Tirthankar Lahiri — File system.
- Kinshuk Govil — Reboot process, low-level hardware management.
- Yasuichiro Izumi — Memory load balancing.

My primary contribution was to the system architecture, including problem definition, design of the multicellular architecture, choice of the error model, and decisions about which features to put in hardware. I designed the firewall, the short interprocessor send mechanism, the memory fault model, and other hardware-software interfaces. I implemented the initial prototypes of the multicellular boot process, the remote procedure call subsystem, and the anonymous memory manager, as well as the final versions of intercell address space duplication and page fault handling across cells.

Because of the nature of my contribution, this dissertation focuses on the overall system design and its reliability characteristics. Detailed descriptions and evaluations of the various subsystems will appear in the other team members' dissertations.

Contents

List of Tables

List of Figures

Chapter 1
Overview

This dissertation describes Hive, an operating system designed to improve the reliability and scalability of large general-purpose shared-memory multiprocessors. Hive is targeted to run on the Stanford FLASH multiprocessor, a machine currently being built at Stanford.

Hive is a prototype implementation of a novel kernel architecture called a *multicellular kernel*. Rather than running as a single shared-memory program that manages all the machine's resources, a multicellular kernel partitions the machine and runs an internal distributed system of multiple kernels called *cells*. This architecture improves the reliability of the system compared to previous multiprocessor kernel architectures since a hardware or software error in one cell does not crash the whole machine. It also improves scalability since most application requests are serviced by the cell where the application is running, reducing both kernel synchronization delays and memory system bisection bandwidth requirements.

Previous work on multicellular kernels has focused only on their scalability benefits. The research described in this dissertation makes three primary contributions:

- demonstration that distributed system techniques can provide fault containment inside a shared-memory multiprocessor, despite the possibility of wild writes due to software errors;
- specification of a set of hardware features for FLASH, generalizable to other multiprocessors, that is sufficient to support hardware and software fault containment; and
- demonstration that cells can take advantage of shared-memory hardware across cell boundaries at both application and kernel level while preserving fault containment.

This chapter introduces the system and the experimental evaluation done in the dissertation. I start with a description of the FLASH machine whose design stimulated the development of Hive.

1.1 The Stanford FLASH multiprocessor

FLASH is a shared-memory multiprocessor designed to scale to thousands of processors. To reach this gigantic size, FLASH distributes main memory across the nodes of the machine and uses a scalable interconnect network rather than a shared bus (Figure 1.1).

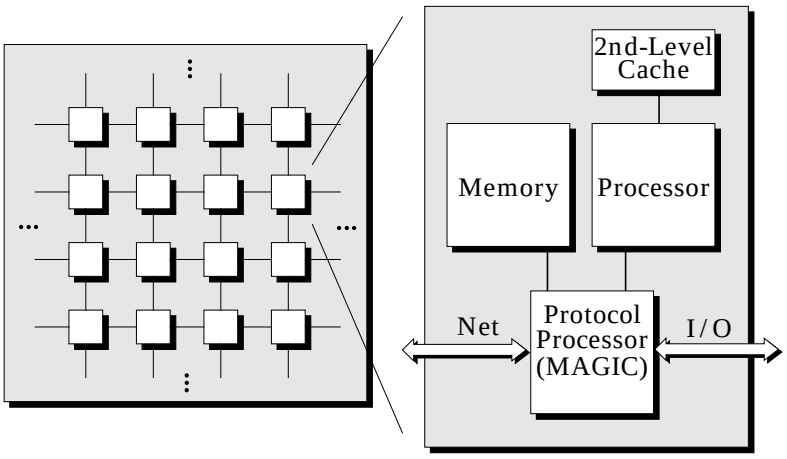


Figure 1.1. The Stanford FLASH multiprocessor.

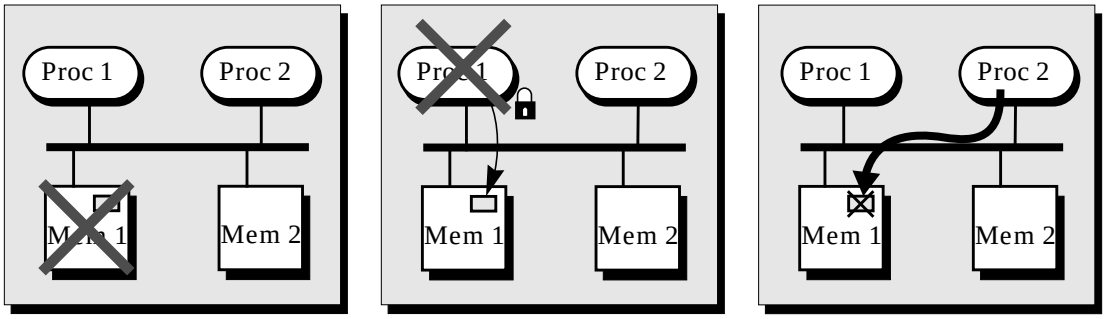
FLASH is a representative *CC-NUMA multiprocessor* (Cache Coherent with Non-Uniform Memory Access time). The name arises because unlike bus-based machines, where all memory is accessible in uniform time, the physically-distributed memory of a CC-NUMA machine makes some addresses slower and some faster to access from the perspective of any given processor. Like other CC-NUMA multiprocessors, FLASH uses a directory-based cache coherence protocol [KOH+94].

The core of the FLASH design is a programmable *protocol processor* in each node that implements the complex algorithms required for directory-based cache coherence. The protocol processor chip is called *MAGIC* (Memory and General Interconnect Controller). It executes microcode from and stores its data structures in the main memory of the node, using caches to reduce instruction and data access latency.

Because MAGIC is programmable and has essentially unlimited code and data storage, FLASH is much more flexible than previous multiprocessors whose memory system protocols are implemented in dedicated logic. Memory system behavior can be changed and new features added by recompiling the protocol microcode and rebooting the machine. This flexibility offers the opportunity to design new hardware and operating system software features that work together to provide novel system functionality.

1.2 Motivation for a new operating system

The developers of FLASH and most other scalable CC-NUMA machines (including those manufactured by Silicon Graphics [Sil96], Sequent [LoC96], Data General [Dat96], and Hewlett-Packard [HP95]) intend them to be general-purpose multiprocessors. A general-purpose machine is one that runs standard commercial or engineering applications and efficiently provides the



(a) Memory module failure: run queue lost.

(b) Processor failure: run queue lock never released.

(c) Wild write: run queue corrupted.

Figure 1.2. What can go wrong in an SMP kernel.

services expected from commercial systems, such as multiprogramming, networking, security, and system administration.

Multiprocessors are also used in other ways than as general-purpose computers. They are used as supercomputers, an environment in which design goals such as efficient multiprogramming and compatibility with standard applications can be sacrificed in order to improve raw performance. Multiprocessors can also be used to implement high-availability (fault-tolerant) systems, in which the multiple processors provide replication in order to ensure continuous operation. These other uses are important, but developers of scalable multiprocessors target general-purpose use because the market, notably for managing commercial databases, is much larger than that for supercomputers or fault-tolerant systems.

Unfortunately, current general-purpose operating systems have significant reliability and scalability problems when run on large multiprocessors. This is due to their structure as *symmetric multiprocessing* (*SMP*) operating systems, in which all processors share a single copy of most kernel data structures. Monolithic kernels such as UNIX and VMS and microkernels such as Mach, Chorus, and Windows NT are all SMP kernels by this definition.

This architecture causes reliability problems because an SMP operating system must be rebooted to recover from most errors. To build intuition for this observation, consider the effects of an error that damages a core kernel data structure such as the run queue (Figure 1.2). In case (a), memory module 1 fails, causing the run queue data to become inaccessible; no new processes can be scheduled anywhere in the machine. In case (b), processor 1 acquired the run queue lock to ensure mutual exclusion, but failed before releasing the lock; any other processor trying to access the run queue will spin forever waiting for the lock to be released. In case (c), processor 2 uses an

uninitialized pointer for a store instruction, thereby corrupting the data in the run queue; the operating system will probably fail when the next processor to access the run queue finds it in an inconsistent state.

The variety of possible errors in an SMP kernel makes it extremely difficult to recover without rebooting. Therefore, as machines grow in size and the error rate increases, the mean time to failure decreases. Beyond some size, SMP operating systems will no longer be reliable enough for most general-purpose applications.

Scalability is also limited by the widely-shared data structures of an SMP operating system. Improving the parallelism of an SMP operating system is an iterative trial-and-error process of identifying and fixing bottlenecks. At larger machine sizes the bottlenecks become subtle, such as false sharing of cache lines or high conflict miss rates caused by simultaneous access to otherwise unrelated data structures. Memory system bottlenecks are also highly workload-dependent, so an operating system that performs well when tested by the developer can perform poorly in production use. These problems cause the cost of delivering a high-performance SMP operating system to increase sharply with the size of the multiprocessor, limiting the size of the machine that is economically feasible to manufacture.

Without significant improvements in operating system reliability and scalability, large multiprocessors are unlikely to succeed as general-purpose computing platforms.

1.3 Multicellular kernel architecture

These reliability and scalability problems can be addressed by restructuring the operating system as an internal distributed system of cells (Figure 1.3). Each cell is an SMP kernel. Each independently manages a portion of the processors, memory, and I/O devices of the machine, supports applications running in that portion of the machine, and shares resources with other cells as needed for performance.

The multicellular kernel architecture improves reliability because the cells can defend against each other's failures. An error may cause one or several cells to fail, terminating the applications running on those cells, but the rest of the machine is unaffected. Unlike other kernel software architectures that assume that the operating system is correct, the multicellular kernel architecture provides reliability with respect to operating system software errors, which is important because most failures observed in the field are caused by software errors [Gra90, CVJ92, ChB94].

A multicellular architecture improves scalability because few kernel data structures are shared by processes running on different cells. Increasing the number of cells systematically improves the

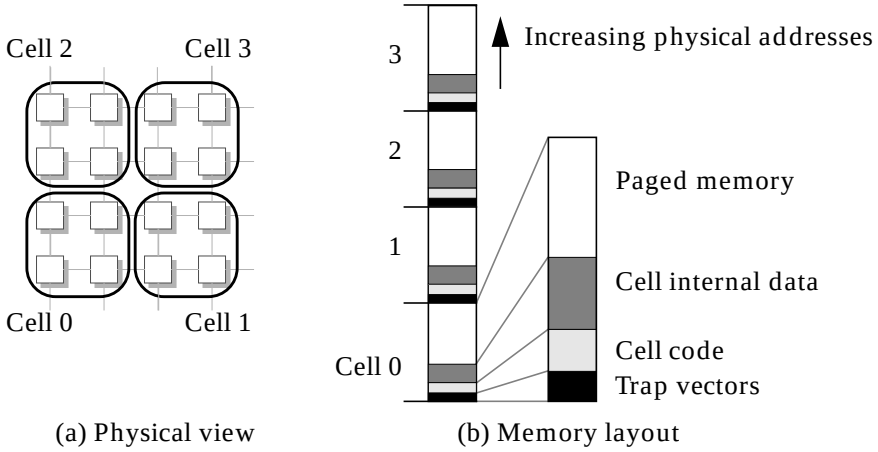


Figure 1.3. Partition of a multiprocessor into Hive cells.

parallelism of the operating system and also increases the locality of kernel memory accesses, which reduces hardware bottlenecks on a CC-NUMA multiprocessor. The scalability problems that remain involve explicit communication among cells and so are easier to control than the implicit communication through shared memory that occurs in an SMP operating system.

1.4 Implementing a multicellular kernel

A multicellular architecture creates several implementation challenges that do not arise in existing multiprocessor operating systems:

- *Cell isolation:* The effects of errors must be confined to the cell in which they occur. Additionally, if the system is to survive hardware errors, the hardware must provide features that support restoring the memory system and interconnect to a functional state after a hardware error.
- *Resource sharing:* Processors, memory, and other system resources must be shared flexibly across cell boundaries, to preserve the execution efficiency that justifies investing in a multiprocessor. The cell boundaries must not add high performance overheads when resources are shared.
- *Single-system image:* The cells must cooperate to present a standard SMP OS interface to applications and users.

Many of the problems faced in implementing a multicellular kernel also arise in single-system-image distributed systems such as Sprite [OCD+88] and Locus [PoW85]. However, the presence of shared-memory hardware between the cells and the dramatically higher bandwidths and lower

latencies of a multiprocessor interconnect make many of the solutions used in distributed systems inappropriate for a multicellular kernel.

A particular problem that affects the whole system design is the possibility of *wild writes*, stores to incorrect addresses caused by kernel software errors. Novel software mechanisms in Hive and novel hardware mechanisms in FLASH work together to implement cell isolation and resource sharing despite the possibility of wild writes.

1.5 Definition of fault containment

The application-visible reliability model that results from the multicellular design is called *fault containment*. This reliability model is familiar from many previous distributed systems. Informally, an application may fail but only if the error occurs in the part of the system that the application is using.

It is useful to define fault containment more precisely:

> *A system provides fault containment if the probability that an application will fail is proportional to the amount of resources used by that application, not to the total amount of resources in the system.*

The precise definition is better than the informal one for several reasons. First, in a multiprocessor with dynamic fine-grained resource sharing, the "part of the system that the application is using" is frequently poorly defined. Second, the operating system or the hardware may need to make the application vulnerable to the failure of some part of the system that it is not using, either to improve performance or to simplify the implementation. Finally, the precise definition gives a metric that allows comparison of different systems or design features that attempt to provide fault containment.

1.6 Usefulness of fault containment

The fault containment strategy differs significantly from the more traditional *fault tolerance* model in which the system attempts to guarantee that no applications fail when an error occurs. Fault containment is weaker than fault tolerance, because some applications will fail when an error occurs, but appears to have lower overheads because it avoids replication. The question is whether the weaker model is useful.

Fault containment in a multiprocessor provides reliability benefits for any workload with multiple processes where some processes can continue doing useful work after others fail. Many general-

purpose workloads appear to have this property. Examples include engineering and computer-aided design, graphics and virtual reality, software development, and general interactive use.

However, fault containment does not improve the reliability of large processes that use resources from the whole machine. These are the applications that probably justified purchase of the machine in the first place, such as database, multimedia, and web servers or large engineering or scientific simulations. Fortunately, these programs are the focus of significant software engineering investment. It is reasonable to assume that they could be modified to improve their reliability on systems with fault containment. There are three ways to do this:

- Large applications can in many cases be decomposed into independent smaller tasks that benefit from fault containment. Examples of such applications include decision support and data mining, where an initial long-running query can be split into smaller independent subqueries. Other large applications, such as multimedia and web servers, naturally split into multiple processes where each services a client or group of clients.

- Nondecomposable applications that require high availability can be restructured as process pairs [Bar81, SiS92]. A system with fault containment will support such applications well if it exposes sufficient control over resource allocation that the processes of the pair can avoid common points of failure. Traditionally such applications have run on fault-tolerant systems, but in cases where human safety is not at risk a general-purpose multiprocessor with fault containment may be a more cost-effective solution.

- Nondecomposable applications that do not require high availability can use checkpointing to provide roll-back recovery from failures. Batch processes such as engineering and scientific simulation or graphics rendering tend to do all their I/O to files or directly to humans (e.g. graphical output and computation steering). Checkpointing such applications is straightforward, and various user-level checkpointing libraries have been written that require little effort to exploit [LiS92, PBK+95]. Operating system support can also be useful for checkpointing [LNP94, SiS92] and could easily be integrated into a multicellular kernel.

1.7 Success conditions

Although fault containment appears to be useful for a wide range of general-purpose applications, this is not sufficient by itself to justify the immense investment required to adopt a new kernel architecture. For a multicellular kernel to succeed as a general-purpose operating system, it must achieve four goals:

- *Fault containment:* The kernel must demonstrate substantial improvements in reliability compared to SMP kernels.
- *Competitive performance:* The kernel must not add substantial performance overheads compared to SMP kernels when supporting workloads on the medium-sized systems (16 to 32 processors) that will dominate the market for the foreseeable future.
- *Scalability:* The kernel must provide excellent performance scalability for operating system-intensive workloads such as databases as the system grows to large sizes (128 processors or more).
- *Binary compatibility:* The kernel must execute unmodified legacy applications correctly and efficiently.

1.8 Implementation of Hive

Hive is a prototype built to evaluate whether a multicellular kernel is capable of achieving these goals. It is not intended to demonstrate a complete implementation of a commercial multicellular kernel. In particular, the single system image is incomplete, the file system is primitive, and some resource sharing mechanisms are not present. However, the current prototype includes all of the features required for an initial evaluation of the multicellular kernel architecture: a complete cell isolation and failure recovery subsystem, a virtual memory system capable of several kinds of memory sharing across cell boundaries, a well-tuned remote procedure call subsystem, and distributed process management sufficient for remote process creation and distributed process groups.

The most innovative parts of the implementation are those that take advantage of and defend against the problems caused by shared-memory hardware. The wild write defense and experiments on using shared memory between cells are particularly noteworthy. One interesting observation is that shared memory is less useful as a kernel-level communication mechanism between cells than was expected (Section 7.2).

To make the lessons learned from the prototype as relevant as possible to potential commercial adopters of a multicellular architecture, Hive is binary compatible with a widely-used commercial SMP operating system (IRIX 5.2 from Silicon Graphics, Inc., a version of UNIX System V Release 4). Hive is implemented as an extensive modification of the IRIX 5.2 code base.

1.9 Experimental evaluation

Because the FLASH machine is not yet operational, the experiments reported in this dissertation use the SimOS machine simulator [RHW+95, WiR96, RBD+]. SimOS offers a choice of processor and memory system models that trade off between the speed and accuracy of simulation. Experiments on the reliability of the system use the high-speed mode of SimOS, while timing experiments use a slower mode that is more accurate coupled with a cycle-accurate reference simulation of the FLASH memory system.

Reliability experiments: The reliability experiments are a series of fault- and error-injection studies. These studies include corruption of data in the kernel heap, corruption of kernel instructions, and failure of FLASH nodes. The workload for the reliability experiments is a parallel make, which has the attractive properties of independent subprocesses and easily-checked output. An experiment is considered to succeed if none of the files produced by the parallel make are corrupted and the only cell that fails is the one where the fault or error is injected. Hive shows no data corruption in any experiments, and limits the effects of faults and errors to the cell where they are injected 96% of the time.

Performance experiments: The performance experiments use microbenchmarks and three high-level workloads: *pmake*, *raytrace*, and *ocean*. Pmake is a parallel make, which stresses the system in ways characteristic of general-purpose use. Raytrace and ocean are parallel scientific applications from the Splash-2 benchmark suite [WOT+95].

The use of simulation limits the performance experiments to small system sizes. The largest system studied has only eight processors and 256 megabytes of memory. When combined with the limited set of features implemented in the prototype, this makes it difficult to draw any conclusions about the performance of multicellular kernels on large machines. However, the prototype does show interesting performance trends and tradeoffs that suggest that the multicellular design is promising for larger systems.

On an eight-processor system, an eight-cell Hive configuration shows no slowdown for pmake, a 20% slowdown for raytrace, and a 5% performance improvement for ocean compared to an IRIX baseline. In all three workloads the amount of time spent executing the operating system is highest when running with two cells, then decreases as the system increases to eight cells due to reductions in memory system stall time and kernel lock contention. Ocean and raytrace both suffer a convoying effect that increases time spent spinning on locks at user level, indicating that it is important to spread the kernel workload evenly across the multiple cells when supporting parallel applications that synchronize frequently.

1.10 Terminology

This dissertation follows the standard terminology used to describe dependable computing systems [Joh89]. A *fault* is a latent problem such as a mistake in a line of code or a short circuit on a chip. An *error* is the result of activating a fault, such as executing incorrect code under conditions that cause it to give the wrong result or reading an incorrect value from a shorted line. A *failure* occurs when an error causes the externally-observable behavior of a device or software system to deviate from specification. A failure is *fail-fast* if the failed component either produces no incorrect output before halting or produces only output that is so corrupt that all other components receiving the output detect the failure immediately.

In the case of a multicellular kernel, a software or hardware error may cause a cell to fail, but the system as a whole does not fail if it successfully confines the effects of the error to the directly affected cells. An error that leads to the failure of other cells or a complete system failure is said to cause an *uncontained failure*.

There is one violation of the standard terminology. The term *fault containment* used throughout the dissertation should properly be replaced by *error containment with forward error recovery*. The term *fault containment* was chosen by analogy to the widely-used term *fault tolerance*, which similarly values clarity and conciseness over pedantic precision.

1.11 Outline of the dissertation

The rest of the dissertation is structured as follows:

- Chapter 2 provides the motivation for developing Hive by analyzing the limitations of SMP operating systems.
- Chapter 3 describes the overall fault containment architecture of the system and the software architecture of Hive.
- Chapters 4 and 5 describe the implementation of the Hive prototype and the experimental setup for the dissertation.
- Chapters 6 and 7 focus on the implementation of fault containment and resource sharing in the prototype. Chapter 8 measures the performance of the prototype.
- Chapter 9 discusses what has been learned about the tradeoffs of the multicellular architecture.
- Chapters 10 and 11 conclude the dissertation with a survey of related work and a summary of the main results of this research.

Chapter 2
SMP OS limitations

The previous chapter gave an overview of the motivation and design of the system. I now turn to a more detailed investigation of the reasoning behind the design of Hive. In particular, I provide evidence that current SMP operating systems are insufficient for future large multiprocessors.

SMP operating systems have been a successful design for small-scale general-purpose multiprocessors. However, it is commonly accepted that they are difficult to scale even to current high-end machines with a few tens of processors. New problems arise in larger multiprocessors that make scaling even more difficult. The first part of this chapter focuses on reliability issues, while the second studies performance issues.

2.1 Reliability issues

When considering reliability on larger machines, it would be best to start with data about the reliability of current machines. Unfortunately, no information on the failure rate of multiprocessors running current SMP operating systems is available. However, several published studies that combine uniprocessors and multiprocessors give hints about the magnitude of the rate.

2.1.1 System failures in practice

[TaI92a] uses automatically-maintained error logs to analyze failures in VAXcluster systems running VMS. Over 30 machine-years of data shows an overall mean time between failures (MTBF) of 20 machine days. This rate is high due to the immaturity of the operating system at the start of the data collection period. The MTBF due to hardware errors was 60 machine-days, while the MTBF due to software errors improved from 19 days in the first year to 2.8 years four years later. Since this data was collected from VAXclusters, which are more tightly coupled than the separate machines on a standard LAN, the software error rate is probably higher than would be observed on independent machines.

[CBR95] analyzes problems reported to IBM's service organization about two releases of "a large IBM operating system." This data covers software errors only, not hardware errors, and includes errors that did not lead to system failures. Release 1 had a rate of about 5 months between errors at

one year after release, while Release 2 had a rate of almost 2 years between errors at a similar point in its life cycle. At the end of the measurement period the rates were 4 years and 2 years between errors. Assuming that between 60 and 90 percent of operating system errors lead to system failures, as measured in the VMS study, the IBM systems at one year after release had a MTBF of most of a year for Release 1 and several years for Release 2. This failure rate is low because it excludes hardware errors.

These studies support the hypothesis that general-purpose machines require a MTBF due to hardware and software faults of between one month and one year. In each case the operating system was released at the bottom end of this range and the manufacturer then invested sufficiently in improving it to reach or somewhat exceed the top end of the range.

There have been other studies on failure rates in the field, but the systems examined differ more from the SMP operating systems currently used by commercial multiprocessor vendors than those investigated in these studies. [Iye95] provides a survey of the literature.

2.1.2 Failure rates of large machines

Scaling to larger systems will cause the MTBF of SMP operating systems to decrease significantly. This is intuitively obvious with respect to hardware errors, because larger machines have more errors and an error anywhere in the machine causes the operating system to crash. Note however that the hardware error rate does not scale directly with machine size, because improved integration in each hardware generation reduces the number of components required to build larger machines.

Less obviously, scaling the size of the machine also increases the rate of software errors in an SMP operating system. For efficient performance on a larger machine, the parallelism of an SMP operating system must be increased using finer-grained locking, data structures partitioned among the processors, and similar techniques. These changes create an increased risk of software errors in operation, because parallelized code is the hardest type of code to analyze and test during system development.

This observation about parallelized code is common wisdom among programmers but is also supported by field data. For example, one study of customer problem reports on Tandem Guardian90, an operating system tested thoroughly by the manufacturer because of its intended use for high-availability applications, found that race conditions and timing problems were more prevalent than any other type of software faults that crashed systems in the field. This was true both for the number of faults in the code and for the number of failures resulting from those faults [LeI93].

As a general-purpose machine grows in size, not only does the operating system require increased internal parallelism, but more applications run at the same time. This leads to an increased number of dynamic interactions between processes, increasing the potential for stimulating any latent parallelism-related faults in the operating system code. The relationship between system workload and software error rate is well-documented [CaS82, MoA87].

Put together, the increased rate of hardware errors, the increased use of parallel constructs in the operating system code, and the increased dynamic parallelism of the system suggest that the MTBF of large multiprocessors will be substantially lower than that of current multiprocessors if current SMP operating systems are used.

2.1.3 Reliability requirements

At the same time that technological factors push MTBF downwards, market pressures require that large multiprocessors provide substantially higher MTBF than current small-scale systems. There are two reasons for this:

- *Change in applications:* Among the primary applications that can justify purchase of these expensive machines are the transaction processing and decision support databases that currently run on mainframes, where the MTBF expectation is much higher than on current small-scale systems. Web and multimedia servers may also be important applications for large multiprocessors, and they have very high availability requirements.
- *Greater impact of each failure:* The sheer size of large multiprocessors will give each failure a greater impact than in current multiprocessors. Consider that an organization that uses a large multiprocessor will centralize more of its computing resources in that one machine than in any current small-scale multiprocessor. Therefore a failure will interrupt service to or work done by more people. The machine will also take much longer to return to steady state performance after a reboot, since it will require much more work to refill a larger main memory, restart a larger number of processes, and recover all the cached knowledge those processes maintain about their own workloads and the state of the world.

The change in applications and the greater impact of each failure imply that the cost of a failure on these machines will be substantially higher than on current small multiprocessors. Therefore users will require higher reliability or they will find the machines uneconomical to use.

An MTBF similar to current mainframes seems necessary to open much of the mainframe market to multiprocessors. MTBF levels similar to current small multiprocessors may leave large multiprocessors without a sufficient market to justify their development.

2.1.4 Improving reliability

It may be the case that the only improvement necessary is better engineering and testing techniques for the operating system. The software error rate dominates the hardware error rate on current systems [Gra90, CVJ92, ChB94] and is likely to do so even for machines several times larger than the largest current multiprocessor. Reducing the rate of software errors may provide sufficient reliability improvements to eliminate the need for more radical changes.

However, this approach will require high engineering investments due to the difficulty of finding parallelism-related software faults. At larger system sizes where the hardware error rate becomes significant, or if significant reliability improvements are required as suggested in the previous section, SMP operating systems appear unlikely to provide an acceptable solution.

2.2 Performance issues

Although reliability is important, performance is the primary motivation for users to invest in large multiprocessors. Large multiprocessors must achieve near linear scalability of system throughput to attract users who would otherwise choose clusters of smaller machines, and must achieve substantial price-performance advantages compared to existing mainframes to attract their users.

On larger machines, operating systems encounter two primary performance problems that have traditionally been ignored on smaller machines. Both are related to the memory system. Communication latency in these machines is higher relative to processor speed than in small machines, so any algorithm that suffers frequent cache misses is a potential performance bottleneck. Moreover, locality of memory access is important to reduce access latency and minimize interconnect contention, so widely-shared data structures are slower than localized data structures even when the frequency of cache misses remains constant.

This section reports data on these memory system costs and the problems that SMP operating systems face in reducing them. The data comes from a study of an SMP operating system, IRIX 5.2, running on a 32-processor CC-NUMA machine, the Stanford DASH [LLG+92]. IRIX has been parallelized to run efficiently on large bus-based multiprocessors (SGI Challenge machines support up to 36 MIPS processors), so its memory system behavior is not skewed by bad synchronization behavior at this system size.

Rather than giving complete details of the study here, I highlight a few selected points. Interested readers may refer to [CHR+95] for details and much more data.

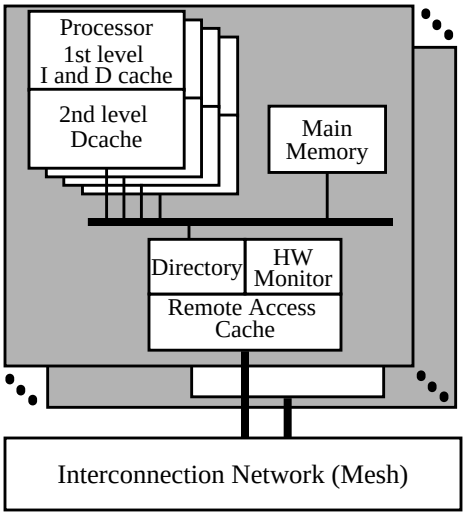


Figure 2.1. Architecture of the Stanford DASH multiprocessor.

2.2.1 Experimental setup

Detailed trace data was collected from a DASH configuration with eight clusters (Figure 2.1). Each cluster is a slightly modified Silicon Graphics POWER Station 4D/340, which is a bus-based multiprocessor with four 33-MHz MIPS R3000 processors. The processors have relatively limited caches: 256 kilobyte external data cache (backing a smaller on-chip data cache) and 64 kilobyte instruction cache, both direct-mapped with 16-byte lines.

Cache fills from local memory stall the processor for a minimum of 29 processor clock cycles. Remote memory requests take at least 101 processor clock cycles if the data is clean in memory and 132 processor clock cycles if it must be fetched from a cache in another cluster. Remote cache misses are satisfied in the latency of a local access if they hit in the remote access cache, which is a 128 kilobyte direct-mapped cache with 16-byte lines.

The workload used for the study has features characteristic of a software development or engineering environment. The workload contains two 16-way parallel makes, five copies of a microbenchmark suite, and eight copies of a moderately large engineering analysis program.

Each DASH cluster includes a hardware monitor that traces bus activity without affecting the timing of the machine. Disk capacity on DASH limits the total trace that can be collected to about eight seconds of execution. The data reported here comes from multiple eight second samples taken at different times during separate workload runs.

2.2.2 Overall behavior

Under the workload studied, the system spends an average of 63% of non-idle time running applications and 37% of non-idle time in the operating system. The components of the non-idle operating system time are 5% of non-idle time executing useful instructions, 3% spinning on locks, and 29% stalled on cache misses. In other words, this SMP operating system spends four-fifths of its work time stalled on cache misses.

Roughly half of the cache miss time is due to instruction cache misses, which are higher on DASH than on more recent systems because of the small direct-mapped instruction cache. Excluding instruction cache misses, IRIX still spends over three times as much time stalled on data cache misses (16% of non-idle time) as it does executing useful instructions.

Cache simulations using the trace data from DASH show no substantial reduction in data cache misses even with a one megabyte two-way set-associative data cache. This indicates that most of the data cache stall time is caused by communication misses.

Clearly, even at a relatively modest system size of 32 processors, this SMP operating system suffers from a significant memory system performance bottleneck. The bottleneck will become even more significant on FLASH and other large multiprocessors which have a higher relative remote cache miss latency than DASH.

2.2.3 Detailed observations

Half of the operating system data cache miss time comes from the memory block transfer routines (`page_copy`, `page_zero`, and similar routines). The stall time in these routines is primarily spent waiting for cache misses to remote memory. The cause of the problem is not the SMP kernel architecture but the lack of NUMA-aware page allocation and processor scheduling in IRIX 5.2, combined with its failure to take advantage of the hardware prefetch support provided by DASH. Proper policies and use of hardware mechanisms would both reduce block transfer latencies and improve memory access locality.

The other half of the data cache miss time is dominated by interprocessor communication hotspots that are caused by the SMP kernel architecture. Table 2.1 shows the data structures with the highest cache miss times, leaving out large data structures such as the process table whose high stall times come from their large size. There are two types of data structures in the table: those whose functions make them inherently hot and those which are accidentally hot.

A structure is inherently hot if its function requires it to be written frequently and read or written by multiple processors. In this category, the table shows the fast clock, the control counters for a

Table 2.1. Data cache hotspots in IRIX over 22.8 seconds of execution on DASH.

Structure	Stall Time (msec)	Description of Structure
`fclkcount`	863	Fast clock
`freemem`	510	Number of pages available
`bucket`	490	Fast clock acknowledge dummy variable
`anon_shake_lock`	422	Reader / writer counters for copy-on-write tree
`bfreelist`	198	I/O buffer free list header
`splimplock_owner`	169	Processor with rights to execute network code
`Runq`	148	Head of process run queue
`memory_lock`	148	Pointer to hardware spin lock for physical memory management structures

frequently-acquired reader-writer lock, the head links of a free list, and the run queue head pointer. Reducing the cost of accessing these structures requires algorithmic changes, for example changing the globally-shared run queue into multiple per-processor or per-cluster run queues.

A structure is accidentally hot if the cache miss time is due to programmer error or false sharing. For example, it is surprising to see high cache miss times for `memory_lock` because it is written once at boot time and never changed. However, it is read frequently and happens to be on the same cache line as a frequently written counter (`usermem`, the number of pages available to users). Note that under the workload studied this innocuous variable accumulated as much stall time as the head pointer of the global run queue.

2.2.4 Improving performance

Reducing communication hotspots and the overall memory system performance bottleneck will require a never-ending cycle of iterative improvements to SMP kernels as the system grows. In particular, the uncontrolled interprocessor communication inherent in the SMP operating system architecture means that it is likely that each increase in system size or change in workload will stress new parts of the system and cause new hot spots that limit performance.

A familiar example of this architectural characteristic is the synchronization behavior of SMP kernels. For example, in IRIX 4 running on four processors the global run queue lock is the only one with substantial contention [TGH92]. The developers made significant algorithmic changes to

eliminate this hotspot by the time IRIX 5.2 was released. However, in growing the system to 32 processors, another lock (`memory_lock`) came under high contention. Additionally, otherwise innocuous locks can create significant performance problems when workloads behave differently than the ones tested by the developers. The ocean workload measured in Chapter 8 provides an excellent example of this problem.

Even when the memory system costs of an SMP kernel are reduced by modifying its algorithms to reduce communication between processors, false sharing can cause significant performance problems. Even on DASH, with 16-byte lines that hold only four integers or pointers, false sharing caused significant hot spots. Systems with longer lines, such as FLASH at 128 bytes, can expect severe performance impacts. Improved measurement and linking tools will help reduce false sharing, but each increase in system size or change in workload can be expected to stress the system in different ways and bring out new false sharing problems.

Improving operating system performance without continual reengineering requires a systematic way to increase operating system parallelism and reduce global data sharing. The multicellular kernel architecture offers one promising approach, by ensuring that processes running on separate cells share few kernel locks or data structures. A multicellular kernel still faces problems of reducing communication, but both the fundamental communication rate and the number of points at which communication can occur are much lower, making it much easier to detect and solve scalability problems. At the end of the dissertation, Chapter 9 returns to this discussion and compares the complexity and scalability of multicellular and SMP kernels in light of the experience gained from the Hive prototype.

Chapter 3
Hive architecture

Chapter 1 describes a multicellular kernel as an operating system structured as an internal distributed system of cells. However, the architecture of Hive covers a much wider domain than just the software architecture that implements this idea. Because reliability is a goal, the choice of error model is part of the system design. Because Hive is an operating system, the interface and feature set exported to applications are important. Because Hive is an integral part of a hardware project, the design includes novel hardware features that are appropriate for a multicellular kernel.

This chapter describes the architecture in two parts. The first part is the *fault containment stack*, a layering similar to a networking protocol stack that defines both the expected errors and the responsibilities of each level of the system. The second part is the software design of Hive itself, which implements one layer of the fault containment stack. The chapter concludes with a brief description of the implementation of the lower layers of the stack in FLASH.

Some of the design features described in this chapter are not implemented in the prototype. It is useful to present the whole system design despite the limitations of the current implementation, both because the design drives ongoing research on Hive and FLASH and because it explains the decisions made in implementing the prototype. The next chapter describes the current implementation.

3.1 Error model

The error model is a set of assumptions about the types of errors that can occur and their effects on the system. The error model drives the *reliability design* of the system, that is, design of the hardware and software mechanisms for cell isolation and failure recovery. If an error occurs that is not in the model, the behavior of the system is unpredictable.

Just as with any other system designed to improve reliability, choosing the error model for Hive requires making tradeoffs between reliability and performance. An error is left out of the model, and thus can cause the system to fail, if its probability is low relative to the complexity and performance cost of defending against its effects.

Table 3.1. Fault containment stack.

Layer	Example design features
Application	None needed; checkpointing and process pairs useful
Operating System	Independent cells, wild write defense
Memory System	Timeouts on cache misses, recover consistency after hardware errors
Network	Drain undeliverable packets, reroute around failed components
Data Link	Error correcting codes, completion of truncated packets
Physical	Multiple fans and power supplies, hot-pluggable boards

The error model is described in detail in the next section on the fault containment architecture. To summarize it, the most likely errors are assumed to have the following descending order of probability: operating system software errors, power or cooling failure to a portion of the machine, link failure in the interconnect, and halt of an individual node. Link failure is relatively probable compared to what would be expected in a smaller machine, because FLASH has multiple cabinets with network links stretched between them that are vulnerable to human error (i.e. tripping over a cable).

Hive and FLASH carefully avoid assuming that only one error will occur at a time. A multiprocessor is a tightly coupled system and a single problem such as a cooling failure could cause errors in multiple cells at the same time. Even worse, multiple errors are likely to occur at slightly different times, as when boards overheat at different rates. A software error in one cell that manages to corrupt another cell will cause similar behavior. Therefore the most likely time for an error to occur is while the system is recovering from a previous error.

3.2 Fault containment architecture

Table 3.1 summarizes the fault containment stack. The bottom four layers are part of FLASH; Hive is the operating system layer.

Fault containment must be implemented from the top down of this stack rather than from the bottom up as one might hypothesize. For example, having cabinets with independent power supplies does not improve the reliability of applications if loss of a cabinet crashes the operating system. In contrast, a multicellular kernel could run without hardware support and still provide reliability benefits through reducing the impact of operating system software errors.

In the next sections I use the fault containment stack to describe the architecture of the system. First I describe the errors that can occur at each layer, then the fault containment features it provides.

3.2.1 Application level

Error model: Just as in SMP operating systems, the reliability design of the system assumes that applications are potentially erroneous or even malicious.

Fault containment: I assume two types of applications in the workload: a large number of naive applications that are unaware of the fault containment properties of the system, and a small number of sophisticated applications implemented with fault containment in mind.

Naive applications use only the standard UNIX system interface. These applications require the operating system to automatically provide fault containment. Some of the naive applications might be very large, for example parallel scientific or engineering simulations that run with as many threads as there are processors in the machine. I assume that it is important to provide fault containment to a workload that combines large and small naive applications. It is acceptable in this case for the large applications to receive no automatic fault containment benefits, but the small applications should continue to be protected.

Sophisticated applications that seek to improve reliability at user level, such as by using process pairs or the other mechanisms described in Section 1.6, need to control the set of cells whose failure could damage the application. This requires the operating system to export information about the configuration of the system and its cell layout, and to allow applications to limit the activity of the automatic resource sharing and load balancing mechanisms.

Any application that currently runs on a cluster of machines, such as shared-nothing databases and web servers, could run reliably on a multiprocessor by taking advantage of the features provided for sophisticated applications. It may be possible to significantly improve the performance of such applications by adding shared-memory communication regions between the independent processes. The operating system should support this by allowing a sophisticated process to survive the loss of some of its memory resources.

3.2.2 Operating system level

Error model: The reliability design of the system assumes that a software error in a cell will eventually result in deadlock, panic (halt due to failure of a self-check), or partial halt (loss of ability to schedule further processes). However, before a cell reaches that point it may issue wild writes to arbitrary addresses or send corrupt messages to other cells.

In a key simplification of the error model, cells are assumed not to exhibit Byzantine faults [LSP82]. In particular, the sanity-checks that cells apply to messages and data received from other cells are assumed to detect all incorrect messages. Furthermore, a cell that appears failed to one correct cell will appear failed to all correct cells that subsequently test it.

The success of the fault injection experiments described in Chapter 6 suggests that these assumptions are reasonable. These assumptions free Hive from the performance impact and implementation complexity of Byzantine fault-tolerant distributed algorithms [Lyn96]. The assumptions are strong enough that intermittent faults can cause incorrect behavior; if this turns out to be a significant problem in practice, algorithms are known that enable weakening the assumptions without implementing full Byzantine fault tolerance (see Chapter 10).

Fault containment: The operating system is responsible for detecting software errors in cells, recovering operating system data structures to a consistent state after software or hardware errors, and rebooting failed cells so their hardware resources are not lost. It also implements the wild write defense and other mechanisms for cell isolation, using hardware features provided by the memory system level.

For naive applications, Hive maximizes fault containment by minimizing the number of cells to which any given application is vulnerable. It also provides the automatic resource sharing and load balancing required to achieve the execution efficiency that justifies investing in a shared-memory multiprocessor.

Hive provides a mechanism called *spanning tasks* that improves fault containment for naive applications. A spanning task is a UNIX process with threads running on multiple cells. A process that merely uses memory or files from multiple cells does not use the spanning task mechanism. The spanning task mechanism enables the system to run with small cells, providing fault containment for small naive applications, even when the workload contains large naive applications that use many or all of the processors in the machine. The difference between single-cell processes and spanning tasks is not visible to naive applications.

For sophisticated applications, Hive provides several straightforward extensions to the UNIX system interface that give control over the resource sharing and load balancing mechanisms. The only nonobvious extension to the system interface is the management of the *essential dependencies* of a process. When a given cell fails, the process may be terminated immediately. Alternatively, it may receive an error the next time it accesses a resource such as a memory page that was lost in the failure. If the former could occur then the process has an essential dependency on the given cell.

Hive allows an application to discover and, with some restrictions, to modify its *essential set* of cells. This approach avoids cluttering the system interface with controls over various internal kernel resources that are not normally visible to applications.

3.2.3 Memory system level

The memory system level consists of the MAGIC chip and its firmware that together respond to cache misses and uncached accesses, maintain cache coherence, and provide features such as interrupts and interprocessor message sends.

Error model: The reliability design assumes that the memory system level is fail-fast. Potential errors that are therefore outside the error model include returning the wrong memory line to a cache miss and sending a cache writeback to the wrong address in memory.

Note that the fail-fast assumption could be violated more easily in a large-scale multiprocessor running a complicated cache-coherence protocol than in simpler multiprocessors. The FLASH memory system is managed by about 10,000 lines of microcode running on MAGIC. This code is extremely well-tested but presumably still contains faults. Self-checking assertions have been added to the microcode that convert some software errors into fail-fast node halts, but there are no assertions in the most performance-critical sections of the code. A microcode error that does not trigger an assertion is outside the error model and may lead to data corruption or total system failure.

Fault containment: The memory system level provides features for both software and hardware fault containment. Its two most important features are the *firewall* and the *memory fault model*.

- *Firewall:* The firewall supports software fault containment by the operating system. For each page of memory, the firewall gives the cell that owns the page the ability to control which processors in the system can modify the page (Figure 3.1). A processor that attempts to modify data without firewall write permission receives a bus error. The firewall also protects I/O devices, node-control registers, and the firewall state itself, returning bus errors to disallowed uncached reads and discarding disallowed uncached writes.

- *Memory fault model:* The memory fault model supports hardware fault containment by the operating system. The memory fault model is analogous to a memory consistency model which specifies the behavior of reads and writes on a multiprocessor. The fault model tells the operating system designer what assumptions can be made by the algorithms that seek to recover from hardware errors. The most important feature of the memory fault model is the assumption that when the operating system sets the firewall to limit write permission for a

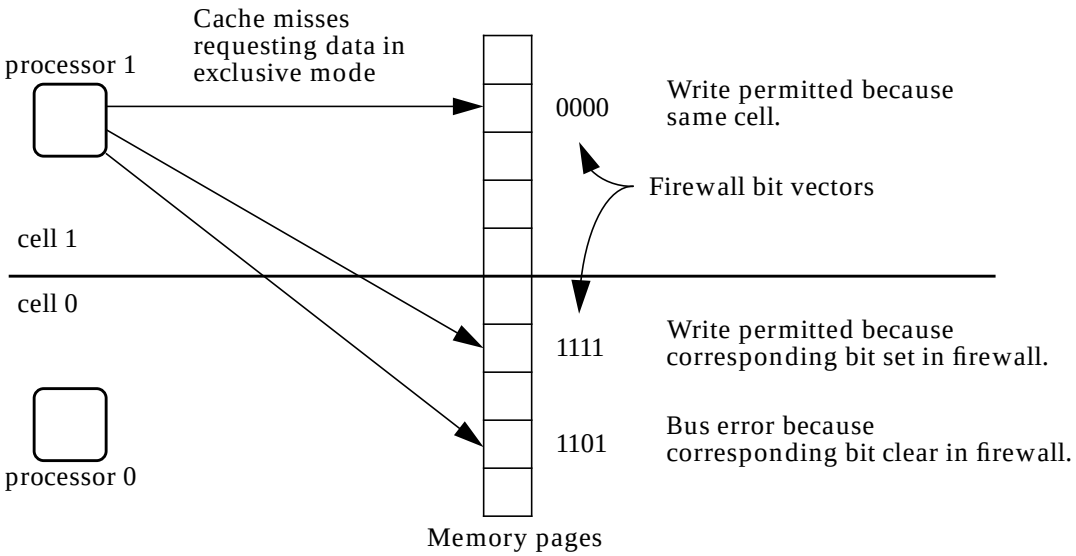


Figure 3.1. Operating system view of the firewall.

page to a set of cells, the data on that page will not be damaged by a hardware error outside that set of cells.

The firewall is a straightforward extension to standard cache coherence mechanisms. In contrast, the memory fault model requires design features at the memory system level and all lower levels of the fault containment stack. Features at the memory system level include detection of hardware errors through timeouts, initiation of network level and operating system level recovery processes, and recovery of the cache coherency protocol to a consistent state after hardware errors. Features at the lower levels are described in the next three sections.

3.2.4 Network level

In normal operation the network level provides a reliable communication service to the memory system.

Error model: Only two network-level errors are assumed to occur: packet loss and delivery of packets to the wrong destination node. Surviving packet loss is a key part of the reliability design because misdelivered packets, packets flagged as errors by the data link level, and almost all nontrivial physical-level errors lead to actual or apparent packet loss.

However, unlike local-area and wide-area networks, the reliability design assumes that congestion does not lead to packet loss. Like most multiprocessor interconnects, FLASH provides lossless

flow control in order to enable the endpoints to avoid the cost of an end-to-end retransmission protocol. Therefore packet loss can be assumed to be a rare event, occurring only after a hardware error. Thus it is acceptable for packet loss to lead to an application-visible error.

Fault containment: The network level plays two fault containment roles. It reconfigures the routing tables in the interconnect so that surviving nodes remain connected after a physical-level error. It also guarantees that a physical-level error in one cell does not cause loss of internal packets sent between the nodes of another cell, by ensuring that only packets destined to or routed through a failed area are discarded when recovering from a physical-level error. This guarantee is required to enable the memory system to implement the memory fault model.

3.2.5 Data link level

In normal operation the data link level provides bit error detection and correction to the network level.

Error model: A data-link level error would occur if a corrupted packet were delivered but appeared correct. The reliability design assumes that there are no data-link level errors. It would be too expensive to implement an end-to-end error correction protocol to cope with this case, given the low-latency requirements of the memory system.

Fault containment: In addition to assuming that the data link layer detects and corrects bit errors in packets, the reliability design requires the data link layer to complete (and flag as corrupt) packets that are truncated by an error in a router, link, or node. This feature allows the network level to treat packet delivery as atomic.

3.2.6 Physical level

The physical level consists of all the hardware components of the machine, both active such as processors and infrastructure such as power and cooling.

Error model: Errors at the physical level can result from logic faults, power or cooling failure to a portion of the machine, and link failure in the interconnect. All hardware components except links are assumed to be fail-fast. Transient errors in links are masked by bit error detection and packet retransmission at the data link level.

The minimal fail-fast unit is the router, MAGIC chip, memory module, processor, or link in which the error occurs. In FLASH, internal self-checks implemented by the recovery algorithms immediately convert the halt of any component in a node to a node halt, so other nodes and the operating system can assume that nodes are fail-fast units.

Fault containment: The reliability design requires the hardware to be partitioned into failure units that support the cellular structure of the operating system. For example, a physical-level error such as loss of a fan or removal of a board should not be a single point of failure for all cells. Moreover, to support the memory fault model, the failure units contain sets of nodes that occupy non-overlapping ranges of the physical address space. These sets of nodes are assumed to be convex in the interconnect. That is, any node in a failure unit can send packets to any other node in that failure unit without traversing links or routers outside the failure unit. Finally, the physical design is assumed to provide hardware reset functionality on a per-cell basis.

3.2.7 Summary of the fault containment stack

The fault containment stack includes features at the operating system, memory system, network, data link, and physical design levels of the system. The features at the hardware levels of the system work together to implement the memory fault model required for hardware fault containment by the operating system. The hardware also provides the firewall, which enables the operating system to implement software fault containment. The goal of all these features is to improve the reliability of naive applications and to allow sophisticated applications to implement fault tolerance at user level.

3.3 Operating system software architecture

Having completed the description of the error model and the fault containment architecture of the system, I now describe the software architecture of Hive itself. A brief section at the end of the chapter covers the lower levels of the stack.

Converting an SMP operating system into a multicellular kernel like Hive conceptually proceeds in three stages. First, modify it to create a single-system image distributed system where the separate kernels coexist in a single machine. Second, add the features required to isolate the cells so that an error in one does not damage others. Finally, add the resource sharing features required to achieve performance competitive with the original SMP operating system. I discuss each of these parts of the system in turn.

3.3.1 Distributed system

At the distributed system level, Hive is similar to previous single-system image distributed systems such as Sprite [OCD+88], Locus [PoW85], and Solaris MC [KBM+96]. Hive is implemented using techniques borrowed from these systems:

- The cells communicate primarily through remote procedure calls.

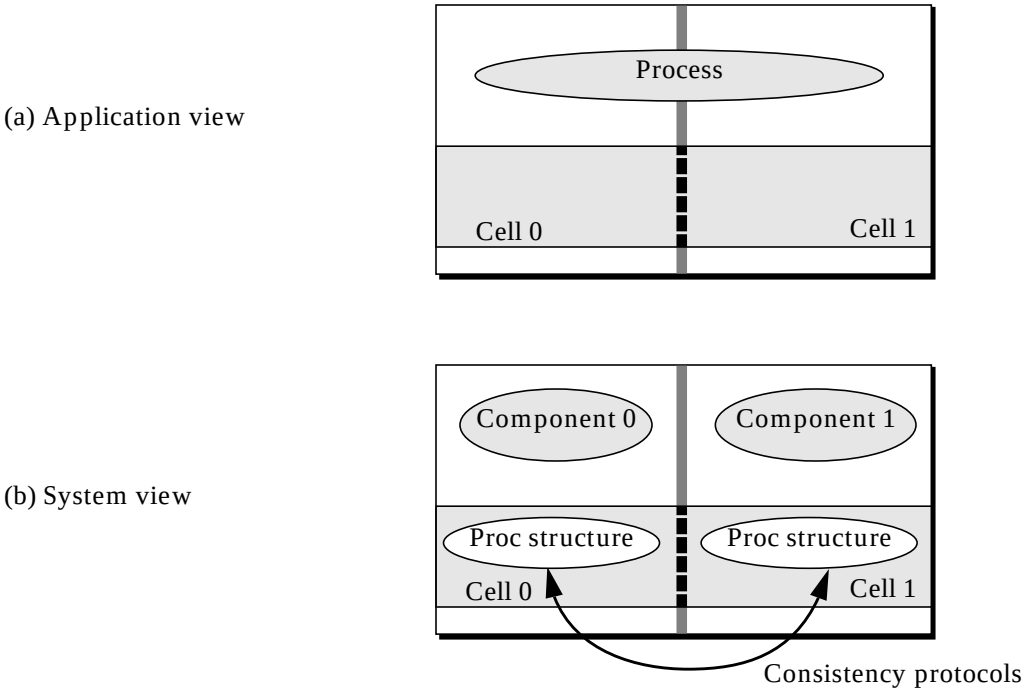


Figure 3.2. Implementation of spanning tasks.

- The cells cooperate in a shared-namespace distributed file system with each acting as both client and server.
- Process management is distributed so users and applications see a single-system image. When a process forks, the new child may be created automatically on a different cell. Operations related to the interaction between processes are extended to communicate with remote cells when needed. For example, a cell sending a signal to a process or process group may send messages to other cells in order to complete the operation.

The major difference between Hive and previous systems at the distributed system level is the support for spanning tasks. To implement spanning tasks, each cell runs a separate local process containing the threads that are local to that cell (Figure 3.2). The cells keep shared process state such as the address space map consistent among the component processes of the spanning task.

To optimize the implementation of spanning tasks, Hive sets the minimal essential dependency set of a process to be the cells where the threads of that processes are running. This enables Hive to use shared memory across cell boundaries for performance and to migrate process state between cells where desirable, since a thread need not continue running after the failure of other threads in that process. This does not appear to be an excessive limitation from the application perspective;

an application designed to survive loss of some of its threads can be implemented as multiple processes that map shared memory.

3.3.2 Cell isolation

Cell isolation is the challenge of preventing corruption in one cell from damaging other cells. There are three channels by which a fault in one cell can damage another cell: by sending a corrupt RPC request or reply, by providing corrupt data or errors to direct remote reads, or by causing wild writes.

- *Defense against corrupt messages:* Each cell sanity-checks all information received from other cells and sets timeouts whenever waiting for a reply. Experience with previous distributed systems shows that this approach provides excellent isolation, even though it does not defend against Byzantine faults.
- *Defense against corrupt remote reads:* It is the reading cell's responsibility to defend itself against deadlocking or crashing despite such problems as invalid pointers, linked data structures that contain infinite loops, or data values that change in the middle of an operation. This is implemented with a simple *careful reference protocol* that includes checks for the various possible error conditions. Once the data has been safely read, cell isolation is provided by sanity checks exactly as in the case of receiving message data.
- *Defense against wild writes:* Cells never write to each other's internal data structures, as this would make cell isolation extremely difficult. This is enforced by using the firewall to protect kernel code and data against remote writes. However, cells frequently write to each other's user-level pages since pages can be shared by processes running on different cells. This creates the possibility that wild writes due to a software error in one cell can corrupt user data read by an application on another cell.

The sanity checks used to defend against corrupt messages and data read through the careful reference protocol are standard mechanisms used by other distributed systems. The novel cell isolation mechanism in Hive is the *wild write defense* used to prevent applications from being corrupted by software errors in the kernel of a different cell.

3.3.3 Wild write defense

The wild write defense has four components: firewall management, preemptive discard, fast error detection, and fast null recovery.

Firewall management: The system seeks to minimize the number of cells with write permission to each page. This is a nontrivial problem because protection changes can be expensive. Revoking

write permission previously granted to a cell requires communication with the nodes of that cell, to ensure that all cached lines have been returned to memory. This cost makes it too expensive to guarantee that only the cells actively using a page have write permission. Instead, policies in the file system and virtual memory system reduce the vulnerability of pages as much as possible without causing an excessive number of permission changes.

However, each cell guarantees that user-level pages used only by processes local to the cell are not writable from outside the cell. This ensures that processes which use resources from only one cell are not vulnerable to errors in other cells. If cells are sized large enough at boot time, Hive will not need to share memory across cell boundaries for load-balancing reasons, so most small processes should receive the best possible fault containment from the system.

Preemptive discard: The system discards pages writable by a failed cell when a software error is detected. This is called the preemptive discard policy. It reduces the chance of application corruption by preventing corrupted pages from being read by applications or written to disk.

If a page is clean with respect to disk or other backing store, discarding it is transparent to applications. If the page is dirty, Hive gives an I/O error to applications that subsequently try to access the discarded data. Naive applications will simply exit, but sophisticated applications can field the error and recover.

Unfortunately, the preemptive discard policy does not prevent all user-visible data integrity violations caused by wild writes. Corrupt data might be used before the software error is detected, or a faulty cell might corrupt a page and give up its write permission before the error is detected, in which case the corrupted page will not be discarded.

This problem appears to be fundamental to any use of hardware shared memory across fault containment boundaries, because it is impossible to guarantee that software errors are detected immediately. The only way to prevent all data integrity violations is to avoid write-sharing user pages across cell boundaries. Giving up write-shared pages would give up one of the main performance advantages of a shared-memory multiprocessor.

However, there are several ways to reduce the probability of data integrity violations. The designer could reduce the probability of wild writes by rewriting the kernel in a type-safe language or by using a microkernel to reduce the amount of code with direct access to physical memory. These mechanisms are beneficial but not complete; for example, misprogramming a TLB entry or DMA request would bypass the type checks done by a compiler. Hive reduces the probability of data integrity violations with a more general mechanism.

Fast error detection: Hive strives to detect software errors quickly and thereby shorten the time window within which corrupt data might be used. This has the secondary benefit of reducing the delay experienced by users or applications stalled because of the error.

Error detection is a well-studied problem in the context of distributed systems. Given a non-Byzantine error model, the primary challenge is that any algorithm that reports an error might itself be in error. If one cell could declare that another had failed and cause it to be rebooted, an erroneous cell that mistakenly concluded that other cells were corrupt could destroy a large fraction of the system.

Hive uses a two-part solution. First, cells monitor each other during normal operation with a number of heuristic checks. The checks include timeouts and sanity checks on messages, sanity checks and type tag checks on remote reads, and periodic active probes that read the internal state of other cells through shared memory. A failed check provides a hint that triggers recovery immediately.

Second, consensus among the surviving cells is required to reboot a failed cell. When a hint alert is broadcast, all cells temporarily suspend processes running at user level and run a fault-tolerant distributed agreement algorithm. This algorithm is fault-tolerant in that agreement is reached correctly even if further cells fail while it is running. If the surviving cells agree that a cell has failed, user processes remain suspended until the system has been restored to a consistent state and all potentially corrupt pages have been discarded.

This approach ensures that the window of vulnerability to wild writes lasts only until the first check fails and the agreement process runs, assuming that the software error is correctly confirmed by the agreement algorithm. The window of vulnerability can be reduced by increasing the frequency of checks during normal operation. The frequency of checks performed is a tradeoff between fault containment and performance.

Fast null recovery: The final component of the wild write defense is a fast *null recovery*. Null recovery occurs when a error check triggers but no error has occurred. The system reaches agreement that no cells have failed and returns to normal operation.

Fast null recovery supports the wild write defense by making it acceptable to bias the error checks to give false alarms rather than false negatives. A bias towards false alarms is required because the checks only provide hints that an error may have occurred. For example, an active probe reading the internal state of another cell cannot synchronize with the processes that might be modifying that state, so it might observe a transient situation that mimics an error condition.

Given a fast null recovery such that false alarms have low performance impact, the developer of the system can iteratively reduce the chance of data integrity violations by adding checks targeted to each new software error that is observed.

3.3.4 Error recovery

Assuming that cell isolation is successful, error recovery in Hive is essentially the same problem faced and solved by previous single-system image distributed systems. Hive uses standard techniques to ensure that operations such as changing process groups, sending signals, and modifying files have exactly-once semantics.

Processes executing in the kernel are allowed to continue executing while error detection, preemptive discard, and error recovery are operating. This makes certain aspects of recovery more difficult; for example, careful design is needed to cope with external RPC requests arriving at a cell while recovery is running. However, this approach has the significant advantage that the recovery algorithms can acquire kernel locks and can assume normal consistency of the data structures protected by those locks.

The design of the error recovery algorithms values simplicity over speed. Although null recoveries are frequent, actual errors are assumed to be rare and therefore user-visible recovery pause times are acceptable.

3.3.5 Resource sharing policies

I now turn from features designed to improve reliability to features that provide resource sharing. To compete in performance and efficiency with existing SMP operating systems, Hive must share resources across cell boundaries much more tightly than was necessary in previous distributed systems. Both the policies and mechanisms for resource sharing raise challenging problems.

Hive separates the policy and mechanisms for intercell resource sharing. The mechanisms are implemented through the cooperation of the various kernels, but the policy is implemented outside the kernels, in a user-level spanning task called Wax (Figure 3.3).

Justification for Wax: Wax addresses a problem faced by previous distributed systems, which were limited to two unattractive resource management strategies. Resource management can be distributed, in which case each kernel must make decisions based on an incomplete view of the global state. Alternatively, it can be centralized, in which case the kernel running the policy module can become a performance bottleneck, and the policy module has difficulty responding to rapid changes in the system.

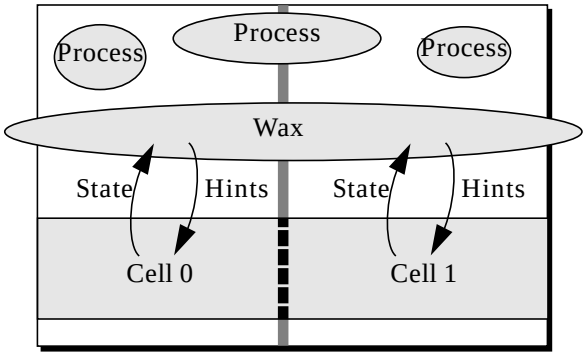


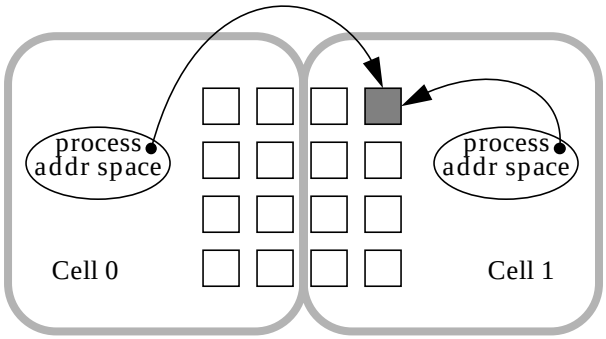
Figure 3.3. Intercell optimization using Wax.

Wax takes advantage of shared memory and the support for spanning tasks to provide efficient intercell resource management. Wax has a complete, up-to-date view of the system state but is not limited to running on a single cell. Unlike policy modules running at kernel level, which must carefully maintain fault containment, the threads of Wax running on different cells can synchronize with each other using standard locks and nonblocking data structures. This enables efficient resource management decisions.

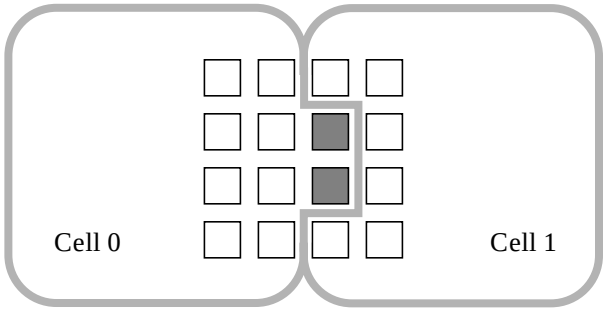
Moving intercell resource management out of the kernels is acceptable because in Hive, unlike in previous distributed systems, the resources of each cell belong to the system as a whole rather than to some local user. Making the correct tradeoff between local and remote requests requires a global view of the system state, which is available only to Wax. Each cell is responsible only for maintaining its internal correctness (for example, by preserving enough local free memory to avoid deadlock) and for optimizing performance within the resources it has been allocated by Wax.

Implementation of Wax: Despite its special privileges, Wax is not a special kind of process. It is a spanning task with threads running on all cells, so it is terminated whenever any cell fails. The recovery process starts a new incarnation of Wax which forks to all cells and rebuilds its picture of the system state from scratch. This avoids the considerable complexity of trying to recover consistency of Wax's internal data structures after they are damaged by a cell failure. The system may become unbalanced and run more slowly while the policy modules in Wax restart, but this is acceptable because cell failures are assumed to be rare.

Wax does not weaken the fault containment boundaries between cells. Each cell protects itself by sanity-checking the inputs it receives from Wax. Also, operations required for system correctness



(a) Logical-level sharing of data pages



(b) Physical-level sharing of page frames

Figure 3.4. Types of memory sharing across cell boundaries.

are handled directly through RPCs rather than delegated to Wax. Thus if Wax is damaged by a faulty cell it can hurt system performance but not correctness.

To make this work, cells do not block waiting for Wax to make decisions such as which remote cell to allocate memory from. Instead, Wax periodically modifies the internal tables in each cell used by the resource management routines. Cells provide state updates to Wax by exporting data structures for it to read and sending it signals when action appears necessary.

3.3.6 Resource sharing mechanisms

Given appropriate global policy support from Wax, the remaining challenge is to make resource sharing efficient. The resources that need to be shared particularly efficiently across cell boundaries are memory, I/O devices, and processors.

Memory sharing: Hive provides two memory sharing mechanisms (Figure 3.4). One type, called *logical-level sharing,* enables processes on different cells to share data. The other, called *physical-level sharing,* allows free memory pages to flow to cells where demand is high.

Logical-level sharing is implemented by the virtual memory system and the file system, which ensure that any two processes that open or map the same file can share the same cached pages of that file in memory. This supports efficient use of the file cache, a performance-critical resource for many workloads, and allows processes such as those forked from the same parent to share data pages efficiently across cell boundaries.

Physical-level sharing is implemented by the memory allocation modules of the different cells. When a cell is short on memory, Wax may direct it to borrow a set of pages from some other cell. The allocation module of the original owner moves the pages to a reserved list and ignores them until Wax directs the borrower to return them or the borrower fails. This mechanism supports load-balancing of memory, preventing a situation where a cell starts paging even though there is free memory elsewhere in the system.

I/O device sharing: Sharing of I/O devices is implemented just as in other distributed systems. A request to access a remote I/O device is forwarded to the cell that physically owns the device. That cell then executes the request and returns the result to the client cell. Hive avoids the extra data copy that this design would normally require, to or from the memory of the cell that owns the device, by using the intercell memory sharing mechanisms to allow devices to issue DMA accesses directly to the memory of the client cell.

Processor sharing: Processor sharing in Hive is different from previous distributed systems. In systems without shared memory, the only option for load-balancing the system is to migrate processes completely, which can be slow. Hive takes advantage of the spanning task mechanism instead (Figure 3.5). A new thread is created on another cell that shares the process context of the original process, the thread on the original cell is suspended, and its state is moved to the new thread.

After a new thread for the process has been created on a given cell, which can be done without pausing the process, execution can be moved to or from that cell much faster than traditional migration because much less state must be transferred. Migrating a process still creates memory system costs due to the overheads of remotely accessing the working set of the process, but these costs are comparable to those created by moving a process between the processors of an SMP kernel. Once the process is running in its new location, the normal page migration and replication mechanisms activate to reduce its memory system costs.

3.3.7 Cell size tradeoffs

In the resource sharing mechanisms just described, the cell that owns the resource retains ownership permanently. Choosing to do resource sharing this way (*static cell size*) was an

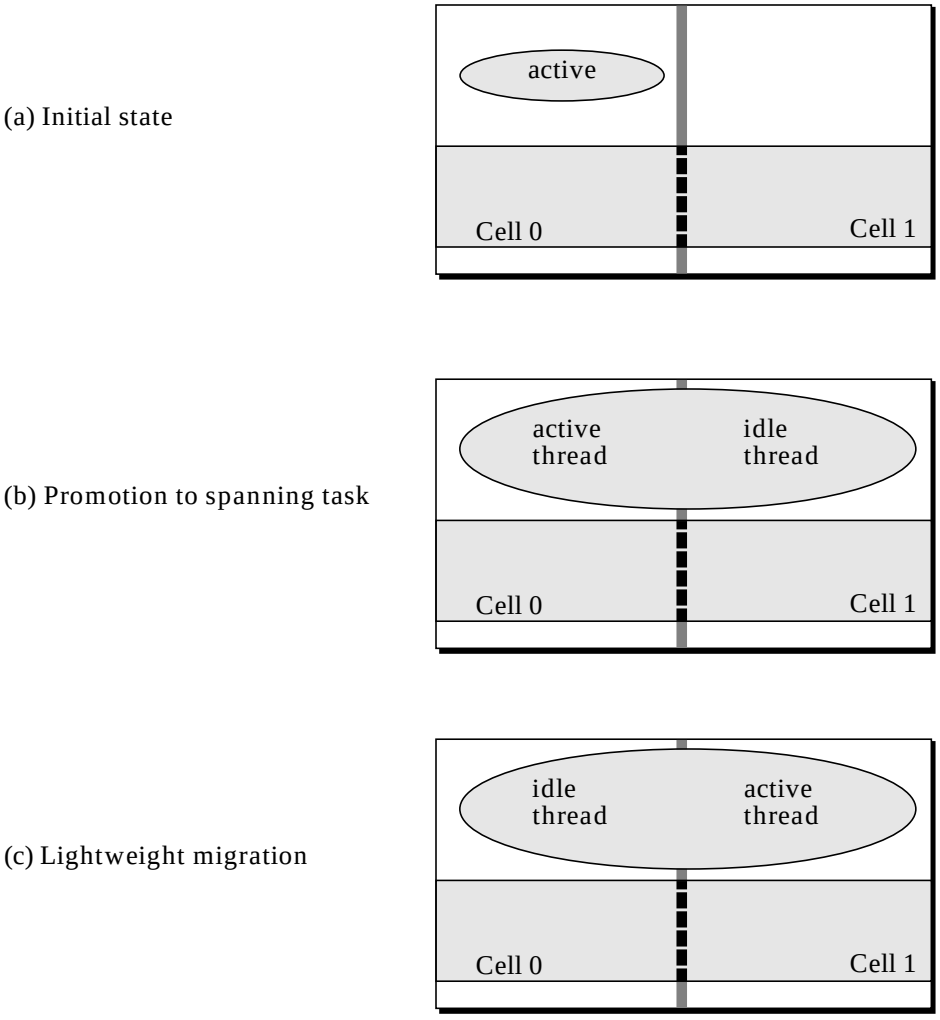


Figure 3.5. Lightweight process migration using spanning tasks.

important decision made early in the design of the system. It would also be possible to transfer ownership of nodes between cells, moving processors, I/O devices, and memory simultaneously, in effect allowing cells to grow and shrink as the system runs (*dynamic cell size*).

Static cell size is attractive because it simplifies the implementation. Hardware resources such as processors and I/O devices that each cell manages are all present at boot time, so the code in the base SMP operating system that discovers and manages these resources need not be changed. Cell boundaries do not move dynamically, so the hardware implementation of the memory fault model is simplified.

In exchange for increased complexity, dynamic cell size offers the advantage of potentially reduced inter-cell communication rates. It seems likely that kernel overheads would be reduced if

the system could resize cells to match the applications that run in them. In particular, dynamic cell size might be able to eliminate most of the performance overheads caused by running large applications as spanning tasks. It might even be possible to significantly simplify the system by eliminating support for spanning tasks.

The deciding factor in choosing static cell size for Hive was the assumption that the workload might contain a mix of very large and small processes yet still require fault containment for the small processes (Section 3.2.1). Given this assumption, spanning tasks would still be required and would still have to be implemented efficiently even if cell size were dynamic, so the implementation complexity of moving nodes between cells does not appear to be justified.

Given static cell size, there are obviously workload-dependent tradeoffs between performance and reliability when choosing cell size at boot time. As cell size increases, resource allocation efficiency improves and intercell communication costs drop, but kernel memory system and synchronization costs increase. System reliability also initially increases as cell size increases because more processes become local (Section 3.3.3), but then decreases as each process becomes vulnerable to a larger fraction of the system. The turnaround point for reliability is determined by average process size in the workload and the relative frequency of hardware and software errors. Hive leaves it to the system administrator to balance these factors and choose the appropriate cell size at boot time for each site's workload and requirements.

3.3.8 Summary and other features

The Hive architecture just described has three primary features. First, it has an internal distributed system, similar to other distributed systems, that maintains a single-system image and relies primarily on RPCs for communication. Second, it provides cell isolation for the distributed system using a wild write defense that consists of the firewall, the preemptive discard policy, fast error detection and fast null recovery. Finally, it seeks to achieve performance competitive to SMP operating systems through physical and logical-level memory sharing and lightweight process migration based on spanning tasks, all managed by global policy modules in Wax.

In addition to these primary features, the Hive architecture has several secondary features that are important for the reliability and scalability of the system. Implementations of these features have not yet been designed but are clearly required.

- *Graceful shutdown:* A key advantage of a multicellular kernel for commercial installations is that parts of the system can be shut down transparently for preventive maintenance. Graceful shutdown requires an additional process migration mechanism that leaves no dependencies

on the home cell, unlike the lightweight migration mechanism based on spanning tasks. The complete-migration operation is not performance-critical.

- *Page migration and replication:* To improve locality on a CC-NUMA machine, the virtual memory system must be able to replicate read-shared pages to multiple cells or migrate write-shared pages among cells. This is independent of the replication and migration that occur within a cell.

The most important architectural feature not yet designed is the file system. The file system for a multicellular kernel must provide a globally shared namespace, replication of critical directories and files, striping and software RAID, and takeover of dual-ported disks by a backup cell after the primary cell fails. It must do all this while tolerating the loss of any cells in the system. Achieving these goals while preserving competitive performance is a substantial research challenge.

3.4 Error recovery at lower levels

Hive depends only on the firewall and the memory fault model exported by the memory system (described in detail in Section 6.3). However, for completeness, this section surveys how the required error recovery functionality is provided by the lower levels of the fault containment stack in FLASH.

Memory system level: Hardware fault containment requires recovering the memory system to a consistent state after an error. FLASH simplifies the challenge of memory system recovery by avoiding the need to determine the exact state of the system after an error. Consider that any event that damages the interconnect may damage arbitrary cache-coherence operations that are incomplete at the time of failure, for example by deleting random network packets. The possibility of further errors during recovery makes it even more difficult to achieve consensus about the state of the system.

Instead, FLASH recovers the consistency of the memory system by doing a warm reset. This is called the *sledgehammer* algorithm. After an error is detected and network recovery has completed, all nodes flush their caches, the network is drained, then all directory entries are reset to show the lines as clean in memory and uncached. Because the caches were flushed, a line found to be marked dirty during the reset phase must have been lost in the fault. An error condition is marked in the directory entry for the line, ensuring that the operating system will receive an error the next time any processor tries to access the line. If further hardware errors occur while recovery is in progress, the sledgehammer algorithm simply restarts.

Memory system error recovery is complicated enough that it is desirable to implement it on the main compute processors rather than in protocol microcode. The system reserves a non-maskable interrupt for this purpose. When issued by the memory system, the interrupt forces the processor to begin executing code in uncached mode from a special range of memory not modifiable by the operating system.

Network level: When an error occurs that prevents delivery of packets, such as a power failure, node halt, or broken link, the entire interconnect can become congested as undeliverable packets fill queues in the routers. Recovering from this condition creates the key challenge of dropping only those packets that are destined to or routed through the failed portion of the machine.

The network level recovery algorithm starts by diagnosing which portions of the machine have failed. Given this information, it sets a flag in the routers bordering that region so that packets attempting to enter the failed region will be discarded. This unclogs the network, since all packets will either be discarded or reach their destinations. When the network has drained, the recovery algorithm modifies the network routing tables reconnect the surviving nodes while avoiding the failed region.

To support communication while the network is clogged, FLASH reserves two virtual lanes through the routers for use by network-level recovery. Network recovery is complicated enough that FLASH implements it with code that runs on the main compute processor, in the same way as memory system recovery.

More information: See [TBG+] for more details on memory system and network recovery and [Gal96] for normal operation of the network and data link levels.

Chapter 4
Hive prototype

The architecture described in the previous chapter is a system design. This chapter describes the current Hive prototype that implements the design and some of FLASH features that support it.

Section 4.1 surveys the implementation status of the prototype. The remaining sections of the chapter describe three aspects of the system that are necessary to evaluate the results of the experiments but are otherwise outside the focus of the dissertation: the partition of the multiprocessor into multiple cells, the RPC subsystem, and the file system. I defer discussion of fault containment and memory sharing to Chapter 6 and Chapter 7 where their implementation is described in detail.

4.1 Implementation status

The Hive prototype, implemented as an extensive modification of SGI IRIX 5.2, does not provide all of the architectural features described in the last chapter.

- The single-system image is limited. System V streams (which includes interprocess pipes and network connections), system V shared memory, and direct device access are not supported across cell boundaries.
- Wax has not been implemented. Each cell periodically reads data structures exported by other cells to make memory sharing decisions, and uses a simple rotor to decide which cell to fork to next.
- Spanning tasks are not implemented. The parallel applications in the workloads run in a mode where they fork independent subprocesses that share memory.
- The intercell file system is a straightforward port of NFS and does not provide a true shared name space or file replication.

The net effect of these missing features is that the performance comparison between Hive and IRIX is not conclusive. Many performance problems may remain undetected. However, the prototype is sufficient to demonstrate solutions to the fundamental challenges of fault containment in a shared-memory environment and memory sharing across cell boundaries.

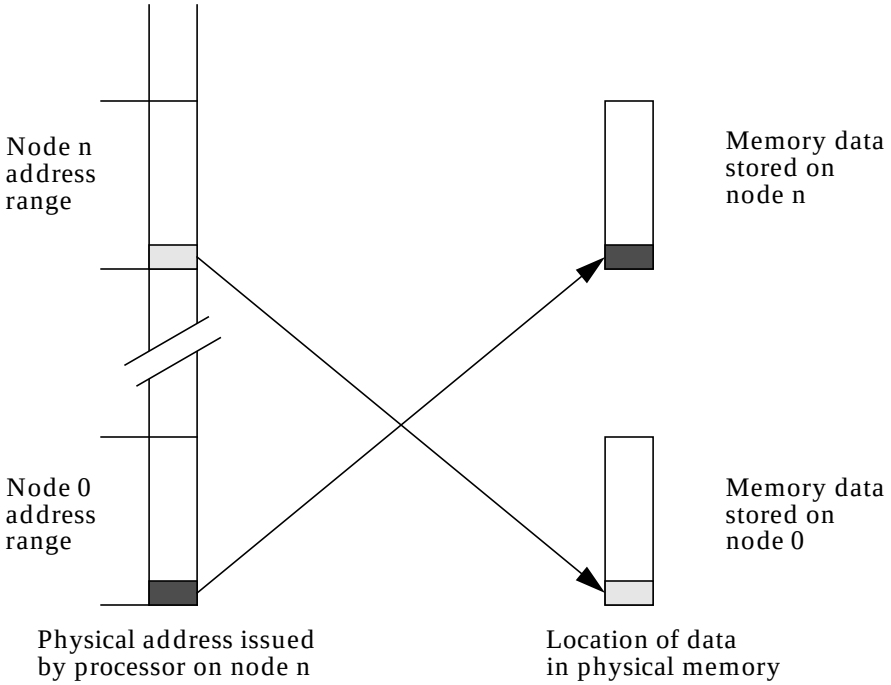


Figure 4.1. FLASH remap region.

4.2 Multiple cells

There are some issues that must be resolved for a kernel to run in just part of a machine rather than owning the entire system as is the case with current SMP operating systems.

Remap region: The MIPS processor architecture has fixed addresses in low memory to which a processor jumps when an interrupt or exception occurs. To support a multicellular kernel, the hardware must remap these addresses so they are cell-local or node-local. Otherwise, a hardware or software error on node zero could cause the entire system to fail.

FLASH provides a *remap region* feature for this purpose. Logic in the MAGIC chip processor interface modifies addresses received from the processor before they reach the memory system proper, and performs the inverse mapping on addresses sent from the memory system to the processor. The remap region can be set to any power-of-two number of bytes. When the remap region is set to *b* bytes, the lower *b* bytes of the physical address space are exchanged with the lower *b* bytes of the local node's memory (Figure 4.1).

Kernel relocation: The base IRIX design accesses kernel code directly through physical memory addresses. For simplicity Hive retains this design. The kernel build for Hive generates multiple versions of the kernel executable, each with a different starting address.

In a production-quality system where it is desirable to store only one copy of the kernel code on disk, there are several ways to adapt the code to multiple starting addresses. The boot loader could relocate the kernel depending on where in memory it is placed. The kernel could be relinked to access its code and data through virtual memory addresses. A third option would be to place the kernel code in the remap region, but this would waste memory since the remap region references node-local rather than cell-local memory.

Using virtual memory addresses appears to be the best option, especially since it is a necessary step towards running most of the kernel in a reduced privilege state (such as MIPS supervisor mode) that would deny direct access to physical memory and hence reduce the chance of wild writes. It should be possible to use a wired superpage TLB entry to eliminate the performance impact and implementation complexity of TLB misses to kernel code.

Booting: The initial kernel booted on the system does full resource exploration, then partitions the system into the number of cells specified on its boot line. It boots up to single-user mode in the partition that it was compiled for and triggers recovery. The normal recovery algorithms run, determine that no other cells are alive, elect the current cell master, and boot and integrate the remainder of the cells. All slave cells boot in parallel so this approach should be efficient on a large machine.

4.3 RPC subsystem

RPC latency is even more critical in Hive than in previous distributed systems because of the tight resource sharing across cell boundaries expected by applications. FLASH provides a hardware message primitive called *short interprocessor sends* (SIPS) that is designed to support Hive RPC.

4.3.1 Hardware support

The SIPS facility delivers 128 bytes from one processor to a receive queue inside the MAGIC chip on another node with overflow to the node's main memory. Minimum latency between nearest neighbors is 1.41 μsec, from the first uncached write on the sending processor until the interrupt is delivered to the receiving processor.[1] This is quite fast, comparable in speed to the highly-tuned hardware support for cache misses, which take about 1 μsec if directed to the memory of a neighboring node. If the receiver is waiting for the message, for example with the reply half of an RPC, it can issue a cache miss in advance to a special address that stalls in MAGIC until the message arrives. With this *receive posting optimization* that eliminates interrupt dispatch overhead, total latency from first word sent until last word received in the receiver's cache is 1.74 μsec.

1. Performance is measured using the experimental setup described in the next chapter.

RPCs could be implemented on top of normal cache-coherent memory reads and writes. Therefore it is important to justify why custom hardware is needed. There are four design points that must be considered:

- *No custom support:* The shared memory and interprocessor interrupts (IPIs) provided by a standard multiprocessor are sufficient to support an RPC implementation. The sending processor places the message in a producer-consumer buffer and sends an IPI to the receiving processor.

  Fault containment problems limit the usefulness of this design. There cannot be one queue per receiver, as granting global write permission to shared queues would allow a faulty cell to corrupt any message in the system. Therefore the receiving cell would have to poll per-sender queues to determine which cell sent the IPI, which is an inherently non-scalable approach.

- *Single-word messages:* The IPI hardware can be extended to transmit a number of argument bits to the receiving processor. If just a few bits are transmitted, they can be used to specify the sending processor or cell and thus eliminate the polling problem. If the IPI carries more bits, certain common messages such as ACK can be directly encoded.

  When a processor receives a single-word message, it will frequently need to read more argument words from the sender. Thus the expected latency of communication for a single-word message is three trips from sender to receiver: one for the initial message, then another round trip as the receiver cache-misses on the argument words. (This assumes an optimized design where senders maintain per-receiver outgoing queues so the receiver need not fetch queue metadata to learn where the next message for it is stored.) However, this performance cost does not lead to much savings in hardware cost compared to cache-line messages, since similar complexity is required to send messages, queue messages, and handle overflows in the two designs.

- *Cache-line messages:* The IPI mechanism can be extended to transmit a cache-line worth of data to the receiving processor. This is the design chosen for the SIPS mechanism of FLASH. Cache-line messages are efficient since the memory system is optimized to move data in cache-line sized units. For example, after the interrupt is delivered, the receiving processor can force a cache miss in order to retrieve the data efficiently. Critical-word-first restart provided by the processor hardware enables the operating system to begin dispatching the message while the tail of the cache line is still flowing across the processor bus.

- *Block move:* The memory system can provide the capability to efficiently copy a number of cache lines from one place to another. This is useful if the system runs programs coded to one

of the message-passing interface standards such as PVM or MPI. However, there are few opportunities to apply block moves to optimize kernel-level RPC performance, since the key problem is low-latency transfer of control rather than high-bandwidth transfer of data.

The cache line size of FLASH is 128 bytes. At this size for a SIPS message, a high fraction of RPC requests and replies require no additional cache misses because all the argument and result data fits in the initial message. Therefore the expected latency of a message send is just one trip from sender to receiver.

4.3.2 RPC architecture

The Hive RPC subsystem is streamlined to minimize the overhead added to the efficient SIPS primitive. Timeouts are loose and checked infrequently since their only purpose is to detect hardware faults and server cell failure, which are assumed to be rare. No message fragmentation or reassembly is required because any data beyond a cache line can be sent by reference, although the careful reference protocol must then be used to access it.

Interrupt-level RPCs: In the base RPC system, all requests are serviced immediately at interrupt level by the receiving cell. After sending a request, the client processor posts a receive and waits for the reply rather than context-switching to a different process. If the reply is delayed the client processor will eventually time out and block the requesting process. This is transparent to the client of the RPC and should only occur if there is a server failure or significant contention in the memory system or server cell.

Requiring the client processor to wait in the common case simplifies the interrupt-level RPC outbound path since each processor can only have one outstanding request at a time. If a timeout occurs causing the processor to context-switch, then some other process attempts to send an interrupt-level RPC, the second process will block until the previous RPC completes or the system declares a fault and enters recovery.

The requirement that the sender be able to block creates the single most important constraint of the interrupt-level RPC mechanism: interrupt handlers cannot send RPCs. This affects the design of higher levels of the system. For example, the handlers for interrupt-level RPCs cannot send nested RPCs, and events stimulated by timeouts must be queued for server processes rather than sending RPCs directly from the clock interrupt handler.

Process-level RPCs: A more general mechanism is layered on top of the base interrupt-level RPC mechanism. Process-level RPCs provide a queuing service and server process pool to handle longer-latency requests, including any that cause I/O or must block on kernel semaphores. A process-level request is structured as an initial interrupt-level RPC that launches the operation,

then a completion interrupt-level RPC sent from the server back to the client to return the result. Context switch and scheduler latencies make process-level RPCs significantly more expensive than interrupt-level RPCs.

As an optimization, each process-level handler can be paired with a corresponding interrupt-level handler. The interrupt-level handler makes an initial best-effort attempt, only scheduling the server process for execution if it cannot satisfy the request.

Deadlock avoidance: To avoid deadlock for interrupt-level RPCs, FLASH provides two SIPS receive queues in each node, one for requests and one for replies. The reply queue raises a higher-priority interrupt than the request queue.

To avoid deadlock for process-level RPCs, the processes in the server pool are organized into levels starting from one. All non-server processes are at level zero. A cell that receives a process-level request from a process at level N places it in the queue for a level N+1 server process. The deepest RPC call chain in the system currently reaches level two.

4.3.3 Evaluation

The Hive RPC subsystem achieves its low-latency goals. The latency of a null interrupt-level RPC sent to an idle neighboring node is 7.2 μsec. A process-level RPC handled by a server process has a minimum 41.6 μsec latency, of which 20.4 μsec comes from the scheduler, context switch mechanism, and implementation of kernel semaphores.

There are two caveats to the excellent speed measured for interrupt-level RPCs. First, the current implementation does not implement a full retransmission and duplicate suppression protocol for recovery from network and physical-level errors or receive queue overruns that cause loss of SIPS. It correctly handles some but not all types of packet loss. Second, the measured time is short enough that the measurement might be affected by the use of an R4000 processor model for the simulations rather than the R10000 processor that will be installed in FLASH.

The Hive RPC subsystem and the SIPS mechanism closely resemble other low-latency message delivery systems such as active messages [vEC+92], U-Net [vEB+95], and FLIPC [BSS+96]. The key difference from these previous systems is that SIPS is implemented directly by the memory controller and thus can take advantage of optimizations unavailable to the other systems. One example is the receive posting optimization, in which a processor waiting for a message issues the receive cache miss in advance and the cache miss stalls in the memory controller; eventually the controller either returns an incoming message or returns a timeout. This approach is substantially faster than the polling or interrupt-based approaches that are required when the memory controller is not part of the implementation.

4.4 File system

The intercell file system in Hive is called cell-NFS. It is a variant of the standard NFS distributed file system [SGK+85]. This file system provides functionality sufficient for the prototype but is not designed to provide the functionality needed for a production-quality multicellular kernel.

Name space: In cell-NFS each cell is the file server for a portion of the file system name space. When a cell fails, that portion of the name space becomes inaccessible. Each cell has a local root directory and accesses the parts of the name space served by other cells through mount points in that directory.

The names of the mount points are hardwired into the kernel in two places. First, the remote cell's root directory is automatically mounted when the corresponding mount point is accessed for the first time. Second, when a process forks a child on another cell, the child process executes a `chroot` to the mount point corresponding to the parent cell, in order to maintain a consistent view of the file system name space.

The initial Hive design did not include automounting, but it turned out to be necessary. A newly-booted cell becomes a target for load-balancing by other cells that attempt to fork processes to it as soon as it is integrated into the system. The fork will fail if the root directory of the parent cell is not mounted. However, it is not possible to preemptively mount all remote cells during boot, because multiple cells may be booting at the same time. Therefore the fork attempt must automount the root directory of the parent cell.

Universal file identifiers: The vnode pointer used as the internal name for a file in the base IRIX code is insufficient for Hive. The problem arises when a process forks a child on another cell. The open files of that process must be reopened on the new cell, but the vnode pointer on the parent cell is local to that cell, so some other name is needed to identify the file to the file server cell in the reopen request. The external file name used to open the file initially cannot be used because it may no longer be valid.

Cell-NFS provides a *universal file identifier* to solve this problem. The universal file identifier contains a file server cell ID and vnode pointer on that cell. This is sufficient because the identifier's duration is limited. It need only be valid while some process on the system has the file open. The file system ensures that the vnode on the file server cell is not deallocated as long as the file is in use. The universal file identifier is stored in each local vnode that represents a remote file.

Data access: When a process reads or writes to a file whose file server cell is remote, cell-NFS behaves just like NFS with one important exception. Rather than returning a copy of the requested

page from the file server, cell-NFS returns the address at which the requested data is stored in memory. It uses the memory sharing mechanisms provided by the virtual memory system to grant remote read or write access to the memory data. Because the standard NFS writeback mechanism is not used, the write-through on writeback design that creates a high number of synchronous accesses for NFS does not affect cell-NFS.

Data consistency: Cell-NFS inherits its metadata and data consistency mechanisms from NFS. In NFS, metadata is cached without possibility of invalidation for up to 3 seconds and there is no data cache coherence. This approach is acceptable for NFS because files are rarely write-shared in the distributed environments where it is used. However, in a multicellular kernel files are frequently write-shared by processes running on different cells, so better data consistency and metadata consistency mechanisms are needed.

Returning references to data pages rather than copies of those pages to clients solves the data consistency problem for cell-NFS, since the cache-coherence hardware maintains consistency. The metadata consistency problem remains unsolved in Hive. The only application in the experiments potentially sensitive to this problem is pmake, in which the console outputs of the parallel compilations can be written in append mode to a shared file. To avoid the problem, parallel make is run in a mode where the parallel compilation processes output their logs to individual temporary files. The master process polls these files and copies the output to the shared log file.

4.5 Summary

The three mechanisms just described are implemented to different levels of completeness. The support for multiple cells, including the remap region, is complete enough that a production-quality multicellular kernel could use this approach, as long as it is acceptable to store multiple kernel executables on disk. The RPC subsystem and the SIPS primitive that support it are fairly fully implemented, lacking only a completed retransmission protocol and whatever hardware support is required to make it efficient. In contrast, the file system does not provide most of the features required for scalable performance and fault containment.

The limitations of these features and the missing single-system image mechanisms, notably spanning tasks and Wax, prevent this dissertation from making any definitive conclusions about the performance of multicellular kernels. However, the implementation is complete enough for experiments to demonstrate that fault containment is possible inside a shared-memory system and that memory can be shared at both application and kernel levels across fault containment boundaries. The next chapter describes how the experiments are performed; the following two chapters describe the implementation of fault containment and memory sharing in detail.

Chapter 5
Experimental setup

This chapter describes the setup for the experiments in the dissertation. Because the FLASH machine is not yet operational, all experiments use the SimOS machine simulation environment.

The first section of the chapter describes SimOS. The second gives information about the performance experiments. The third describes the reliability experiments. The final section analyzes how using simulation affects the experimental results.

5.1 The SimOS simulation environment

SimOS [RHW+95, WiR96, RBD+] is a machine simulation environment that simulates the hardware of uniprocessor and multiprocessor computer systems in enough detail to boot, run, and study a commercial operating system.[1] Specifically, SimOS provides simulators of processors, caches, memory systems, and a number of different I/O devices including SCSI disks, ethernet interfaces, and a console.

There are a number of unique features in SimOS that make detailed workload and kernel studies possible. These include multiple processor simulators, checkpoints, and annotations.

- *Multiple processor simulators:* In addition to the configurable cache and memory system parameters typically found in simulation environments, SimOS supports a range of compatible processor simulators. Each simulator has its own speed-detail trade-off. The highest speed simulator, called *Embra*, uses binary-to-binary translation techniques. This fast mode is capable of executing workloads less than 10 times slower than the underlying host machine. SimOS also contains two more-detailed processor simulators that are orders of magnitude slower than Embra. The processor simulator called *Mipsy* models a static-pipeline processor (one instruction per cycle, one outstanding cache miss at a time) while the simulator called *MXS* models a dynamic-pipeline processor (out-of-order and speculative execution, multiple outstanding cache misses).

1. The description in Section 5.1 is largely taken from [RHW+95].

- *Checkpoints:* SimOS can save the entire state of the simulated machine at any time during execution. This saved state, which includes the contents of all registers, main memory, and I/O devices, can then be restored at a later time. Checkpoints make it possible to restart each experiment at the point of interest without wasting time rebooting the operating system and positioning the applications.

- *Annotations:* To better observe workload execution, SimOS supports a mechanism called annotations in which a user-specified routine is invoked whenever a particular event occurs. Most annotations are like debugger breakpoints, in that they trigger when the workload execution reaches a specified program counter address. Annotations are non-intrusive. They do not affect workload execution or timing, yet have access to the entire hardware state of the simulated machine.

Because SimOS simulates the complete hardware of the system, a variety of hardware-related statistics can be kept accurately and non-intrusively. These statistics cover instruction execution, cache misses, memory stall, interrupts, and exceptions. The simulator is also aware of the current execution mode of the processors and the current program counter. However, this does not provide information on important aspects of the operating system such as the current process id or the service currently being executed.

To further track operating system execution, SimOS provides a set of state machines (one per processor and one per process) and one pushdown automaton per processor to keep track of interrupts. These automata are driven by annotations set in various places in the kernel. For example, annotations set at the beginning and end of the kernel idle loop separate idle time from kernel execution time. Annotations in the context switch, process creation, and process exit code keep track of the current running process. Since they have access to all registers and memory of the machine, these annotations can non-intrusively determine the name and id of the currently running process. Additional annotations are set in the page fault routines, interrupt handlers, disk driver, and at all hardware exceptions. These are used to attribute kernel execution time to the service performed. Annotations at the entry and exit points of the routines that acquire and release spin locks determine the synchronization time for the system, and for each individual spin lock.

Simulator validation: Validating the accuracy of these statistics is less of an issue for this dissertation than for most other simulation-based studies, because performance is not the focus of the results. However, SimOS has been partially validated as reported in [RHW+95]. As for correctness of execution, the applications are unmodified from and generate the same output data as the code that runs on real IRIX systems, and it is difficult to imagine the kernel code successfully completing these complex workloads if SimOS did not execute it correctly.

Table 5.1. System model for performance experiments.

Component	Characteristics
8 processors	200 MHz MIPS R4000 1 CPI when no cache misses
Primary instruction cache	32 kilobytes 2-way associative 64-byte lines
Primary data cache	32 kilobytes 2-way associative 32-byte lines
Unified secondary cache	1 megabyte 2-way associative 128-byte lines
15 disks	Accurate model of HP 97560 drive [KTR94]
256 MB memory	8 FLASH nodes, 32 megabytes each
Mesh interconnect	400 MHz SPIDER routers [Gal96]
8 MAGIC controllers	100 MHz 1 megabyte direct-mapped data cache 16 kilobyte 2-way associative instruction cache

5.2 Performance experiments

The performance experiments use the hardware configuration shown in Table 5.1.

Simulation model: The experiments use the Mipsy processor model, configured with the same caches as the MIPS R10000 used in FLASH. Memory access latencies are computed using the reference memory system model developed by the FLASH hardware team, called *Flashlite* [HKO+94]. A cache miss to a line that is clean in local memory takes about 200 nsec (40 processor clocks), to a line clean in a remote memory takes at least 840 nsec (168 processor clocks), and to a line dirty in a remote processor's cache at least 1.2 μsec (240 processor clocks). These times can be increased by queueing delays in the MAGIC chips, misses in the MAGIC caches, contention in the network, and arbitration for the processor busses. Flashlite includes a protocol processor emulator that executes and times the actual protocol microcode that will run in the machine.

Workloads: The performance experiments use three workloads. In all cases the workload is run once to warm up the file cache, then run again to measure execution time and behavior.

The first workload, *pmake*, models general-purpose use of the system. It is a parallel make of 16 files from the gnuchess 4.0 distribution, compiled eight at a time. Since this is a fairly small

compilation, the files to be compiled are chosen from among the source files in the distribution to minimize the length difference between the shortest and longest source files. This avoids hiding operating system performance differences in the idle time that occurs waiting for the last compilation to complete.

The other two workloads, *ocean* and *raytrace*, are parallel scientific applications from the Splash-2 benchmark [WOT+95]. These applications stress Hive's resource sharing mechanisms. Ocean uses a 258 by 258 grid and a 1400 second interval. Raytrace uses the teapot data set and four antialias rays per pixel. Both applications are run with eight threads, in a mode where the threads are created using the `fork` system call. That is, from the perspective of Hive they run as eight separate processes that happen to share memory.

Operating system configurations: The performance baseline is provided by IRIX 5.2, modified as little as possible to port it to the FLASH hardware modeled by SimOS. The primary changes are the addition of new disk, ethernet, and keyboard device drivers, and modifications to low-level handlers to manage the FLASH interrupt architecture.

Hive's performance is measured across configurations with one, two, four, and eight cells on the eight-processor system. In each case the hardware resources are divided equally among the cells. For example, the two-cell system contains one copy of the kernel text starting at physical address 0 that manages processors 0 through 3 and memory from 0 to 128 megabytes, and another copy of the kernel text starting at physical address 128 megabytes that manages processors 4 through 7 and memory from 128 megabytes to 256 megabytes.

For debugging purposes, Hive is compiled with optimization disabled. To make the performance comparison fair, IRIX is compiled in the same way.

Summary of results: Table 5.2 shows the overall time to completion of the performance workloads on the various operating system configurations. The eight-cell Hive configuration is competitive with IRIX for pmake, slower for raytrace, and faster for ocean. Interestingly, both pmake and ocean are faster in the four-cell configuration than in the two-cell configuration, and pmake is even faster at eight cells than at four cells. These results are explained in Chapter 8 after the implementation details of Hive have been presented.

5.3 Reliability experiments

It is difficult to predict the reliability of a complex system before it has been used extensively. Furthermore, it is impossible to demonstrate complete reliability through fault and error injection

Table 5.2. Time to completion (seconds) of performance experiments.

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	4.9	4.9	6.3	5.3	4.9
raytrace	3.1	3.0	3.5	3.5	3.8
ocean	4.0	3.5	3.8	3.0	3.8

Table 5.3. System model for reliability experiments.

Component	Characteristics
4 processors	MIPS R4000, not timing accurate
7 disks	Single-cycle latency
128 MB memory	Zero latency

tests. Still, these tests can provide an indication that Hive's fault containment mechanisms are functioning correctly.

Table 5.3 summarizes the system model used for the reliability experiments. To make it possible to run a large number of tests, the reliability experiments use a smaller hardware model, a less accurate simulation mode, and a smaller workload than the performance experiments.

Simulation model: The experiments use the Embra processor simulator, which gives up timing accuracy to achieve simulation speed. The effect of the binary-translation techniques used in Embra is that from the perspective of Hive running on the simulated system, the speed of each processor varies nondeterministically over time relative to the speeds of other processors. To improve simulation speed further, both memory accesses and disk accesses complete in one cycle.

Use of this high-level system model implies that the experiments can stress software fault containment mechanisms in Hive but not the hardware fault containment mechanisms in FLASH. Since this dissertation focuses on Hive, the experiments assume that FLASH successfully implements the memory fault model.

Workload: The workload for the reliability experiments is a parallel make of six files from the same gnuchess distribution as used in the performance experiments, compiled four at a time. After the primary run in which a fault or error is injected and the make completes or is terminated, all files successfully output are checksummed to look for data corruption. Then the parallel make is executed again. The success of the second parallel make is taken as evidence that the surviving cells were not corrupted.

Operating system configuration: The fault injection experiments all use a four-cell Hive configuration. The script driving the two parallel make executions runs on cell 0 and the source files are stored on a disk attached to that cell. File system limitations make cell 0 essential for all the compile processes. Therefore, other than a few experiments performed in order to check for faults in the recovery algorithms, the reliability experiments avoid injecting faults or errors into cell 0.

Summary of results: In 1000 software and hardware error injection experiments and 1000 software fault injection experiments, there were no instances of data corruption in the output of the parallel make. In the fault injection studies, Hive has an uncontained failure rate of 4%, which indicates that its fault containment mechanisms are functioning properly. Future multicellular kernels should do significantly better than the 4% uncontained failure rate measured for Hive, because the prototype is still immature: about half of the uncontained failures come from just two software faults that are easily fixed. Chapter 6 analyzes these results after presenting the implementation of fault containment.

5.4 Impact of using simulation

Using simulation constrains the experiments significantly because it limits the size of the system that can be studied. After discussing the impact of limited system size on the results, I describe the other ways in which the differences between the execution environment of SimOS and the FLASH machine affect the performance and reliability experiments.

5.4.1 Limited system size

The four- and eight-processor systems studied are much smaller than the machines with tens to thousands of processors for which Hive is designed. This is another reason that many performance problems may remain undetected.

However, it is more difficult for a multicellular kernel to achieve competitive performance at small system sizes than at medium and large system sizes. Memory system and synchronization costs are lower, so the overheads of SMP operating systems are lower. The competitive performance seen in most of the performance experiments is therefore a promising result. Moreover, the performance experiments show trends that suggest how multicellular kernels will behave on larger systems (Chapter 8).

The reliability results are less affected by the small system sizes studied. Reliability is more affected by the number of cells than by the number of processors, and production systems as large as 128 processors may run with just four or eight cells. The primary effect of increasing the

number of processors per cell is to increase the rate of interaction between cells, which creates more opportunities for corruption in one cell to spread to another if the latency to detect software errors remains constant. However, since each interaction between cells represents an opportunity to execute a heuristic failure check, it seems likely that the expected latency will drop as the rate of interaction increases. Whether or not this conjecture proves to be correct, the demonstration that fault containment is possible in a shared-memory system carries weight irrespective of the size of the system.

5.4.2 Impact on performance experiments

There are two differences between the simulated environment used in the performance experiments and the FLASH machine that have a significant impact on the performance results.

Processor pipeline: The performance experiments use a static-pipeline processor model rather than the dynamically-scheduled pipeline of the R10000 processor in FLASH. The more accurate MXS processor model of SimOS is too slow to use for workloads as large as the ones used in the performance experiments.

The effects on measured operating system performance of using the simpler processor model while keeping the rest of the hardware configuration unchanged are examined in detail in [RHW+95]. Using a *make* workload on a uniprocessor version of IRIX 5.3, that study found that the dynamically-scheduled processor hides about 50% of the kernel level 1 cache miss time experienced by the static-pipeline processor, but only about 15% of the kernel level 2 cache miss time.

The memory system used in [RHW+95] satisfies level 2 cache misses in a uniform 300 nsec, which is substantially faster than FLASH. This suggests that changing from a static-pipeline processor to a dynamically-scheduled processor will have a minor impact on kernel level 2 cache miss time in FLASH.

If so, the primary effect that could change the relative performance measured for IRIX and the various Hive configurations is the substantial reduction in level 1 cache miss time. This effect strengthens the results of the performance experiments. As Table 5.4 shows, the percent of kernel time spent on level 1 cache misses increases steadily as the number of cells increases. A 50% reduction in this time will improve the performance of Hive more than it improves the performance of IRIX.

MAGIC caches: Implementation constraints forced the MAGIC hardware design team to switch to a less aggressive instruction cache for references made by the protocol microcode. The memory

Table 5.4. Percent of kernel time spent on level 1 cache misses.

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	9.3	9.0	8.9	14.3	19.6
raytrace	10.8	11.0	11.5	16.0	17.9
ocean	5.1	5.7	6.0	10.3	14.3

system model used in the performance experiments is configured as the original MAGIC design, containing a two-way set-associative instruction cache.

Changing to a direct-mapped cache could change the relative costs of local, remote, and communication cache misses, thereby changing the relative memory system costs of IRIX and the various Hive configurations. However, the hardware design team plans to lay out the frequently-executed protocol microcode routines to minimize conflict misses, so that the slowdown due to cache effects will be spread evenly over the microcode. If this effort succeeds, the effect of changing the cache design on the relative performance of the operating system configurations is likely to be minor.

5.4.3 Impact on reliability experiments

Using simulation makes it significantly easier to study the reliability of the system. SimOS is deterministic, enabling exact repetition of fault and error injection experiments so their effects can be analyzed in detail. However, use of SimOS also affects the results of the reliability experiments due to its limited address space.

SimOS only implements 32-bit virtual and 29-bit physical addresses. These are significantly smaller than the 64-bit virtual and 40-bit physical addresses of FLASH. Peter Chen has argued that the 64-bit virtual address space of a next-generation processor reduces the probability of wild writes [CNC+96]. If so, the inaccuracy caused by SimOS strengthens the primary experimental results in this dissertation, which argue that Hive successfully provides fault containment despite the possibility of wild writes.

Chapter 6
Fault containment

This chapter presents and evaluates Hive's fault containment mechanisms. The mechanisms fall into two broad categories: those that provide cell isolation and those that provide failure recovery.

Cell isolation in Hive includes both software and hardware features. The first two sections of the chapter describe the software features that preserve cell isolation despite the possibility of corrupt remote reads and wild writes. The third section of the chapter gives a precise definition of the memory fault model, which specifies the hardware features required for cell isolation despite the possibility of hardware errors.

Failure recovery in Hive consists of a set of software mechanisms that restore the operating system to a consistent state after one or more cells have failed. The fourth section of the chapter provides an overview of these mechanisms.

Finally, the last section of the chapter reports the results of fault injection experiments that stress Hive's implementation of fault containment.

6.1 Safe remote reads

One cell reads another's internal data structures rather than using RPCs in cases where RPCs are too slow, an up-to-date view of the data is required, or the data needs to be published to a large number of cells. Once the data has been read, it has to be sanity-checked just as an RPC received from the remote cell would be checked. However, the remote reads create additional fault containment problems.

Two mechanisms make remote reads safe: the *careful reference protocol* and the *publisher's lock*. The former ensures that a reading cell checks for misaligned pointers, data structures with loops, and so on. The latter allows the reading cell to synchronize with the cell that owns and may be updating the data.

6.1.1 Careful reference protocol

The careful reference protocol defends against two types of errors in the remote cell: hardware errors that are reflected through the memory fault model as bus errors, and software errors that

can appear as corrupt data. Bus errors are significant because cells normally panic (shut themselves down) if they detect such hardware exceptions during kernel execution. This is an essential self-check that is frequently the first indication of internal corruption, so it is important to weaken the self-check as little as possible when implementing careful references.

The reading cell follows these steps.

(1) Call the `careful_on` function, which captures the current stack frame and records which remote cell the kernel intends to access. If a bus error occurs due to reading the memory of that cell, the trap handler restores to the saved function context rather than causing a panic.

(2) Before using any remote address, check that it is aligned properly for the expected data structure and that it addresses the memory range belonging to the expected cell.

(3) Copy all data values to local memory before beginning sanity-checks, in order to defend against unexpected changes.

(4) Check each remote data structure by reading a structure type identifier. The type identifier is initialized by the memory allocator and overwritten by the memory deallocator. (This feature was simple to add to the allocator in IRIX, and seems likely to be similarly straightforward in most kernels.) Checking for the expected value of this tag provides a first line of defense against invalid remote pointers.

(5) Call `careful_off` when done so future bus errors experienced by this processor will correctly cause the kernel to panic.

Implementation details: `Careful_on` is implemented just like `setjmp`. There are two sets of `careful_on` register save areas, one for each process (stored in the user area) and one for each processor (stored in the processor private data area). Use of two register save areas allows a process that is in the middle of a careful reference to be interrupted safely, since the interrupt handler may itself need to do a careful reference. Interrupt handlers use the per-processor register save area and disable further interrupts while making careful references.

A process making a careful reference is allowed to specify a secondary target cell that it will access simultaneously with the primary cell. The case that forced adding this capability is obscure: if a parent forks a child process while it has locked copy-on-write pages, the child cell must preemptively copy those pages during the fork to avoid breaking the parent's lock. If any of those pages happen to be borrowed from a third cell, the child cell may need to remotely access two different cells simultaneously.

The usage paradigm for careful referencing is:

```
if (careful_on(cellid)) {
   /* Come here only if bus error occurred before careful_off.
    * Trap handler turns off careful referencing automatically.
    */
   hint_failure(cellid); /* inform recovery subsystem of error */
   return ERROR;
}
if (FCBADPTR(remote_pointer, remote_structure_type, cellid)) {
   /* bad pointer */
   careful_off(cellid);
   return ERROR;
}
bcopy(remote_pointer, local_buffer, sizeof(remote_structure_type));
careful_off(cellid);
return SUCCESS;
```

The FCBADPTR macro abstracts the necessary pointer and type identifier checks:

```
#define FCBADPTR(p,t,c) (   bad_pointer(p, alignof(t), sizeof(t), c)
                         || (p->fcmagic != typeidof(t)))

int bad_pointer(void* p, int align, int size, cellid_t c)
{
   nodeid_t t = NODE_NUMBER(p);  /* extract node bits from address */
   if (!IS_PHYSICAL_MEMORY_ADDRESS(p))          return 1;
   if (c != nodemap[t].cellid)                  return 1;
   if (p < nodemap[t].startaddress)             return 1;
   if (p+size > nodemap[t].endaddress)          return 1;
   if (p & (align-1))                           return 1;
   return 0;
}
```

The type tag is stored by convention in a field named `fcmagic`. The functions `alignof` and `sizeof` are compiler builtins. `Typeidof` is a macro that converts a type name to a unique integer at compile time. The type tag field is only read after checking that the pointer is valid, so if a bus error occurs the trap handler will return to the saved careful reference point rather than causing the kernel to panic.

6.1.2 Publisher's lock

A cell cannot acquire a lock on a remote data structure it needs to read. If a cell allowed remote cells to modify the lock variable of one of its data structures, the cell could no longer trust that its own accesses to the data structure would be free of race conditions or deadlocks. The *publisher's lock* mechanism allows a remote cell to acquire a consistent view of a data structure without acquiring a lock.

The publisher's lock is an integer variable incremented by the cell that owns the data structure before modifying the structure and again after modification is complete. The remote cell copies the data structure and uses the publisher's lock to determine whether its snapshot is consistent.

```
typedef struct {
   unsigned short v;
} publ_lock;

void publ_init(publ_lock* p)
{
   p->v = 0;
}

void publ_lock(publ_lock* p)
{
   p->v++;
   /* hardware store barrier here if the multiprocessor
      is not sequentially consistent */
}

void publ_unlock(publ_lock* p)
{
   /* hardware store barrier here */
   p->v++;
}

/* precondition: careful referencing enabled,
   p and remote_struct pointers already checked with FCBADPTR */
int read_publ_structure (volatile publ_lock* p,
                         struct st* remote_struct,
                         struct st* local_copy)
{
   int timeout = TIMEOUT;
   while (timeout-- >= 0) {
      unsigned short a = p->v;
      if (a & 0x1) continue;
      *local_copy = *remote_struct;
      if (a == p->v) {
         return SUCCESS;
      }
   }
   return ERROR;
}
```

The only subtlety of this algorithm is the placement of the hardware store barriers, which are needed on machines with relaxed memory consistency models [GLL+90]. Hive does not use the store barriers. The base IRIX implementation is not well-labeled so FLASH must run in sequential consistency mode to execute Hive correctly.

When implementing Hive, an unexpected usage pattern for the publisher's lock appeared. Hive does not have publisher's locks on any kernel data structures inherited from IRIX, only on the new

ones added to support information dissemination between cells. The reason is that in cases where one cell reads another's internal data structures, such as during remote process creation, the operation is invariably requested by the cell that owns the data structures and thus it can lock the data structures before requesting the operation.

6.1.3 Evaluation

These mechanisms make it efficient to read remote data directly. An example use of the careful reference protocol is the clock monitoring algorithm, in which the clock handler of each cell checks another cell's clock value on every tick. With warmed-up caches, the average latency from the initial call to `careful_on` until the terminating `careful_off` call finishes is 1.46 μsec (292 cycles), of which 1.0 μsec (200 cycles) is the average latency of the cache miss to the memory line containing the clock value. This is substantially faster than sending an RPC to get the data, which takes a minimum of 7.2 μsec and requires interrupting a processor on the remote cell.

The larger question is how useful it is to read remote data directly. Chapter 7 examines this question in detail.

6.2 Wild write defense

The second way that one cell can corrupt another is by writing to its memory. Hive defends against wild writes using a three-part strategy. First, it manages the FLASH firewall to minimize the number of pages writable by remote cells. Second, when a cell failure is detected, other cells preemptively discard any pages writable by the failed cell. Finally, Hive implements mechanisms to detect failures quickly after they occur. The next sections describe each part of the strategy in turn.

6.2.1 Firewall management

I describe the implementation of the firewall in FLASH in more detail before presenting and evaluating the firewall management policy of Hive.

FLASH firewall: The firewall controls which processors are allowed to modify each region of main memory. FLASH provides a separate firewall for each 4 kilobytes of memory, specified as a 64-bit vector where each bit grants write permission to a processor (the granularity and number of bits can be changed in protocol microcode). On systems larger than 64 processors, each bit grants write permission to multiple processors. A write request to a page for which the corresponding bit is not set fails with a bus error.

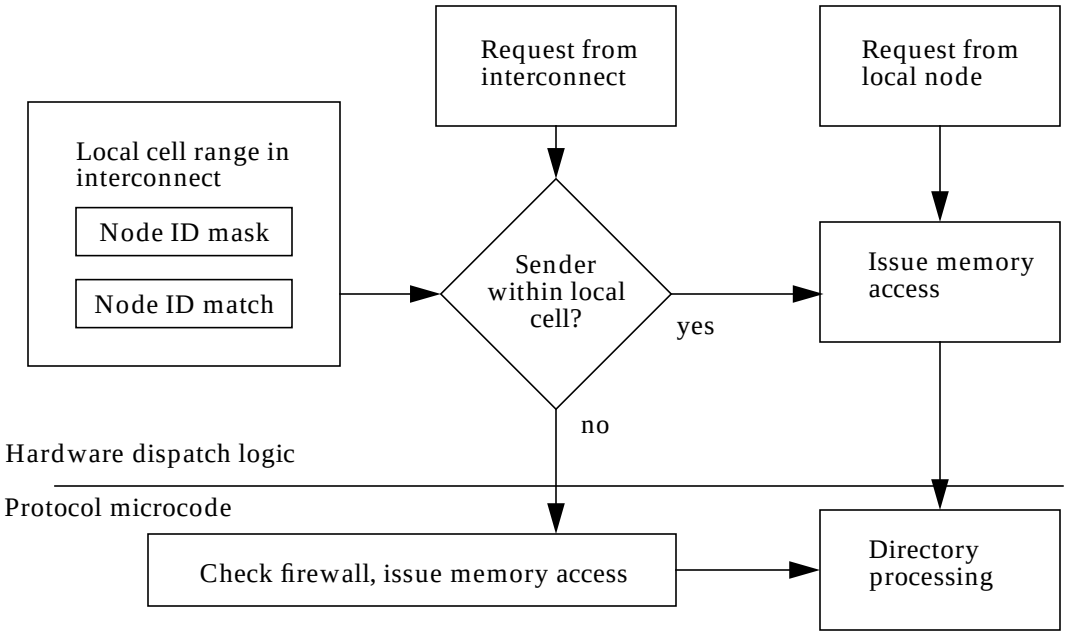


Figure 6.1. Firewall support in MAGIC.

The protocol microcode running on each node stores and checks the firewall bits for the memory of that node (Figure 6.1). It checks the firewall on each request for cache line ownership (read misses do not count as ownership requests). Uncached accesses to I/O devices on other cells always receive bus errors, while DMA writes from I/O devices are checked as if they were writes from the processor on that node. To reduce the performance impact of the firewall check, dedicated logic determines whether the incoming request is from a node of the local cell before dispatching the protocol code.

A cell can change the firewall for a page with a single uncached write. The rule for firewall modification is interesting: any cell that has write permission to a page may modify the firewall for that page. FLASH initially implemented the more intuitive rule that only the cell that owns a page may modify the firewall for that page. Section 7.1.5 on physical-level memory sharing explains why the rule was changed and discusses the implications of the current rule.

The 4 kilobyte firewall granularity matches the operating system page size. Anything larger would constrain operating system memory allocation, whereas it is unclear whether a finer granularity would be useful.

A bit vector per page was chosen over two other options that would require less storage. A single bit per page, granting global write access, would make processes that use any remote memory vulnerable to errors anywhere in the machine. A byte or halfword per page, naming a processor

with write access, would make it inefficient to allocate a remote page to a parallel process since multiple threads could not write-share the page.

The performance cost of the firewall is minimal. The firewall check increases the average nearest-neighbor remote write cache miss latency under pmake by 6.3% and under ocean by 4.4%, comparing runs of these workloads with the firewall enabled and with it disabled. This increase has little overall effect since write cache misses are a small fraction of the workload run time. In fact, the overall run time of pmake was longer with the firewall disabled than with it enabled, due to a change in the interleaving of process execution.

Firewall management policy: Firewall management is a tradeoff between fault containment and performance. For the best possible wild write defense, write permission to a user data page should only be granted while a write-enabled mapping to that page is active in the TLB of a processor of another cell.

However, it is relatively expensive to change the firewall write permission for a page. It requires sending an RPC (minimum 7.2 μsec) to the cell that manages that page. There is also significant hardware cost when retracting firewall write permission for a page. All lines cached exclusively by the processors of the cell that is losing permission must be retrieved, since a line that is cached remains modifiable irrespective of the firewall state. In practice it is difficult to determine which lines are cached exclusively, so the cell that manages the page must read through the entire page after retracting write permission. This could add 10 μsec or more if many lines are cached exclusively. The total cost is far too expensive to pay on each change in TLB mappings, which can occur as often as once every 1000 user instructions in some workloads [RHW+95].

Hive uses a more relaxed policy. Write access to a page is granted to all processors of a cell when any process on that cell faults the page into a writable portion of its address space. Granting access to all processors of the cell allows that cell to freely reschedule the process on any of its processors without sending RPCs. Write permission remains granted as long as any process on that cell has the page mapped in its address space. This reduces the number of firewall write permission changes to an acceptable level.

Evaluation: The effectiveness of this policy can be measured by comparing pmake, which shares few writable pages between the separate compile processes, with ocean, which shares its data segment among all its processes. During 5.0 seconds of execution sampled at 20 msec intervals, pmake had an average of 15 remotely writable pages per cell at each sample (out of about 6000 user pages per cell), while ocean showed an average of 550 remotely writable pages.

The low number of writable pages under pmake shows that this policy should provide good wild write protection to a system used predominantly by sequential applications. The highest recorded number of writable pages during the workload was 42, on the cell acting as the file server for the directory where compiler intermediate files are stored (`/tmp`). This is low enough to make it extremely unlikely that a wild write will bypass the firewall and corrupt memory.

In the case of ocean, the current policy provides little protection since the global data segment is write-shared by all processes. However, the application is running on all processors and will exit in any case when a cell fails, so any efforts to prevent its pages from being discarded will be wasted. The firewall management policy avoids protection status changes that would create unnecessary performance overheads for this type of application.

6.2.2 Preemptive discard

When a cell fails and issues wild writes, it will be able to corrupt the data stored in pages to which it has firewall write permission, thereby corrupting applications that subsequently read the data. The preemptive discard policy attempts to prevent this problem.

Determining potentially corrupt pages: It is difficult to determine efficiently which pages to discard after a cell failure. Many cells could be using a given page and therefore need to cooperate in discarding it, but only the cell that manages the page knows its precise firewall status. Distributing firewall status information during recovery to all cells using the page would require significant communication.

Hive avoids this cost. Each cell determines locally which of its pages were writable by the failed cell or cells and marks those pages as discarded. Additionally, all TLBs are flushed and all remote mappings are removed during recovery. This ensures that a future access to a remote page will fault and send an RPC to the cell that manages the page, where it can be checked.

Recording discarded pages: Accesses to discarded pages might occur arbitrarily far in the future, making it quite expensive to record exactly which pages of each file have been discarded. Hive solves this problem by relaxing the application-visible error semantics slightly.

In most current UNIX implementations the file system does not attempt to record which dirty pages were lost in a system crash. It simply fetches stale data from disk after a reboot. This is acceptable because no local processes can survive the crash, so a process that accessed the dirty data will never observe that it was unstable.

Hive takes advantage of these semantics by allowing any process that opens a damaged file after a cell failure to read whatever data is available on disk. Only processes that opened the file before

the failure receive an error upon accessing discarded pages. This is implemented with a generation number, maintained by the file system, that is copied into the file descriptor or address space map of a process when it opens the file. When a cell discards a dirty page of a file, it instructs the file system to increment the file's generation number. A subsequent access via a file descriptor or address space region with a mismatched generation number generates an error.

Evaluation: The preemptive discard policy has two costs. The first is the number of pages that are discarded that have not been damaged by wild writes. The fault injection experiments reported later show that the chance of wild writes is very low. However, it is safer to discard a valid page, perhaps forcing the user to run an application again, than to preserve a corrupted page, which could cause an application to generate incorrect output. The number of pages discarded is governed by the firewall management policy described earlier.

The second cost is an increase in the execution time of undamaged applications after a failure. This is an artifact of the implementation of preemptive discard rather than the policy itself. Removing all remote memory mappings during recovery forces surviving applications to rebuild all their mappings, which adds a substantial number of page faults. This cost is acceptable because errors that require full recovery (as opposed to null recovery) are assumed to be rare.

6.2.3 Failure detection

Preemptive discard can only prevent the use of corrupted data if it runs soon enough after the data is corrupted. Therefore part of the wild write defense is detecting cell failures as quickly as possible, using heuristic checks.

The heuristic checks in the prototype are not sophisticated. In particular, the timeouts are chosen arbitrarily. I list them here to allow evaluation of the reliability experiments in the dissertation. The timeout values chosen are large enough to avoid false alarms during the performance experiments, and therefore are somewhat larger than ideal for reliability.

A cell is considered potentially failed if one of the following timeouts occurs:

- The cell fails to respond to an RPC request. This timeout is set at 500 msec for RPCs serviced at interrupt level and one second for RPCs that must schedule a server process.
- The cell fails to respond to a ping request sent by the heartbeat algorithm that is part of recovery. This timeout is set at 160 msec.
- A shared memory location that the cell updates on every clock interrupt fails to increment. Monitoring the clock detects hardware errors that halt processors but not entire nodes, as well

as operating system errors that lead to deadlocks or the inability to respond to interrupts. This timeout is set at 30 msec.

Additionally, the prototype uses other mechanisms that are not based on timeouts. A cell is considered potentially failed if one of the following conditions occurs:

- An attempt to access the memory of the cell causes a bus error. This will occur if there is a serious hardware error or if the cell incorrectly denies firewall access to a page that another cell is writing.
- Data read from the cell's memory or received in a message fails sanity checks. This detects software errors.
- A memory location (the `alive` field of the cell public area, Section 7.2.1) changes away from the correct value, indicating that the cell has detected an internal error and halted. Each cell's `alive` field is read by some other cell on each clock interrupt.

Detecting one of these timeouts or conditions provides a hint that some cell may have failed. It is only a hint since the error may have been in the cell where the check algorithm executed. For instance, in one case a software fault caused the clock interrupt handler in a cell to execute too frequently. Pending operations on that cell timed out prematurely, leading the damaged cell to wrongly suspect that another cell had failed.

When a hint is raised, the distributed agreement algorithm runs and all cells come to a consensus about which cells are live and which have failed. To prevent a corrupt cell from repeatedly broadcasting alerts and damaging system performance, a cell that broadcasts the same alert twice but is voted down by the distributed agreement algorithm both times will panic.

Evaluation: This set of failure detection mechanisms has minimal performance overhead because most of the checks are those that must be performed in any case to defend against corruption due to receiving bad data. However, the checks appear to detect failures quickly enough to prevent damage due to wild writes, as shown in the fault and error injection experiments reported later.

The major unanswered question about failure detection is whether the number of false alarms will scale with the number of cells or the number of processors per cell, forcing looser timeouts and less frequent heuristic checks to maintain acceptable performance overheads. If so, the reliability of the system will decrease with size. Of the mechanisms implemented in the prototype, only the rate of RPC timeouts appears likely to increase with system size, since the increased interaction rate between cells creates both a greater possibility of congestion and a greater chance of detecting any temporary congestion that occurs. The small system sizes and large timeouts used in the

experiments do not provide the data needed to judge whether this will be a problem, either with RPC timeouts or with the other heuristic checks that may be added later.

6.2.4 Summary and possible improvements

The wild write defense consists of the firewall, policies that manage write permission, the preemptive discard policy, and fast failure detection mechanisms. For the fast failure detection mechanisms to be efficient, the null recovery case must be fast. Its implementation is described in Section 6.4.

All four architectural components of the wild write defense could be implemented differently or better in future multicellular kernels.

- *Firewall:* The firewall is implemented in hardware by FLASH, but could be provided in software by a microkernel or low-level supervisor on multiprocessors that do not provide it in hardware. This design alternative is discussed in Section 9.1.

  Whether implemented in hardware or in software, the firewall could be extended with an additional vector that records which processors have modified the data in the page (to be more precise, have ever cached any line in the page in exclusive mode). This feature would double the storage requirements and add significantly to the memory bandwidth consumed by the firewall, which currently is mostly read-only and can be effectively cached in MAGIC. However, adding a write detection vector would enable the kernel to significantly reduce the number of pages discarded after a failure, which may be a strong enough benefit to justify the hardware cost.

- *Firewall management:* The simple firewall management policy is successful at minimizing writable pages in sequential workloads like pmake because the prototype lacks processor sharing mechanisms. When implemented, these mechanisms will tend to increase the number of shared pages by creating components of spanning tasks on multiple remote cells. It seems likely that the firewall management policy will need to be coupled to the migration mechanism. In particular, firewall permission should not be granted to a remote cell when the only processes on that cell with the page mapped are inactive spanning task components, since the migration mechanism will leave these components in place for long periods of time to support future load-balancing demands.

- *Preemptive discard:* The preemptive discard policy may not be appropriate for all applications. For example, a database process pair using a shared-memory communication region is likely to have its own application-level sanity checks on the data read from that region. The system

should permit sophisticated applications to declare that certain files or mappings are checked by higher-level correctness protocols so preemptive discard is not necessary.

Moreover, certain types of cell failures are much less likely than others to cause wild writes. In particular, cell failures due to events such as power failures and interconnect link failures could be assumed to be fail-fast. The system should differentiate these types of failures from software errors and preserve as much data as possible, including perhaps recovering data pages from the memory of the failed cells.

- *Failure detection:* Possible advanced failure detection mechanisms include running processes that walk the internal data structures of other cells looking for inconsistencies, periodically sending RPCs to other cells that exercise system functionality such as file and process management operations, and adding heartbeat outputs visible from outside the cell to critical system processes such as the virtual memory clock hand and the disk buffer flush daemon. Which of these advanced mechanisms are necessary for a particular multicellular kernel will become apparent as it is tested under long-running stressful workloads on a large-scale machine.

6.3 Memory fault model

The previous two sections described how cell isolation is implemented with respect to reading and writing data through shared memory. However, these are not the only channels by which corruption can propagate. FLASH is a tightly-coupled machine and hardware errors in one cell can easily perturb the operation of other cells. Hive requires hardware support from the lower levels of the fault containment stack if it is to recover from hardware errors. This section describes the memory fault model that specifies the support that Hive needs.

6.3.1 Specification

FLASH is partitioned into *hardware failure units* (HFUs). A *correct HFU* is one in which there have been no hardware errors since it was last reset. Each HFU is the *home HFU* for a range of *memory lines* (cache-line-sized and cache-line-aligned units of main memory). The memory ranges of different HFUs do not overlap.

Memory lines that function correctly are both *coherent* and *accessible*. A coherent line is one for which reads by any processor in a correct HFU return the data specified by the memory consistency model of the machine. An accessible line is one for which cache misses by any processor in a correct HFU complete in less than some fixed time *t*.

Given these definitions, the memory fault model has six properties:

(1) A hardware error in an HFU can make any memory lines for which it is the home HFU incoherent or inaccessible.

(2) A hardware error in an HFU will not make memory lines with a different home HFU inaccessible to any processors in correct HFUs.

(3) A hardware error in an HFU can only make a memory line with a different home HFU incoherent if some processor in the faulty HFU had firewall write permission to that line before the error.

(4) An access to an incoherent line by a processor in a correct HFU returns an error. An access to an inaccessible line by a processor in a correct HFU returns an error in less than some fixed time *u*.

(5) A reset operation is available that changes a memory line from incoherent to coherent. The contents of the line are undefined after reset.

(6) A hardware error may cause any uncached access in the machine to fail to complete except those issued and satisfied within the same correct HFU.

Note that these properties are assumptions made by Hive, not absolute guarantees made by the hardware implementation. A hardware error may occur that violates the model, but in this case Hive does not guarantee to provide fault containment and an uncontained failure is likely to occur.

6.3.2 Design discussion

The memory fault model is a compromise between what would be ideal for a multicellular kernel and what can be implemented cheaply and efficiently in a CC-NUMA multiprocessor. Several aspects of this tradeoff are interesting:

- *Remote reads:* A processor is allowed to read memory lines that it should not make incoherent or inaccessible (fault model points 2 and 3). Therefore a processor that fails with a line shared in its cache must not prevent further writes to that line by other processors. This requires the memory system to implement fairly complex recovery mechanisms such as timeouts on invalidation requests and the sledgehammer algorithm. This complexity is justified by the importance of memory sharing for achieving competitive performance.

  The converse is that a processor should not be damaged by reading remote lines. This interacts with the choice of data-link level error detection mechanisms. In a highly pipelined

architecture like FLASH, the first word of a cache line that is arriving over the network can be delivered to the processor before the last word has been received by the network interface. Therefore it is not sufficient to put a checksum at the end of the network packet. Error detection and correction must be provided for each word of a cache line during transmission, which reduces the fraction of the network bandwidth available for transmitting data.

- *Retracting the firewall:* Point (3) is stated in a way that a line remains vulnerable to faults in a remote HFU after firewall permission has been retracted. This is unavoidable because the sole copy of the line may still be remotely cached or in flight through the network when the firewall changes. Full application of this aspect of the model would suggest that a cell could never reallocate a page from shared user data to kernel data, since kernel data must not be lost in a remote failure.

  To avoid this limitation, FLASH provides a *retract* operation to ensure that a line is no longer vulnerable to a remote cell that previously had write permission. A cell can retract a line by reading any word in it. This ensures that a clean copy is written to memory.

- *Uncached accesses:* Point (6) is a concession to implementation constraints. It turns out to be quite difficult to provide guarantees to launch-and-forget uncached writes and non-idempotent uncached reads. Hive ensures correctness by using a read-after-write verification protocol when issuing uncached operations across HFU boundaries, for example when remotely changing the firewall state of a line.

- *Network partitions:* The fault model implies that hardware faults do not lead to network partitions. Although software techniques for managing network partitions are known [PoW85], enough separate links cross each slice of the FLASH interconnect that the implementation complexity of these software techniques is not justified. A network partition will cause Hive to fail.

6.3.3 Limitations of the FLASH implementation

FLASH does not implement the full memory fault model. There are two major deviations, potential loss of incorrect packets and potential corruption of memory through incorrect writebacks.

Network packet loss: FLASH has a non-fully-connected network. Therefore communication between HFUs A and C can pass through HFU B (Figure 6.2). If B suffers a catastrophic failure such as loss of power, packets in flight between A and C can be lost. Since FLASH frequently sends the sole copy of a line into the network, for example when writing back a dirty cache line,

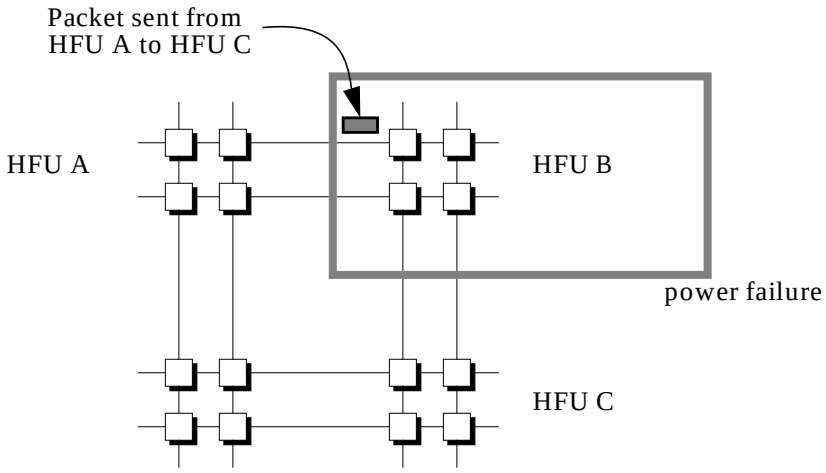


Figure 6.2. Network packet loss that violates the memory fault model.

this means that a hardware failure can destroy data even though no processor in the HFU in which the failure occurs has firewall write permission.

The memory fault model does not include the potential for data loss due to network packet loss because such errors make an application that uses memory from more than one cell vulnerable to hardware failures anywhere in the machine. If Hive considered this vulnerability in making its memory allocation decisions, it would face significant restrictions when attempting to balance memory demand across cell boundaries, reducing the performance of the system.

However, unlike other hardware errors in which violation of the memory fault model can lead to uncontained failures, Hive is designed to cope with random user data lines suddenly becoming incoherent due to network packet loss. This is standard recovery code, exactly that used for ECC errors in current SMP operating systems, and so adds little complexity to the system. The network packet loss problem does not affect kernel data internal to a cell because cells are convex and internal kernel data is never writable outside its home cell.

Incorrect writebacks: Implementation constraints in the FLASH protocol prevent using the firewall to check some writebacks of dirty data. The problem arises with sharing writebacks, which occur when a processor with an exclusive copy of a cache line transfers the line to another processor that has requested a shared copy. The processor with the exclusive copy sends the line out twice, once to the requesting processor and once to the home node to update it. The protocol header is fixed and has a single source processor id field, so in order for the protocol microcode at the home node to know which processor to add to the sharing list, the sending processor puts the

requesting processor's id into the source field rather than its own processor id. Therefore the processor sending the writeback cannot be checked against the firewall.

This is not the only problem with checking writebacks. A race condition can occur even with normal (non-sharing) writebacks. If the operating system were to deny writes from a processor that happened to have the line cached exclusively, the writeback would be denied and the line would never return to the clean state in memory. Because speculative execution makes it impossible for the operating system to guarantee that a line is not cached exclusively, a machine that checks writebacks would have to support an intermediate firewall state. In this state write requests would be denied but writebacks would be permitted, with reversion to a full protection state when the retract operation has been issued for the line.

Because of these complications, FLASH sinks all writebacks to memory without checking the firewall. Although it is possible for the operating system to force an incorrect writeback by directly corrupting the cache tags, in general an incorrect writeback represents a memory system error and thus is outside the error model.

6.3.4 Evaluation

Both the precision and the abstraction of the memory fault model were helpful in the design process. The precision enabled the operating system and hardware development teams to discover and resolve different assumptions early in the design. For example, the teams had made opposite assumptions about the legality of remote reads when firewall permission is not granted.

The high abstraction level of the specification has two benefits. First, it sets a goal that allows hardware implementations with varying degrees of completeness. The network packet loss problem described above is one example of this.

Second, it covers hardware contingencies not considered when the model was developed. For example, there is an ongoing effort at Stanford to implement a *cache-only memory architecture* (COMA) for FLASH by modifying the protocol microcode. COMA systems evict memory lines from a node when there are conflict misses in the associative memory cache. The developers of the COMA microcode can use the memory fault model to determine which other nodes are candidates to receive the evicted line.

Most of the features in the model can be implemented fairly completely. The sledgehammer algorithm at the memory system level, when combined with network-level recovery and an appropriately-structured physical level, covers most of the cases.

Only one aspect of the model turned out to be impossible to implement in FLASH: the time limit *t* on cache misses to accessible lines. Memory system hotspots can easily occur that cause significant queuing delays, while overall network congestion can reach truly staggering levels under pathological conditions. Therefore the timeout set on cache misses by the memory system to detect hardware errors must be very loose. Cache misses, even those internal to a cell, can be stalled by an error and will not complete until after the timeout has fired and hardware recovery has run. The result is that *t* is the same as the time *u* to detect that a line is inaccessible, which is multiple milliseconds, suggesting that large multiprocessors may not work well in environments with real-time constraints unless more responsive memory systems can be developed.

The timing assumption is in the model because Hive delegates the solution to the hotspot and congestion problems to the memory system developers. It does not include the features (such as idling processors if the memory system is overloaded) that would be required to manage these latency problems at the operating system level.

6.4 Failure recovery

The cell isolation mechanisms described in the previous three sections—safe remote reads, the wild write defense, and the memory fault model—provide the abstraction of fail-fast cells but generate occasional false alarms. The failure recovery mechanisms are responsible for distinguishing false alarms from actual failure conditions and restoring the system to a consistent state when cells have failed.

The recovery subsystem contains four kernel-level processes (Table 6.1). Each process provides an abstraction layer to the next process that simplifies the higher-level task.

Recovery layers: The lowest-level process, called *ping*, is the only real-time process in recovery. When a hint is raised by the cell isolation mechanisms, or a hint alert is received from another cell, the ping process activates and begins exchanging heartbeats with all other cells in the system. It provides continual awareness of which cells are alive to the liveset process.

The *liveset* process runs a distributed algorithm to ensure that all correct cells in the system agree on which cells are alive and correct. Each cell takes its initial vote from the output of the first ping round. These votes are then intersected to produce agreement. The distributed algorithm used is a fault-tolerant flood algorithm, in which each cell broadcasts its current understanding of the intersection as many times as there are cells in the system [Lyn96]. When the liveset process completes agreement, each cell independently compares the new liveset to the previous one to compute the *die set*, the set of cells that have failed. The liveset process aborts all RPCs waiting for responses from failed cells and passes the die set to the recovery process.

Table 6.1. Recovery subsystem design.

Layer name	Responsibilities	Operating system features used
Ping	Heartbeat algorithm: track which cells are up	Message send, real-time response
Liveset	Flood algorithm: reach distributed consensus	Message send
Recovery	Preemptive discard and all other system cleanup	RPC
Reboot	Diagnostics, reboot failed cells, reintegrate rebooted cells	File system

The *recovery* process is responsible for returning the system to a consistent state that excludes the cells in the die set. It calls the cleanup routines of the virtual memory system, file system, process subsystem, memory manager, and others. This requires communication between cells that is implemented using RPCs. RPCs are safe to use because the liveset process will run again and unwedge the recovery process if another failure occurs.

The *reboot* process executes hardware diagnostics on the cells in the die set, resets them, loads new copies of the kernel image from disk, and integrates rebooted cells back into the system. The reboot process only activates on one cell, the master cell, which is elected by being the lowest numbered cell in the live set. If the master cell fails before the cells it is rebooting have been reintegrated, recovery runs again and the reboot process on the new master cell restarts all the rebooting cells from the beginning.

When the reboot process detects that a cell it is rebooting is ready to join the system, it raises a hint that triggers recovery. When recovery runs the liveset processes will reach agreement on a new live set that is larger than the previous live set, and the resources of the rebooted cell immediately become visible to the rest of the system. This is called the *reintegration recovery round*. To support use of recovery for reintegration and initial system boot, the first ping round of each recovery round sends pings to all cells in the system, not just those in the previous live set.

Recovery process: The internal structure of the recovery process supports important performance optimizations in the virtual memory system. Figure 6.3 shows the two global barriers that synchronize the recovery processes of different cells. When all live cells have reached the first global barrier, each cell knows that no more virtual memory or file system requests are pending from cells in the live set. Each cell proceeds to discard all state related to pages mapped across cell boundaries (part of the preemptive discard algorithm described in Section 6.2.2). When all cells have reached the second global barrier, each cell knows that the others have cleaned up their

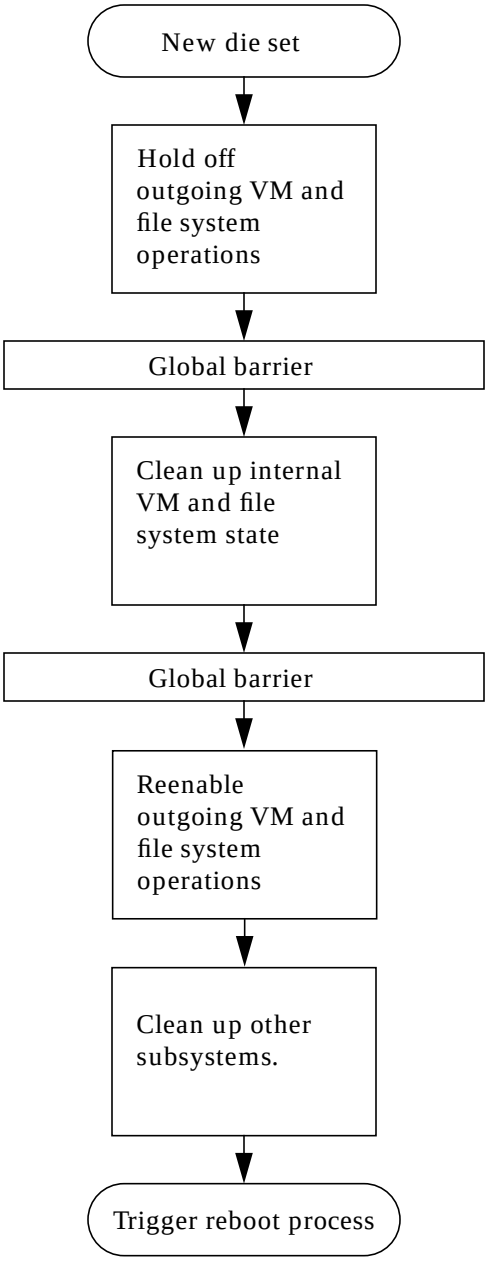


Figure 6.3. Control flow of recovery process.

internal state, so it is safe to begin sending virtual memory requests as part of other cleanup actions.

This design moves the synchronization between virtual memory operations and recovery out of the RPC handlers that service virtual memory requests, and thereby enables these requests to be serviced at interrupt level rather than paying the cost of scheduling a server process.

Performance: The primary performance metric for the recovery subsystem is the latency of null recovery. Null recovery for an eight-cell system takes 11.0 msec. This is an artifact of the current recovery subsystem implementation, which runs the recovery process through to completion (including checking for preemptive discard) even if the die set is empty.

Preemptive discard accounts for all except 3 msec of the current null recovery time. Preemptive discard is expensive because it involves scanning all firewall entries to find pages writable by failed cells. At present this costs 250 μsec per megabyte of storage; it would be worth parallelizing this algorithm across the multiple processors of each cell to reduce pause times when cells have large amounts of memory.

With an improved implementation that skips the preemptive discard check in a null recovery, the dominant cost will be the live set algorithm. It takes 1.6 msec on two cells, 2.0 msec on eight cells, and, by extrapolation of the $O(n^2)$ algorithm, around 80 msec for 64 cells, the maximum supported with the current FLASH firewall. As even systems with 128 processors are likely to run with eight cells or so, this time is fast enough that most Hive systems could raise several false alarms per second with no significant performance impact. The number of false alarms raised can be tuned by changing the tightness of timeouts and the frequency of active probes of other cell's memory, so the system administrator can adjust the performance overhead to match the requirements and behavior of the workload running at each site.

6.5 Fault and error injection experiments

Three types of experiments are used to evaluate the implementation of fault containment. Heap corruption models the result of programming errors that leave incorrect values in kernel data structures. Node failures model the loss of a portion of the system due to a catastrophic hardware fault such as a power failure. Finally, instruction corruption directly models programming errors. As described in Section 5.3, the experiments use a parallel make workload running on a four-processor system booted with four cells.

6.5.1 Heap corruption

To make the heap corruption experiments as stressful as possible, they are targeted to data structures that are used frequently by the kernel and read by remote cells through shared memory (Table 6.2). In each experiment one word of memory is changed at a random time after the parallel make has begun.

In each experiment a pointer in the listed data structure is changed in one of three ways:

Table 6.2. Heap corruption experiments.

Corrupt pointer in:	Number of tests using each type of corruption		
	Random address	Old value +4 or -8	Address of pointer itself
process address map	50	50	50
process table entry	50	50	50
page hash table	50	50	50
run queue	50	50	50
kernel malloc free list	50	50	–
per-process user structure	100	–	–

- *Random address:* Overwriting the pointer with a valid but random physical memory address forces the case most likely to generate wild writes.
- *Old value +4 or –8:* Overwriting the pointer with a small integer offset from the old value damages kernel malloc headers and causes reads through the pointer to return random data.
- *Address of pointer itself:* Overwriting the pointer with the address of the pointer itself creates the pathological looping case that stresses the careful reference protocol used for remotely reading data structures.

In all cases listed in the table Hive successfully confined the effects of the fault to the cell where it was injected. Hive did not initially succeed in all experiments performed, but when uncontained failures occurred, the bug was identified and fixed and the experiment repeated until successful. In no cases, either those listed or those where uncontained failures occurred, was any of the data output to disk by the compiles incorrect.

6.5.2 Node failures

Catastrophic hardware errors such as node failures create a lesser software fault containment challenge than heap or instruction corruption, because if the hardware implements the memory fault model correctly, the faulty cell simply halts and its memory becomes inaccessible. In 150 experiments, a node was halted at a random time during the make, while in another 50 experiments, a node was halted while a process on one cell was creating a child process on another cell. The remote process creation algorithm uses RPCs and remote memory reads particularly

intensively, so this is a stressful time for a node to fail. Hive successfully contained the effects of the node failure to the affected cell in all 200 experiments.

6.5.3 Instruction corruption

To provide a more interesting statistical sample than is available from the heap corruption and node failure experiments, the instruction corruption experiments are organized differently. After performing an initial 250 instruction corruption experiments and fixing the system bugs that led to uncontained failures, Hive's reliability was measured by doing 1000 corruption experiments with no system modifications.

Experimental design: The instructions to be corrupted were selected as follows. SimOS was first configured to measure the number of times each instruction address in the cell 1 kernel was executed during a 40 million cycle window of the parallel make workload, starting just after the first round of compiles had begun execution on all four cells. Then these addresses were sorted by frequency and the 5% most common addresses were discarded. An error in one of these commonly-executed locations is unlikely in practice because these instructions are well-tested. Discarding this set also eliminates routines in which one would not expect any errors to occur, such as the idle loop and `page_copy`. The instructions to be corrupted were then chosen randomly from among the remaining 95% of the executed addresses. Since SimOS is deterministic and all 1000 experiments start from the same checkpoint, the corrupted instruction is guaranteed to be executed at least once during the parallel make run.

Instruction corruption uses the methodology developed in [CNC+96]. Figure 6.4 shows the decision tree used to determine how the instruction is corrupted. If the instruction is a branch and the decision tree randomly decides to change the destination register, the sense of the branch is inverted, simulating an off-by-one bug or inverted conditional. The instruction is changed 10 million cycles into the run.

Overall results: The results fall into six categories (Table 6.3). In 59% of the experiments Hive successfully limited the effects of the fault to the failure of cell 1. The fault had no visible effect 31% of the time. About 6% of the faults caused processes in the workload to exit but did not lead to cell failure. This can occur, for example, if a cell incorrectly returns an error to an I/O request. Simulator limitations prevented completion of 1% of the experiments. An example of this is that SimOS does not implement MIPS 64-bit mode or supervisor mode, so if the malfunctioning cell writes a value to the status register that selects one of these modes, the experiment must be terminated.

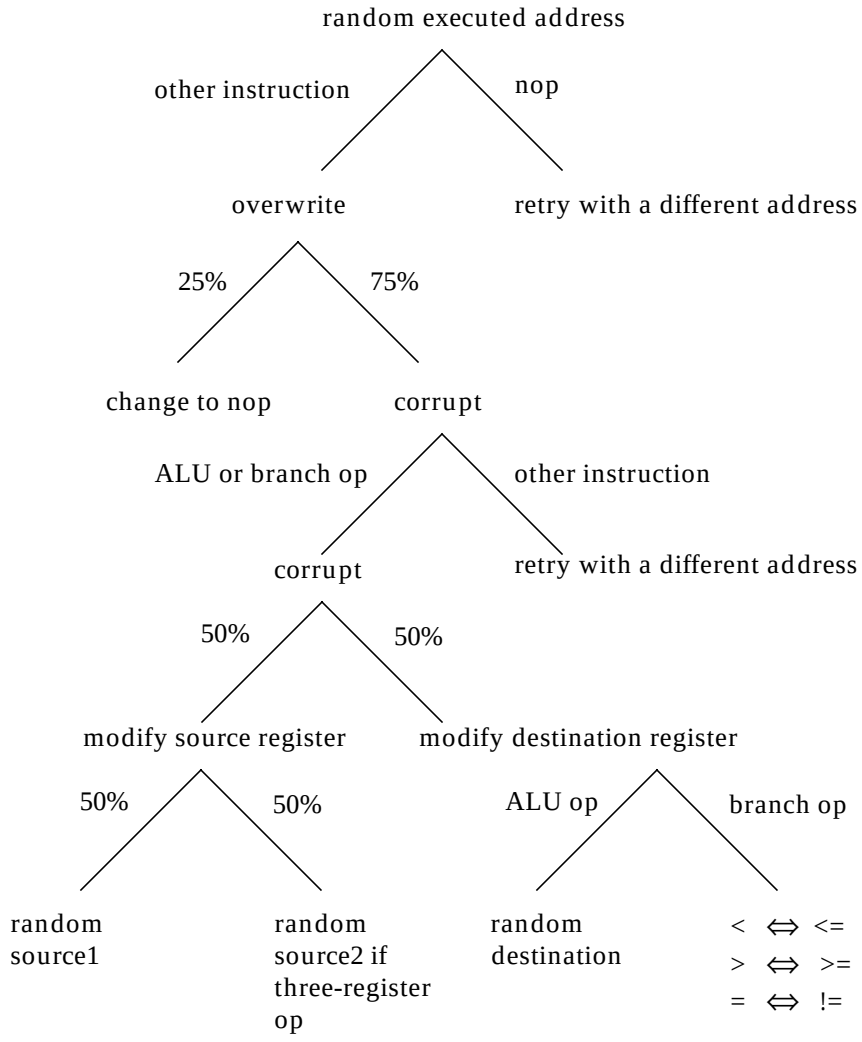


Figure 6.4. Decision tree for instruction corruption experiments.

Finally, 3% of the injected faults lead to uncontained failures, defined as failure of processes in other cells, failure of other cells, or failure of the entire system. This is a 4% uncontained failure rate when only the cases where an SMP operating system would have failed are considered (the first and last rows of the table).

Causes of uncontained failures: The uncontained failures come from a variety of causes (Table 6.4). The first three causes listed in the table are software faults in the implementation of Hive that are easily fixable, but perhaps are representative of the types of faults that can lead to uncontained failures in multicellular kernels.

Table 6.3. Instruction corruption experiments

Result type	Number of occurrences out of 1000	Percent of cases where SMP kernel would have failed
Cell 1 failure, Hive recovers successfully	586	96%
No visible effect on execution	311	
Processes terminated but cell 1 does not crash	64	
SimOS limitations prevent completion of experiment	12	
Uncontained failure	26	4%

Table 6.4. Causes of uncontained failures.

Category	Cause of failure	Number of occurrences
Software faults in Hive	Premature reintegration round	8
	File system neglects to acquire recovery lock	7
	File system sanity checks insufficient	4
Conditions not handled by recovery subsystem	Injected fault prevents RPC replies, other cells panic	3
	Injected fault prevents message sends, faulty cell resets rest of system	2
	Injected fault causes repeated recovery, cell bypasses self-check for this case	1
Byzantine faults	Faulty cell issues too many requests, causes value overflow in server cell	1

- Due to a race condition between the live set process and the reboot process, the reboot process can trigger the reintegration recovery round before the failed cell has finished rebooting. The reintegration recovery round concludes that the cell has failed and resets it again. This condition leads to repetitive recovery executions.
- The client side of the file system occasionally neglects to acquire the lock used by the recovery process to hold off file system activity. A file system request from a live cell that arrives between the two global barriers of the recovery process can cause the file server cell to fail.

- The server side of the file system does not check the validity of request arguments sufficiently to avoid failing under certain types of corruption.

The next three types of uncontained failures result from injected faults that create conditions not handled by the recovery subsystem.

- If the injected fault renders the cell unresponsive to RPC requests but the ping and liveset processes still function, other cells commit suicide one by one. This occurs because any cell that tries to interact with the faulty cell receives an RPC timeout, starts recovery, and learns that the faulty cell is in the liveset. After this occurs twice in a row the cell panics. It would be possible to defend against this type of fault by adding a self-test that checks the success of a loopback RPC at the start of the liveset process.

- If the injected fault renders the cell unable to send messages, it leads to total system failure. This occurs because the cell receives no response to its outgoing pings, concludes the other cells have failed, and resets all other cells. The potential for this to occur if there is a software or hardware problem in the message subsystem argues for adding a heuristic mechanism. If recovery runs and a cell concludes it is the only cell left alive, it should delay before activating the reboot process. If the delay is long enough, the other cells in the system will reset that cell if they are still alive.

- One of the injected faults caused cell 1 to trigger recovery on each clock tick, bypassing the self-check that shuts a cell down if it triggers recovery too many times. It would be possible to defend against this type of fault through an extension to the liveset algorithm. In addition to reaching agreement on the liveset, the cells could vote on which cells should be shut down despite being in the liveset.

The final uncontained failure is due to a Byzantine fault.

- One of the injected faults caused cell 1 to repeatedly request mappings to the same page of cell 0. This is a legal operation, but eventually the reference count in cell 0 overflowed, became negative, and caused internal VM operations on cell 0 to fail. While it is possible to defend against the overflow itself, it is not clear what action cell 0 should take when it detects that the counter would overflow. When the overflow occurs, it is not necessarily the cell currently requesting the mapping that is erroneous.

  In general, it is difficult to defend against non-fail-fast errors that cause cells to issue legal operations in illegal patterns. This type of error violates the error model assumption that the sanity checks applied to incoming messages detect all corrupt requests.

Evaluation: The repeated-recovery fault is technically a Byzantine fault, as are the faults that prevent RPC replies. However, these faults are easily handled by extensions to the recovery system because they affect the recovery and communication substrate.

The overflow error is more challenging than these faults because it arises from a software fault in the higher levels of the system. The Byzantine behavior that it causes is indistinguishable from correct behavior without higher-level semantic knowledge. This is the type of fault that must be assumed to be rare for the Hive design to provide an acceptable level of reliability. The sample size of this experiment is too small to predict the probability of these faults in practice, but it is encouraging that such a fault occurred only once in this series of 1000 fault injections.

Overall, the observed uncontained failure rate of 4% is clearly conservative. Over half of the uncontained failures come from just two software faults in Hive. With a more mature system that implements the recovery system extensions discussed above, a much lower rate should be achievable.

6.5.4 Effectiveness of the wild write defense

Not one of the uncontained failures in the series of 1000 fault injection experiments resulted from a wild write. None of those experiments and none of the error injection experiments corrupted any application output files. This raises the question of whether the experiments demonstrate the effectiveness of the wild write defense or the low probability of wild writes.

Probability of wild writes: Out of the 1000 software fault injection experiments, a firewall write violation provided the first indication of software error in eight cases. In seven of these cases the faulty cell issued the write, while in the eighth case another cell used a corrupt address provided by the faulty cell. Omitting the cases where no cell failed, and assuming that all wild writes in the experiments caused firewall violations, this gives a wild write frequency of 1.3%.

There were an additional eleven cases in which the first indication of error was a write by the faulty cell to a memory address between 128 MB and 512 MB. If the machine had been configured with 512 MB of memory, the system would have had a wild write frequency of 3%.

It is clear that wild writes are a nontrivial problem. It is interesting to compare these results to [CNC+96], in which a similar fault injection methodology led to memory corruption in 11 out of 650 stimulated crashes of Digital UNIX running on an Alpha workstation with 128 MB of memory. Hive shows nearly the same rate, 8 firewall violations out of 612 stimulated cell failures. This correspondence between systems with different code bases and hardware architectures suggests that the results apply more widely than just to the particular implementation measured here.

Usefulness of the firewall: The eight wild writes observed, and the additional eleven that would have occurred on a larger system, would certainly have led to corruption of memory had the firewall not been present. This argues that the firewall provides a useful service for cell isolation.

Usefulness of preemptive discard: Unfortunately, the experiments provide little information about the benefits of the preemptive discard policy. As discussed in Section 6.2.1, Hive's firewall management policy leaves few pages unprotected when running the pmake workload, so there is little chance that a wild write will successfully corrupt memory. However, the rate of wild writes is high enough to create a significant risk of data corruption for workloads that make heavy use of shared pages or that drive the system to do significant intercell load balancing, so preemptive discard appears to be necessary.

Latency to detect errors: The reliability experiments provide approximate timing information, but the system model used (Section 5.3) is not very accurate. To estimate the latency more closely, 50 of the instruction corruption experiments were repeated using a more accurate system model. Both the model and the experiments repeated were chosen to reduce simulation time requirements.

- *System model:* The system model is the one used for the performance experiments, but modified to use a bus-based memory system (1200 MB/sec, 500 nsec access latency when a memory request reaches the head of the memory controller queue). This memory system model is much faster to simulate than the FLASH model.
- *Experiments repeated:* The experiments repeated were chosen randomly from among those where: (1) cell 1 fails but Hive recovers successfully, and (2) the approximate timing shows that the system enters the recovery algorithm within one second after the fault injection. The second constraint eliminates about 50% of the experiments, which are assumed to be those in which the fault does not cause a software error until long after injection. It is possible that some of the experiments eliminated actually represent long delays to detect errors, but the data shows a strong tail-off at much less than one second latency so the number of incorrectly eliminated cases can be presumed to be low.

Table 6.5 shows the distribution of the latencies for failure detection in the repeated experiments. The latency is measured from the time at which the error occurs, which is assumed to be the first time the corrupted instruction is executed, to the time at which the last of the other three cells has begun executing the liveset process.

Over 50% of the experiments show a latency between 3.5 and 7 msec, which is not surprising. Any error that causes a cell to panic is detected when another cell reads the `alive` field of the cell public area (Section 7.2.1) on the next 10 msec clock interrupt, so the expected latency to detect

Table 6.5. Latency from error until last cell enters recovery (50 experiments).

Percentile	Min	25%	50%	75%	Max
Latency (msec)	0.5	3.9	5.5	6.7	109.9

these errors is 5 msec. About 25% of the experiments show a latency less than 4 msec. One common case that triggers this immediate failure detection is corruption of RPC request or reply data by the error. Of the remaining 22% of the experiments, half are randomly distributed between 10 msec and 20 msec and half are randomly distributed between 20 msec and 110 msec. The bias towards the low end of this range may be related to the expected 15 msec latency for the clock monitoring algorithm to detect deadlock (Section 6.2.3).

These times are fast enough to avoid data corruption due to wild writes, at least for the small system sizes used in the fault and error injection experiments. It is unclear whether larger systems will require lower detection latencies. Even if they do, the expected latency to detect errors may drop as the interaction rate between cells increases (Section 5.4.1), while more aggressive error detection mechanisms than those used in Hive can certainly be added to the system (Section 6.2.4). Error detection latency does not at present appear to be a significant problem for the wild write defense.

6.6 Summary

The cell isolation features of Hive are the novel mechanisms that enable use of distributed system fault containment techniques inside a shared-memory multiprocessor. The careful reference protocol and publisher's lock make it safe to read the internal data of other cells. The firewall and preemptive discard policy prevent damage due to wild writes, while the distributed agreement algorithm and fast null recovery allow the system to efficiently detect and correctly recover from cell failures. Finally, the memory fault model defines the separate responsibilities of the hardware and system software in recovering from hardware failures.

Experiments show that these mechanisms implement cell isolation with high probability despite a wide range of faults and errors. The mechanisms have two primary costs. They add overhead to communication between cells, which reduces the efficiency of the operating system, and they assume hardware support that increases the complexity and hence the cost of the machine. The next chapter discusses how Hive reduces the performance overhead of cell boundaries using shared memory, while Chapter 9 considers the question of hardware support in more detail.

Chapter 7
Memory sharing

Once fault containment is provided as described in the previous chapter, a multicellular kernel could be implemented in the same way as any other single-system image distributed system. However, the environment in which it runs differs in a key respect from distributed systems: a multiprocessor provides shared-memory hardware between the cells.

Hive supports use of shared memory across cell boundaries both by applications and by the cells themselves. At the application level it allows processes on multiple cells to share memory (logical-level sharing) and load-balances memory pressure among the cells (physical-level sharing). At the kernel level the cells read each other's data structures and build data structures that cross kernel boundaries.

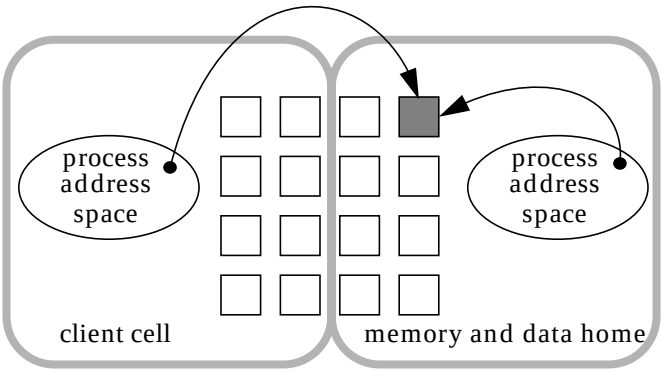
The first section of this chapter describes how the virtual memory system and file system work together to implement application-level memory sharing. The second gives the details of three experiments on memory sharing at the kernel level.

7.1 Application-level memory sharing

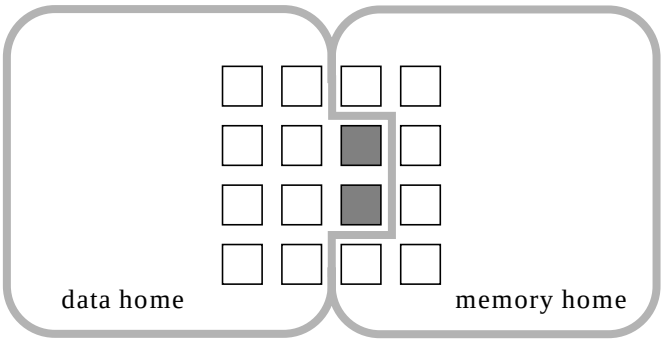
Figure 7.1 repeats a figure from Chapter 3 that shows application-level memory sharing across cell boundaries. There are three roles that cells can play in sharing a page.

- *Client cell:* A cell running a process that is remotely accessing the data. This is the cell on the left in Figure 7.1a.
- *Memory home:* The cell that owns the physical storage for the page. This is the cell on the right in both Figure 7.1a and Figure 7.1b.
- *Data home:* The cell that manages the data stored in the page. This is the cell on the right in Figure 7.1a but the cell on the left in Figure 7.1b.

The data home provides name resolution, manages the firewall if the page is shared, manages the coherency data structures if the page is replicated, and ensures that the page is written back to disk if it becomes dirty. In the prototype the data home for a given page is always the cell that



(a) Logical-level sharing of data pages



(b) Physical-level sharing of page frames

Figure 7.1. Application-level memory sharing across cell boundaries.

owns the backing store for that page, but this is a feature of the filesystem in the prototype rather than a requirement of the design.

I start the description of memory sharing by introducing the virtual memory page cache design in IRIX, because it is the basis for the implementation. Then I discuss each of the types of memory sharing in turn.

7.1.1 IRIX page cache design

In IRIX, each page frame in paged memory is managed by an entry in a table of *page frame data structures* (pfdats). Each pfdat records the *logical page id* of the data stored in the corresponding frame. The logical page id has two components: a tag and an offset. The tag identifies the object to which the logical page belongs. This can be either a file, for file system pages, or a node in the copy-on-write tree, for anonymous pages. The offset indicates which logical page of the object this is. The pfdats are linked into a hash table that allows lookup by logical page id.

When a page fault to a mapped file page occurs, the virtual memory system first checks the pfdat hash table. If the data page requested by the process is not present, the virtual memory system invokes the read operation of the vnode object provided by the file system to represent that file. The file system allocates a page frame, fills it with the requested data, and inserts it in the pfdat hash table. Then the page fault handler in the virtual memory system restarts and finds the page in the hash table.

A page fault that finds the page already present in memory is called a *quick fault*. Quick faults are frequent so their speed is critical to overall system performance.

Read and write system calls follow nearly the same path as page faults. The system call dispatcher calls through the vnode object for the file. The file system checks the pfdat hash table for the requested page in order to decide whether I/O is necessary.

7.1.2 Logical-level sharing

In Hive, when one cell needs to access a data page cached by another cell, it allocates a new pfdat to record the logical page id and the physical address of the page. These dynamically-allocated pfdats are called *extended pfdats*. Once the extended pfdat is allocated and inserted into the pfdat hash table, most kernel modules can operate on the page without being aware that it is actually part of the memory belonging to another cell.

The Hive virtual memory system implements `export` and `import` functions that set up the binding between a page of one cell and an extended pfdat on another (Table 7.1). These functions are most frequently called as part of page fault processing.

A page fault is initially processed just as in other distributed file systems. The virtual memory system first checks the pfdat hash table on the client cell. If the data page requested by the process is not present, the virtual memory system invokes the read operation on the vnode for that file. In Cell-NFS, this is a shadow vnode that indicates that the file is remote. Cell-NFS retrieves the universal file identifier from the vnode and sends an RPC to the data home requesting the page. A more complete file system would have a more sophisticated mechanism, but that mechanism will implement the same function, which is to locate the appropriate data home and send it an RPC. The file system at the data home issues a disk read using the data home vnode if the page is not already cached.

Once the page has been located on the data home, Hive functions differently from other distributed systems. In other systems, a copy of the page is returned to the client. In Hive, the data home returns the address of the page to the client.

Table 7.1. Virtual memory system interface for memory sharing.

Logical level

/* Record that a client cell is now accessing a data page. */
`export(client_cell, pfdat, is_writable)`
 	
/* Allocate an extended pfdat and bind to a remote page. */
`import(page_address, data_home, logical_page_id, is_writable)`
 	
/* Free extended pfdat, send RPC to data home to free page. */
`release(pfdat)`

Physical level

/* Record that a client cell now has control over a page frame.*/
`loan_frame(client_cell, pfdat)`
 	
/* Allocate an extended pfdat and bind to a remote frame. */
`borrow_frame(page_address)`
 	
/* Free extended pfdat, send free RPC to memory home. */
`return_frame(pfdat)`

First, the file system on the data home calls `export` on the page, which records the client cell in the data home's pfdat. This information prevents the page from being deallocated and provides information necessary for the failure recovery algorithms. The `export` function also modifies the firewall state of the page if write access is requested.

Second, the file system on the data home returns the address of the page to the client cell. The client-side file system calls `import`, which allocates an extended pfdat for that page frame and inserts it into the client cell's pfdat hash table. Further faults to that page can hit quickly in the client cell's hash table, making an RPC to the data home unnecessary. The page also remains in the data home's pfdat hash table, which allows processes on other cells to find and share it. Figure 7.2 illustrates the state of the virtual memory data structures after `export` and `import` have completed.

When the client cell eventually frees the page, the virtual memory system calls `release` instead of putting the frame on the local free list. The `release` function frees the extended pfdat and sends an RPC to the data home. If no other references remain, the data home returns the frame to its local free list. Keeping the frame on the data home free list rather than client free lists increases memory allocation flexibility for the data home. The data page remains cached in the frame until the page frame is reallocated, providing fast access if the client cell faults to the data page again.

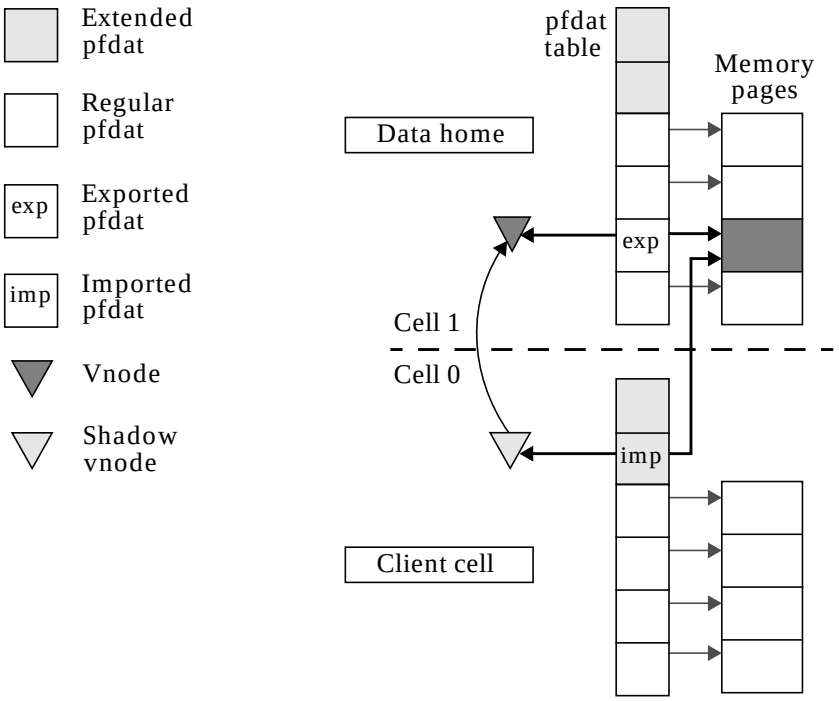


Figure 7.2. Logical-level sharing of data pages.

7.1.3 Performance of logical-level sharing

The overhead of the logical-level sharing mechanism can be measured by comparing the minimal cost of a quick fault that hits in the client cell page cache with one that goes remote and hits in the data home page cache. The local case averages 12.3 μsec while the remote case averages 89.8 μsec when measured using a microbenchmark that issues 1024 remote quick faults in succession.

Table 7.2 shows a detailed breakdown of the remote page fault latency. Each time component is averaged across the 1024 faults in the microbenchmark. 30.7 μsec of the remote case is due to RPC costs. The component of this time spent moving data through shared memory, 16.3 μsec, will be substantially smaller on FLASH than in this measurement since the simulation does not model the lockup-free caches of the R10000. Another 19.3 μsec (listed in the table as miscellaneous VM) is due to an implementation structure inherited from IRIX. IRIX assumes that any miss in the client cell's hash table will result in a disk access, and so does not optimize that code path. Reorganizing this code could provide substantial further reduction in the remote overhead.

In more complex workloads the cost of a quick fault can vary from the microbenchmark results. The local fault handler must acquire several locks and can experience synchronization delays. Some remote faults may take longer if they encounter synchronization conditions at the data home that require them to be queued for an RPC server process. Some quick faults to remote

Table 7.2. Components of the remote quick fault latency.

Total local quick fault latency	12.3 μsec
Total remote quick fault latency	89.8 μsec
Client cell	47.4
File system	19.7
Miscellaneous VM	19.3
Import page	4.4
Interrupt dispatch and return	4.0
Data home	11.7
File system	3.1
Locate and export page	8.6
RPC	30.7
SIPS, interrupts, RPC dispatch, and stubs	14.4
Argument / result copy through shared memory	16.3

pages may take substantially less time if the client cell discovers the page already cached in its pfdat hash table.

For pmake on an eight-cell Hive configuration, these effects combine to make the slowdown due to remote fault latency much less than the microbenchmark suggests. Quick faults to local pages slow down due to synchronization while quick faults to remote pages speed up due to cache hits. Measured over 4.9 seconds of execution on eight processors, in which there are 14146 quick faults to remote pages and 1274 quick faults to local pages, the average quick fault to a remote page takes 73 μsec and the average combining local and remote faults is 70 μsec. This compares to an average quick fault of 26 μsec in the single-cell case running pmake on eight processors.

The overall application pause time from quick faults cumulative across the eight processors in the execution of pmake is 0.4 seconds on the one-cell system and 1.08 seconds on the eight-cell system. This 2.7 times increase is much less than the 7.3 times increase suggested by the microbenchmark. Further optimization is both possible and desirable but the fundamental design appears fast enough to provide performance that is competitive to SMP kernels, as shown by the results of the performance experiments described later.

7.1.4 Physical-level sharing

The logical-level design just described has a major constraint: all user pages in memory must be in their data home's page cache. If this design constrained all pages to be stored in the data home's

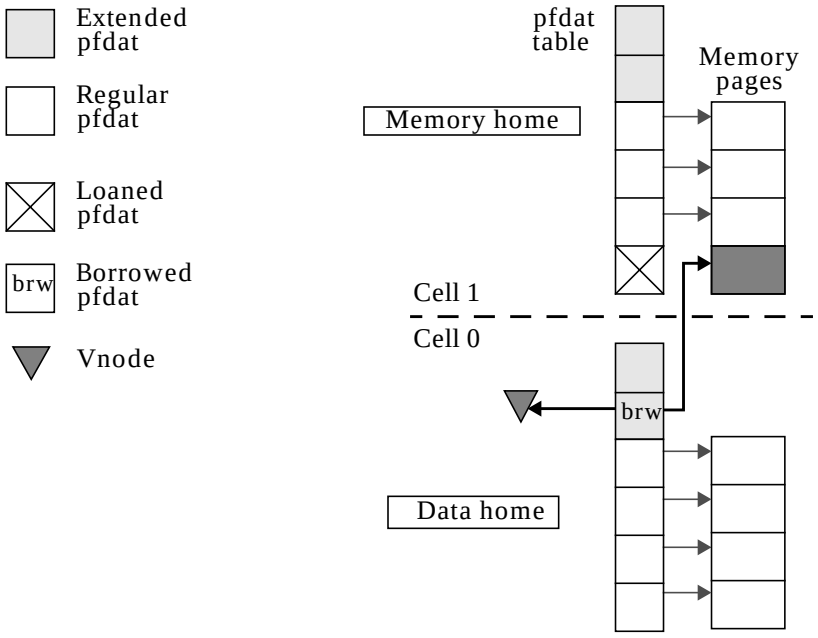


Figure 7.3. Physical-level sharing of page frames.

memory, Hive would have poor load balancing and would not be able to place pages for better locality to the processes that use them, which is required for performance on a CC-NUMA machine. Physical-level sharing solves this problem.

Hive reuses the extended pfdat mechanism to enable the memory home cell to loan one of its page frames to another cell, which becomes the data home (Figure 7.3). The memory home moves the page frame to a reserved list and ignores it until the data home frees it or fails. The data home allocates an extended pfdat and subsequently manages the frame as one of its own.

Frame loaning is usually demand-driven by the memory allocator. When the memory allocator receives a request, it may decide that local free memory is low relative to other cells and it is time to borrow memory from another cell. It chooses a donor and sends it an RPC asking for a set of frames.

Borrowed frames are not acceptable for all frame allocation requests. For example, frames allocated for internal kernel use must be local, since the firewall does not defend against wild writes or hardware errors in the memory home. The memory allocator supports these constraints by taking two new arguments beyond those already in IRIX, a set of cells that are acceptable for the request and one cell that is preferred.

Hive's current policy for freeing borrowed frames is similar to its policy for releasing imported pages. It sends a free message to the memory home as soon as the data cached in the frame is no longer in use. This can be a poor choice because it results in immediately flushing the data. A better approach would be to maintain separate free lists for local and borrowed memory and only return borrowed frames to the memory home when directed to do so by Wax.

7.1.5 Remote firewall management

Initial experience with physical-level sharing demonstrated a performance problem in the design. FLASH initially implemented the straightforward rule that only the memory home could modify the firewall status of a page, requiring the data home to send an RPC request to the memory home to modify the firewall for borrowed pages. However, interrupt handlers in Hive cannot send RPCs (Section 4.3). Therefore this rule implied that client cells could not use interrupt-level RPCs to request data home operations that might need to change the firewall. This added significant process-level RPC overheads to performance-critical virtual memory and file system operations that otherwise could run at interrupt level.

FLASH was modified to implement a rule that allows the data home to manage the firewall directly: a node may modify the firewall state if it has write permission to the page. The memory home initially grants write permission to the data home when loaning it the page.

This rule does not create new wild write vulnerabilities. Consider the possible cases:

- *The data home sets the firewall incorrectly:* This case is the same as with the simpler rule.
- *The memory home sets the firewall incorrectly:* This case is the same as with the simpler rule.
- *A cell without write permission attempts to write to the firewall:* This case is the same as with the simpler rule.
- *A cell with write permission grants write permission to a cell that should not have it:* When the failure of the initial faulty cell is eventually detected, the page will be discarded because the failed cell had write permission. Therefore additional wild writes from other cells are not a problem. The firewall is reset when full recovery runs, as a side effect removing all remote mappings to implement preemptive discard (Section 6.2.2).

  In future multicellular kernels that implement different preemptive discard policies (Section 6.2.4), erroneous firewall grants might become an issue. To avoid the problems they might cause, the data home preemptive discard scan can use the union of the grant set recorded in its VM data structures with the grant set recorded in the actual firewall to determine which cells a given page was vulnerable to. An erroneous cell would have to

overwrite the firewall of the same page more than once, and a wild write would have to occur on some third cell between those two times, for data corruption to survive the preemptive discard.

- *A cell with write permission denies write permission to a cell that should have it:* There are three cases to consider. First, the erroneous cell might deny itself write permission, potentially causing the data home preemptive discard scan to incorrectly preserve the page when the failure of the erroneous cell is detected. Using the grant set in the data home's internal VM data structures, rather than the actual firewall state, for the preemptive discard scan avoids this problem.

  Second, the erroneous cell might deny some other client cell write permission, causing it to receive a bus error. If the recovery algorithms conclude that the erroneous cell has failed, preemptive discard will run and the incorrect firewall state will be cleaned up. If not, the cell that received the bus error will try again, receive another bus error, panic because the hint is again unconfirmed, and cause a full recovery that cleans up the firewall. It would be possible to prevent the panic of the innocent cell under this type of corruption by validating that the actual firewall state of the page causing the bus error matches the VM data structures on the data home.

  Finally, the erroneous cell might deny the data home write permission. The data home could send an RPC to the memory home requesting that the firewall be reset if this occurs.

The prototype does not implement the more complex mechanisms just proposed because firewall corruption has not been observed in any fault or error injection experiments.

7.1.6 Logical/physical interactions

In general, the two types of application-level memory sharing operate independently and concurrently. A given frame might be simultaneously borrowed and exported (when the data home is under excessive memory pressure so it caches pages in borrowed frames). More interestingly, a frame might be simultaneously loaned out and imported back into the memory home. This occurs frequently since the data home tries to place a page in the memory of the client cell that has faulted to it, in order to improve CC-NUMA locality.

To support this CC-NUMA optimization efficiently, the virtual memory system reuses the preexisting pfdat rather than allocating an extended pfdat when reimporting a loaned page. Pfdat reuse is possible because the logical-level and physical-level state machines use separate storage within each pfdat.

7.1.7 Memory sharing and fault containment

Memory sharing allows a corrupt cell to damage user-level processes running on other cells. This has several implications for the system:

- The page allocation and migration policies must be sensitive to the number and location of borrowed pages already allocated to a given process. If pages are allocated randomly, a long-running process will gradually accumulate dependencies on a large number of cells. Some user-tunable policy is needed that trades off optimizing page allocation and migration against minimizing the number of cells that each process depends on.
- The generation number strategy used for preemptive discard makes the file the unit of data loss when a cell fails. Therefore the page allocation and migration policies must be sensitive to the number of different cells that are memory homes for the dirty pages of a given file. This requires similar user-tunable policies.

The tradeoffs in page allocation between fault containment and performance are complex. This is a subject to be studied in depth when the FLASH hardware is available and large workloads can be examined.

7.1.8 Summary of application-level memory sharing

The key organizing principle of application-level memory sharing is the distinction between the logical and physical levels. When a cell imports a logical page it gains the right to access that data wherever it is stored in memory. When a cell borrows a physical page frame it gains control over that frame. Extended pfdats are used in both cases to allow most of the kernel to operate on the remote page as if it were a local page. Naming and location transparency are provided by the file system.

There are no operations in the memory sharing subsystem for a cell to request that another return its page or page frame. The information available to each cell is not sufficient to decide whether its local memory requests are higher or lower priority than those of the remote processes using those pages. This information will eventually be provided by Wax, which will direct the virtual memory clock hand process running on each cell to preferentially free pages whose memory home is under memory pressure.

7.2 Kernel-level memory sharing

In addition to enabling application-level sharing, shared-memory hardware offers the promise of significantly improving the efficiency of the operating system itself. This section reports on three

experiments that give insight into the advantages and costs of kernel-level memory sharing. The three experiments are the implementation of the cell public area, the remote process creation mechanism, and the anonymous memory manager in Hive.

The experiments have moderately negative results. The cell public area works well, but neither remote process creation nor the anonymous memory manager show significant benefits from using shared memory. At the end of the section I discuss ways in which kernel-level shared memory might be important for mechanisms not yet implemented in the prototype.

7.2.1 Cell public area

The cell public area is a data structure in each cell's kernel memory that is used by the fault containment subsystem to publish information about the state and beliefs of that cell. Some of the key fields are:

```
struct PublicArea {
   publ_lock      plock;   /* publisher's lock for PublicArea */
   unsigned int   tick;    /* increments on each clock interrupt */
   int            alive;   /* set to indicate that this cell is up */
   LiveSet        ls;      /* current belief about live cells */
   int            lsgen;   /* current liveset generation number */
};
```

The `tick` field supports the clock monitoring algorithm that is part of failure detection.

The `alive` field is set when the cell has finished booting and is ready to integrate into the system. A cell resets the `alive` field when it panics, thereby notifying other cells in an unambiguous way about the panic without sending messages. This approach has the advantage that a cell that panics repeatedly does not clog the receive queues of other cells with multiple panic messages.

The `ls` and `lsgen` fields allow consistency checks in each cell to verify efficiently that liveset agreement has succeeded for all live cells in the system. If agreement fails, i.e. if some cell in the liveset has an incorrect liveset, the system is prone to significant corruption and should shut down immediately (this is the well-known *split-brain syndrome* [SiS92]).

Evaluation: It was straightforward to integrate information dissemination through shared memory (using the publisher's lock and the careful reference protocol) into the design of the fault containment subsystem. In some cases it significantly simplified the implementation, as in the use of the `alive` field.

One design challenge is the method used by each cell to locate the public area of other cells. Some well-known offset into each cell's memory is needed, at which a cell can publish the addresses of data structures such as the public area that other cells need to read. Since Hive cells all run the same kernel image, which is loaded at the same offset into each cell, the well-known offset for

Hive is the start of the first data page after the end of the kernel boot image. Future multicellular kernels where the kernel boot images can vary will need a different mechanism.

7.2.2 Remote process creation

Process creation is a fairly complex task in which access to the parent-process data structures is primarily read-only. This suggests that kernel-level shared memory could be used to simplify remote process creation. In particular, it seems possible that much of the local process creation code could be reused for remote process creation. Hive implements remote process creation using remote reads to the parent data structures in order to test this hypothesis. The experiment shows that local code cannot be reused for the remote case, at least for this operation.

7.2.2.1 Implementation

Remote process creation proceeds in five steps. I describe them in detail to illustrate the complexities of using kernel-level shared memory. The *parent cell* is the one where the process that requests the fork executes, while the *child cell* is the one where the new process will be created.

Parent prepares to fork: The parent cell chooses a child cell for process creation and allocates a data structure to represent the child process in the parent process' list of children.

Child initializes process table: The parent cell sends the child cell a pointer to the parent process table entry. The child cell uses the careful reference protocol to read it and its associated credentials and user area structures. The child cell uses this information to allocate and initialize the child process table entry and its associated structures. The child cell then returns the child process id to the parent cell.

Two aspects of this step make it slower than the equivalent portion of a local fork, in which the parent process table and user areas are copied directly to the child process.

- The child cell must initialize the process table entry and user area before beginning to copy data from the parent, because a hardware failure in the parent cell during this operation could cause a bus error and immediate jump to the careful reference error handler. When recovery runs, the process table cleanup routines will behave unpredictably if there is stale or uninitialized information in the fields of an allocated process table entry.
- The child cell must validate as far as possible all values read from the parent. A value that fails a sanity-check causes the process creation attempt to abort and the recovery subsystem to be notified that the parent cell may be corrupt.

A third difference from the local process table initialization algorithm is that the child cell cannot copy the kernel stack or registers of the parent process to the child process, because the parent process contains the stack frames of procedures in the parent cell rather than the child cell. The child process needs a local kernel stack and registers. The first local user-level `fork` system call on each cell makes an extra copy of the parent process' stack and registers that is later used for all remote process creation.

Parent identifies files and locks address space: After the child cell returns from initializing its process table entry, the parent cell iterates over all open files of the parent process, storing universal file identifiers (Section 4.4) in an array for the child cell to read. The parent cell also locks all `region` structures of the parent process' address space and makes the first set of the address space modifications required for forking the process. The remainder of these modifications are made after the next child step runs and returns necessary information.

The primary overhead added by this step is that it leaves the `region` structures locked for the next two steps. If any other processes on the parent cell share these structures they will likely stall, as most operations on address spaces acquire at least one `region` lock. The structures are left locked so the child cell can access them directly through shared memory when building a copy of the parent process' address space for the child process.

Child initializes files and address space: The child mirrors the parent's work of the previous step. All files are reopened using the universal file identifiers. The child cell uses the careful reference protocol to read the address space data structures from the parent cell and initialize the child process' address space.

This step includes many operations that potentially send RPCs to other cells. Examples include reopening files, importing cached file and swap pages, and allocating pages on the child cell.

Deadlock can easily occur at this stage. If a failure occurs and recovery begins, recovery on the parent cell will stall trying to access the locked regions of the parent process' address space. Therefore the server process on the child cell must not block waiting for RPC replies from any operation that itself is delayed for recovery. To support this, the operations requested from other cells during this step are designed to fail back to the client if recovery begins.

When the child cell detects that recovery has begun, it aborts and undoes all changes made in this step. After returning to the parent, the parent cell undoes the changes it made in the previous step, releases the region locks, gives up the processor so that the recovery process can run, and goes back to the start of that step.

Parent finalizes address space: The parent finishes the parent-side address space modifications, releases the region locks, and inserts the child process into the appropriate process group and console session. The parent requests the process group and session modifications although the work logically belongs to the child because the parent cell is frequently the home cell for these objects.

At the end of this step the parent cell notifies the child cell that the fork has completed and the child cell adds the new child process to the run queue.

7.2.2.2 Evaluation

Little of the local process creation code survives unmodified in remote process creation. The key problems are cell isolation and failure semantics. Each cell must be prepared for the other to fail or run into resource limitations at any time, which requires careful attention to mechanisms for undoing a partially-completed fork. The deadlock problem in the child files and address space step is another issue that required significant code modification.

This approach to remote process creation might still be desirable if it were fast. Yet it is quite slow in the current prototype, primarily because the attempt to minimize code changes led to reopening files one by one and importing pages one by one rather than batching multiple requests to the file system and virtual memory system. Averaging times across the creation of the seven child processes of the ocean benchmark, the local case takes 1.3 msec while the remote case takes 11.1 msec. Of the 9.8 msec increase, 5.1 msec comes from 18 separate file reopens and 3.4 msec comes from 152 separate page imports, each of which sends an RPC to the parent cell. Even if these performance problems were fixed, which would require substantial code modifications, 0.25 msec of the remaining 1.3 msec slowdown (19%) is RPC overhead in the process creation algorithm itself. This overhead comes from the need to go back and forth between the parent and child cells in order to allocate, lock, and modify data structures at the appropriate points in the algorithm.

With the caveat that the remote process creation implementation in the prototype has not been tuned for performance, it appears that kernel-level shared memory gives no significant advantages for remote process creation. Sufficient code modifications are necessary that it might be simpler and just as fast to use a more traditional distributed-system approach in which the parent cell packages up all the relevant state and sends it to the child cell as a single RPC.

7.2.3 Anonymous memory manager

The third experiment in using kernel-level shared memory is the anonymous memory manager. This represents the most aggressive use of shared memory in Hive: the manager builds a doubly-

linked tree whose links may cross cell boundaries. This experiment also has a negative result: at least for the workloads studied, the implementation complexity is not repaid by significant performance improvements.

Anonymous memory is the set of user data pages whose backing store is in the swap partition rather than in the file system. The anonymous memory manager is responsible for determining which page should be used to satisfy a page fault to a swap region.

In IRIX, the anonymous manager tracks swap pages using a copy-on-write tree, similar to the MACH approach [TeT87]. An anonymous page is allocated when a process writes to a page of its address space that is shared copy-on-write with its parent. The new page is recorded at the current leaf of the copy-on-write tree (Figure 7.4a). When a process forks, the leaf node of the tree is split with one of the new nodes assigned to the parent and the other to the child (Figure 7.4b). Pages written by the parent process after the fork are recorded in its new leaf node, so only the anonymous pages allocated before the fork are visible to the child. When a process faults on a copy-on-write page, it searches up the tree to find the copy created by the nearest ancestor who wrote to the page before forking.

In Hive, different processes might be on different cells, so the anonymous manager must determine not only which ancestor wrote the page but which cell owns the page.

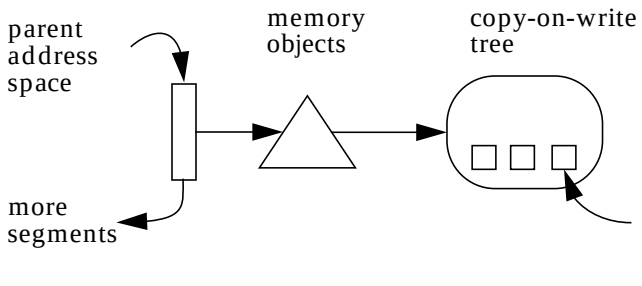
7.2.3.1 Implementation

Hive keeps the tree structure inherited from IRIX nearly intact, but allows the pointers in the tree to cross cell boundaries. When a process forks, the leaf node created for the child process is local to that process but the higher nodes in the tree may be remote (Figure 7.4c). This does not create a wild write vulnerability because information about newly written pages is only recorded at the leaves of the tree, so interior nodes are not modified and can remain protected by the firewall.

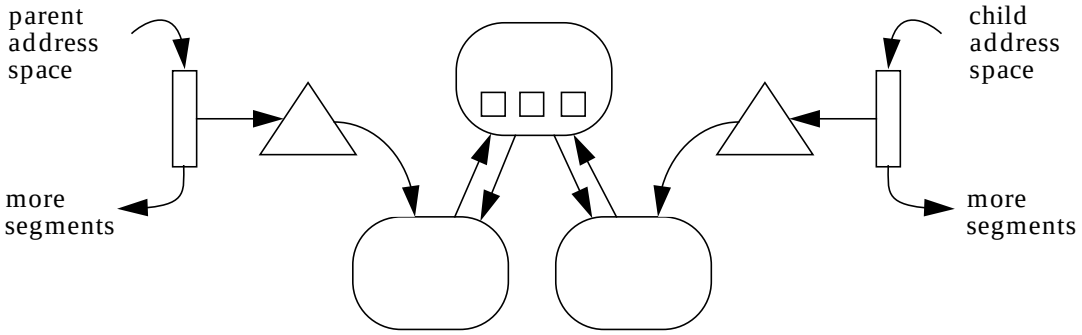
When a child process faults on a shared page, it searches up the tree, potentially using the careful reference protocol to read from the kernel memory of other cells. If it finds the page recorded in a remote node of the tree, it sends an RPC to the cell that owns that node requesting access to the page. The cell that owns the node is always the data home for the anonymous page.

Several implementation details make the remote case more complicated than the local case and reduce performance.

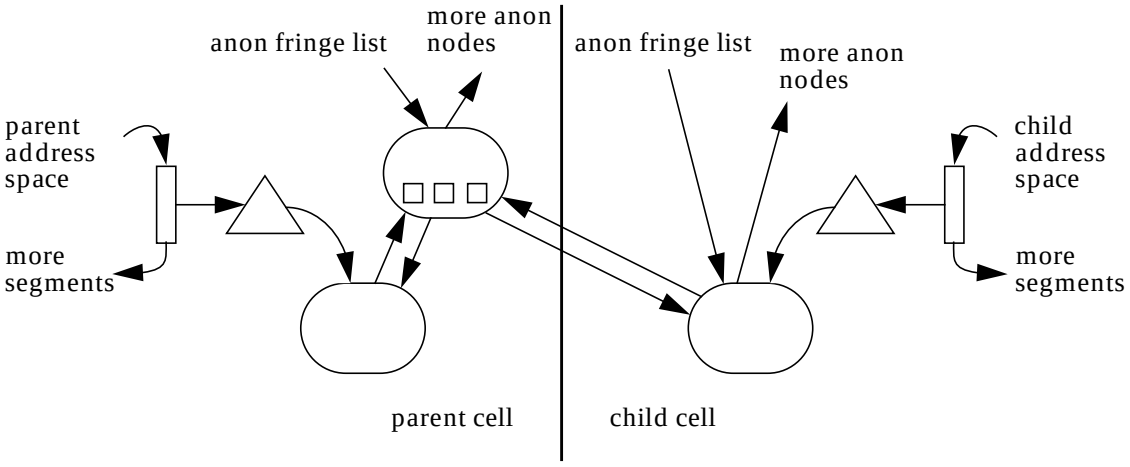
- Recovery must not run on any cell while remote lookup operations are in progress. Recovery may result in deleting interior nodes and thereby cause pending lookups to see stale data. To



(a) Data structures before process creation.



(b) Data structures after local process creation.



(c) Data structures after remote process creation.

Figure 7.4. Anonymous memory manager data structures.

avoid this problem, the lookup routine acquires a lock that delays recovery until the lookup has completed.

- To allow the recovery routine to find dangling references to failed cells, all nodes with either remote parents or children are linked onto a list called the *anon fringe list*. Managing this list requires synchronization among operations that modify the anonymous trees.

- In the local case, when a process exits or removes its mapping to a memory object, the corresponding anonymous tree can be collapsed. This frees both the kernel memory used for the tree and the swap space for the pages it records. However, nodes with remote children cannot easily be collapsed since the address of the node is stored in several child data structures. For example, the node address is a component of the hash key used to look up pages in the child's file cache. Tree collapsing is delayed until all child processes have exited, resulting in greater memory and swap space consumption.

- Kernel virtual memory addresses used by one cell cannot be translated directly on another. Hive avoids this problem by using only physical addresses for links in the anonymous trees, which is only a partial solution. Future multicellular kernels may run each cell in a virtual address space to improve fault containment, so new mechanisms will be needed to translate remote virtual addresses.

7.2.3.2 Evaluation

The fact that the anonymous memory manager appears to work reliably in the fault and error injection experiments described in the previous chapter indicates that distributed data structures can be built without weakening fault containment. To stress the anonymous memory manager more directly, an additional 15 kernel heap corruption experiments were performed using the raytrace workload, which uses anonymous memory heavily. Nodes and pointers in anonymous trees were corrupted in pathological ways just before remote lookups to those trees. The cell doing the lookup successfully defended itself in these experiments.

However, there does not appear to be any substantial performance benefit from the distributed data structures. When the child finds a desired page it usually has to send an RPC to bind to the page in any case, so the use of shared memory does not save much time unless the tree spans multiple cells. Trees that span multiple cells are rare because they are only created when a process forks multiple times, those forks go remote, and none of the child processes use `exec` to start a new program. A more conventional RPC-based implementation of anonymous memory management would be simpler and probably just as fast.

7.2.4 Summary of kernel-level memory sharing

The prototype gives moderately negative results from the memory sharing experiments. On the positive side, memory sharing is quite useful for information dissemination such as in the cell public area. Similar results were obtained from the memory load balancing algorithms in the prototype, which overcomes the lack of Wax by reading memory usage information remotely. This suggests that other forms of information publishing could also enhance performance. For example, cells could publish translations for their internal virtual addresses, making it efficient for remote cells to walk virtual pointer chains (as long as virtual translations used in remotely-read data structures are valid for minimum time durations, which can be implemented using techniques for type-stable memory management [GrC96]).

In contrast, neither of the two more-aggressive experiments with kernel-level memory sharing appears to be successful. The remote process creation subsystem could not reuse local process creation code, and the anonymous memory manager does not on average save any significant time.

Kernel-level shared memory may turn out to be important in two areas not explored yet in the prototype, spanning tasks and the file system. Communication latency is likely to be a problem for spanning tasks because of the tight coupling between the component processes on separate cells. Use of shared memory may be the key mechanism that makes spanning tasks efficient. It may even be possible to use writes across cell boundaries to update spanning task data structures, because the essential dependencies rule means that loss of those data structures in a cell failure is acceptable (Section 3.3.1). Similarly, the file system may be able to use information publishing to reduce the number of RPCs it sends, helping it to achieve execution efficiency competitive with the file system of an SMP kernel.

Chapter 8
System performance

This chapter concludes the description of Hive's implementation by analyzing the results of the performance experiments. The first section characterizes the performance of the prototype, while the second considers the implications of the trends observed in the prototype. As discussed earlier, these results show trends that indicate the possible behavior of multicellular kernels on larger systems, but do not allow definitive conclusions because of the small system sizes studied and the limited feature set of the prototype.

8.1 Performance characterization

Figure 8.1 compares Hive's performance in various configurations against IRIX 5.2 running on the same simulated eight-processor FLASH multiprocessor. For each of the three workloads, the X axis shows the performance of IRIX (labeled as 0), compared to Hive running with one, two, four, or eight cells. The Y axis gives wallclock time, so the height of each bar gives overall performance. Within each bar the sections show the fraction of overall time spent in the corresponding mode. Because there are eight processors, the time spent in each mode during execution of the workload is eight times the height of the corresponding section on the Y axis.

This figure shows several interesting trends. The overall performance of Hive at eight cells is competitive with IRIX for pmake and ocean and slightly degraded for raytrace.

Table 8.1. Wallclock time to completion (seconds).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	4.9	4.9	6.3	5.3	4.9
raytrace	3.1	3.0	3.5	3.5	3.8
ocean	4.0	3.5	3.8	3.0	3.8

Within that overall performance, the amount of kernel work increases as the number of cells increases, but kernel overhead (cache misses, synchronization, and RPC costs) drops significantly. The time spent executing applications is unchanged in pmake but increases with increasing number of cells in raytrace and ocean. Idle time remains constant for raytrace, while decreasing

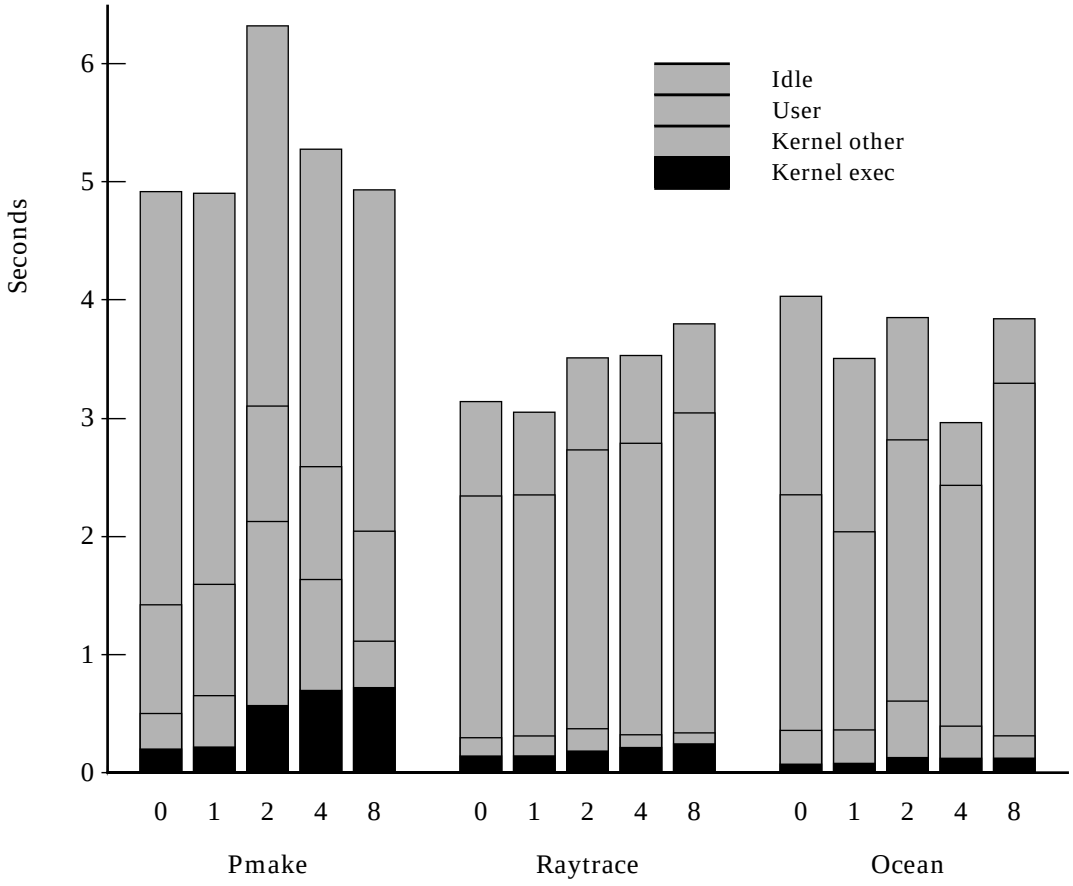


Figure 8.1. Time to completion of workloads.

moderately for pmake and significantly for ocean. The next sections discuss the trends in kernel work, kernel overhead, application time, and idle time in more detail.

8.1.1 Increasing kernel work

As the number of Hive cells increases, the amount of work done by the operating system to support the same workload increases. This is the time labeled *Kernel exec* in the figure, and is

computed as the total non-idle time in the kernel minus time spent stalled on cache misses, spinning on kernel locks, or spinning waiting for an RPC result.

Table 8.2. Total useful kernel time (seconds).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	1.6	1.7	4.5	5.5	5.7
raytrace	1.1	1.1	1.4	1.7	1.9
ocean	0.5	0.6	1.0	0.9	0.9

The big jump in all workloads is the transition from one cell to two cells, when RPC overheads and the software error checks required for fault containment are added to the system. After this jump, useful kernel work continues to increase but total time spent in the kernel decreases significantly.

Table 8.3. Total non-idle kernel time (seconds).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	4.0	5.2	17.0	13.0	8.9
raytrace	2.3	2.4	2.9	2.5	2.6
ocean	2.8	2.8	4.8	3.1	2.4

Total time spent in the kernel decreases as the system grows from two to eight cells because the overhead of kernel execution drops.

8.1.2 Decreasing kernel overheads

Kernel overhead is the time labeled *Kernel other* in the figure, computed as the total time spent stalled on kernel cache misses, spinning on kernel locks, and spinning waiting for RPC replies. The numbers shown in the tables below are computed as overhead time divided by the useful work time shown in Table 8.2. However, these times are added on to the useful work time, not components of it.

Cache miss stall overheads drop dramatically as the number of cells increases.

Table 8.4. Kernel time stalled for cache misses (percent of useful kernel time).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	45	46	49	33	16
raytrace	47	49	42	20	9
ocean	70	66	60	38	17

This is not surprising because the partition into cells reduces several memory system costs simultaneously. It increases the fraction of kernel accesses which go to local data structures, reducing the expected latency. It spreads kernel data accesses across more nodes, reducing queuing delays in MAGIC. Finally, it reduces the number of processors sharing each data structure, which reduces the number of communication misses.

Because there are fewer processors in each cell, synchronization overheads also decrease as the number of cells increases.

Table 8.5. Kernel time spinning on locks (percent of useful kernel time).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	38	61	59	26	9
raytrace	15	13	15	8	6
ocean	63	70	65	37	8

The increase from IRIX to the single-cell Hive configuration, visible in pmake and ocean, is due to increased synchronization added to the page fault handlers to support multicell systems.

The last overhead cost is the time spent waiting for RPC replies. In general this goes up as the number of cells increases, decreasing the fraction of kernel operations that can be completed locally.

Table 8.6. Kernel time waiting for RPCs (percent of useful kernel time).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	—	—	33	31	22
raytrace	—	—	6	14	21
ocean	—	—	30	70	111

The overhead actually drops for pmake because of the large savings in cache miss and synchronization time as the number of cells increases. The latency required for server cells to handle the RPCs drops faster than the number of RPCs increases.

The RPC overhead for ocean is large because the application generates many quick faults and little other kernel work, so cells other than cell 0 spend most of their kernel time on RPC waits.

8.1.3 Increasing user time

Surprisingly, the absolute time spent in user mode in the parallel scientific workloads increases noticeably as the number of cells increases.

Table 8.7. Total application time (seconds).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	7.4	7.5	7.8	7.6	7.4
raytrace	16.4	16.3	18.9	19.7	21.7
ocean	15.9	13.4	17.7	16.3	23.9

The application time increase in raytrace and ocean is due to a convoying effect. The multiple processes in these applications are tightly synchronized. If a process is slow reaching the next barrier, or is interrupted while holding a lock, the other processes spin waiting for it.

The initial jump in application time from one to two cells occurs because of the increased latency of kernel operations for the processes on cell 1 as opposed to the processes on cell 0. For example, the kernel operation that consumes the most time in raytrace running on two cells is the quick fault, which averages 69 μsec on cell 0 and 124 μsec on cell 1.

Subsequent increases in application time as the number of cells increase are due to the increasing skew of kernel execution to cell 0. Cell 0 is the data home for the memory segment shared by all the processes and therefore services many VM operations requested by the processes on other cells. This leads to both a higher frequency of kernel operations on cell 0 and a higher TLB miss and cache miss rate caused by increased kernel activity, all of which slow down the application process running on cell 0.

Table 8.8. Most frequent kernel operations in eight-cell Hive running raytrace.

	Cell 0		Average of other 7 cells	
Operation	Number	Avg cycles	Number	Avg cycles
TLB miss	630114	21	385694	18
`release` (Table 7.1)	5074	3412	0	0
dispatch queued RPC (primarily quick faults)	4248	9549	3	3289

The convoying effect does not appear in pmake because the separate compile processes do not synchronize with each other.

8.1.4 Decreasing idle time

Pmake shows a moderate decrease in idle time, raytrace shows no effect, and ocean shows a significant decrease as the number of cells increases.

Table 8.9. Total idle time (seconds).

Workload	IRIX 5.2	1 cell	2 cells	4 cells	8 cells
pmake	28.0	26.5	25.7	21.5	23.1
raytrace	6.4	5.6	6.2	5.9	6.0
ocean	13.4	11.7	8.3	4.2	4.4

Pmake does many file creations and deletions, each of which requires a synchronous metadata update, so its initial idle time is due to disk waits. The decrease in idle time occurs because the system overlaps the additional kernel work and overhead generated by the multiple cells with disk wait time.

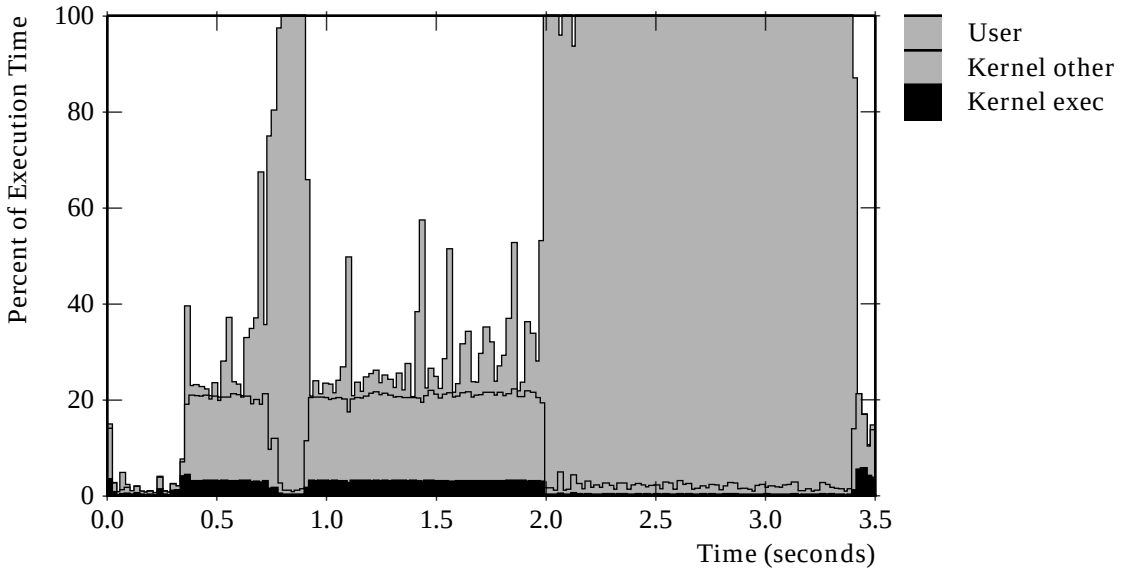
The decrease in the idle time of ocean (as well as the strange behavior it shows in Table 8.7) is a side-effect of its use of the IRIX user-level spinlock library. The execution trace of ocean on a one-cell system (Figure 8.2a) shows a period of execution (between 0.3 seconds and 2.0 seconds wallclock) when little application work is being done.

This poor behavior occurs because the application's threads spin on a user-level lock while the master process initializes the computation. The spinlock library issues a system call to deschedule the process whenever a thread fails to acquire the lock. When multiple threads make this system call repeatedly, they conflict on an internal lock in the process scheduler and serialize the system. The four and eight cell Hive configurations avoid the scheduler lock conflict, reducing the amount idle time (Figure 8.2b).

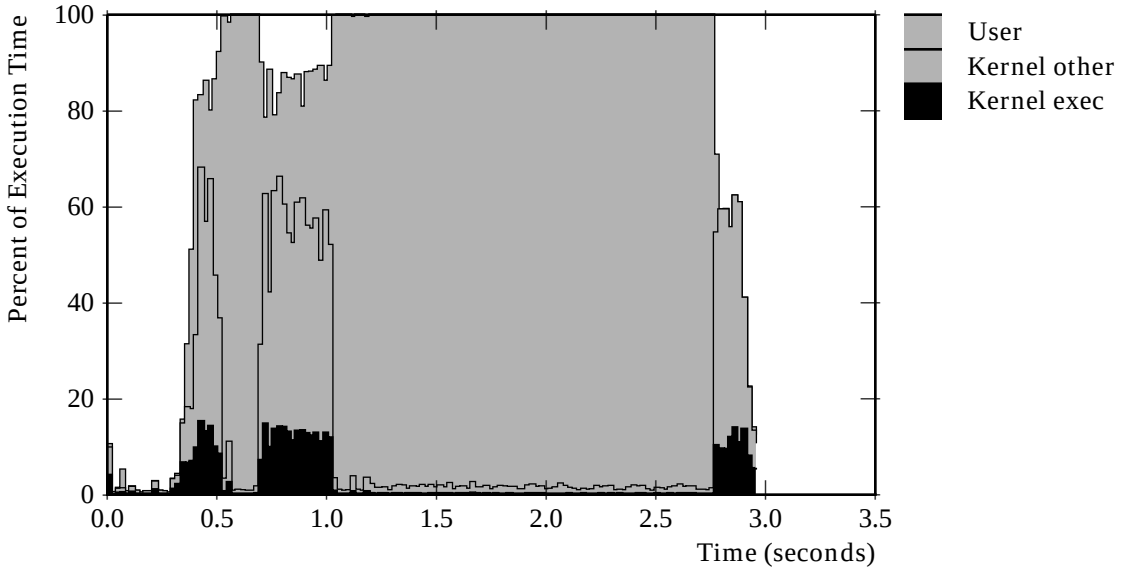
8.2 Evaluation

The above performance trends suggest the possible behavior of multicellular kernels on larger systems. The first trend, increasing kernel work with increasing numbers of cells, agrees with the intuition that there are fundamental overheads caused by replicating the kernels. However, the small increases in useful work time from two cells on up suggests that the rate of increase should be small, at least at the low stress levels caused by these workloads.

These performance costs are mitigated by the second trend, the significant decrease in kernel overhead that occurs as the system is partitioned into an increasing number of cells. This is exactly the type of benefit that the multicellular architecture is designed to provide.



(a) Running on one cell



(b) Running on four cells

Figure 8.2. Execution trace of ocean.

The transition from one to two cells provides little reduction in overhead. This is because there is little change in memory system or synchronization behavior when switching from eight processors in a cell to four processors in a cell, at least given the level of parallelization and data structure replication in the current IRIX implementation. This explains why pmake performs worst in the two cell case: the system must pay the added costs of crossing cell boundaries and

replicating work without gaining the benefits of reduced overhead. These benefits appear when the number of processors per cell drops to two or one.

There is no fundamental reason (at least on FLASH) that two processors per cell are better than four. However, a uniprocessor kernel is substantially more efficient than a multiprocessor kernel [RHW+95]. For example, it can eliminate communication cache misses and all spin lock costs, as well as avoiding CC-NUMA remote cache miss latencies for all local pages. (The 6–9% residual lock spin time at eight cells shown in Table 8.5 is the overhead of acquiring and releasing locks; it remains because the kernel was not recompiled in uniprocessor mode.) One interesting question to explore on larger machines is whether these benefits continue to provide overall performance advantages to a system configured with uniprocessor cells, or whether the reduced load balancing efficiency and increased kernel resource utilization of uniprocessor cells create costs that justify using larger cells.

The idle time behavior of ocean is interesting. The developers of the IRIX user-level synchronization library added a deschedule system call to optimize performance. This turned out to interact pathologically with a lock in the scheduler when driven by certain workloads. The multicellular kernel provides a more systematic parallelization of the scheduler and so eliminates the pathological case. This is a good example of the design benefits that the multicellular kernel is intended to provide.

Finally, the magnitude of the convoying effect on the scientific applications was a surprise. The increase in total kernel time in raytrace from one cell to eight cells is minimal, but the skew in operation latency and interrupt frequency leads to substantial increases in application run time. This has implications for the design of the file system and virtual memory system. They must be able to distribute the processing load evenly across the cells, even in cases where all the activity is going to the same file.

Chapter 9
Architectural evaluation

Having finished the description and evaluation of the Hive prototype, I now return to the larger question of the costs and benefits of the multicellular architecture. I consider the question at three levels.

First, given a multicellular kernel, there are other possible tradeoffs among performance, reliability, and implementation complexity than those made in designing Hive. Section 9.1 considers the hardware support required for a multicellular kernel, while Section 9.2 considers several improvements to the functionality implemented in Hive.

Second, given a large-scale multiprocessor, the system designer could choose either to invest in a multicellular kernel or to scale up an existing SMP kernel. Section 9.3 compares multicellular kernels and SMP kernels as operating systems for large-scale multiprocessors.

Finally, there are fundamental questions about the multicellular architecture itself. Section 9.4 summarizes the limitations of the architecture and Section 9.5 lists open questions not answered by this dissertation.

9.1 Hardware support

One of the major costs of the Hive design is the nonstandard hardware support it assumes from the multiprocessor that it runs on. However, this is a property of Hive in particular, not a general property of multicellular kernels.

Summary of FLASH hardware: The various hardware features added to FLASH to support Hive have been described throughout the dissertation:

- *Hardware fault containment:* The memory system, network, data link level, and physical design of FLASH support the memory fault model (Section 3.2, Section 3.4, and Section 6.3). This includes significant amounts of microcode and careful physical design but only a small amount of custom logic. The routers contain logic dedicated to recovering from hardware errors, but this is not considered custom because it was designed without knowledge of the Hive memory fault model [Gal96].

- *Software fault containment:* The memory system provides the firewall (Section 3.2.3, Section 6.2.1, and Section 7.1.5). The firewall includes a small amount of custom logic plus data storage of 64 bits per 4 kilobyte page (0.2% memory overhead).
- *Communication:* The memory system provides the SIPS primitive to support Hive RPCs (Section 4.3.1). This is implemented using only microcode.
- *Miscellaneous:* FLASH provides the remap region to support running multiple cells (Section 4.2), implemented with custom logic. It also provides physical-level support for issuing a hardware reset to one cell at a time.

All of these features add design cost and small amounts of manufacturing cost. There are three performance costs:

- The firewall check slows down cache misses that request exclusive copies across cell boundaries.
- Assertions in the protocol microcode increase the latency of operations.
- Reserving two virtual lanes in the interconnect for network-level recovery (Section 3.4) prevents their use for congestion reduction or additional communication primitives.

None of these costs is significant. The firewall has no overall impact on measured application performance (Section 6.2.1). Microcode assertions can be omitted from performance-critical operations (Section 3.2.3). Finally, the lanes reserved for recovery cannot be used for congestion reduction in FLASH because the coherence protocol relies on in-order delivery of network packets. The potential performance benefits of using these lanes for other communication primitives are unknown.

Reducing hardware requirements: The reliability features added to FLASH fall into two categories: those that support hardware fault containment and those that support software fault containment. Leaving each of these types of features out in turn generates a reasonable design point with less hardware support than FLASH.

If Hive and FLASH are called the *hardware fault containment* design point, then the two others are:

- *Hardware-supported software fault containment:* The multiprocessor provides a firewall for reliability and perhaps a communication primitive for performance, supporting a multicellular kernel with the same software reliability and performance characteristics as Hive but without hardware fault containment. This eliminates most of the hardware

complexity added to FLASH to support Hive, but should still provide significant reliability benefits at small and medium system sizes where the hardware error rate is acceptably low.

- *Software-only fault containment:* The multiprocessor provides no custom hardware or only a communication primitive for performance. This requires running a trusted microkernel or monitor that controls access to physical memory and provides firewall functionality to support the multicellular kernel. This approach has a reliability cost since a software error in the microkernel or monitor will cause a system failure, and a performance cost due to increases in the TLB miss rate and interrupt dispatch latency.

These design points represent three different ways to support a multicellular kernel that trade off among hardware complexity, reliability characteristics, and performance of the multiprocessor. This flexibility suggests that hardware requirements are not a fundamental barrier to the usefulness of the multicellular architecture.

9.2 Additional operating system functionality

Hive is a relatively simple multicellular kernel. There are several ways in which significant additional functionality could be added to the design without changing the fundamental architecture. These include support for heterogeneity, administrative partitions, and integration with distributed-system clustering mechanisms.

Heterogeneity: Hive assumes that all cells run the same kernel code. Other multicellular kernels may wish to allow greater heterogeneity, for two reasons:

- *Incremental software upgrades:* One problem for current multiprocessors is that the entire machine must be restarted to upgrade the operating system software. Multicellular kernels can support kernel software upgrades that are transparent to applications, using the same mechanisms as used for transparent hardware maintenance (Section 3.3.8). While the upgrade is in progress, different cells will be running different versions of the kernel code.

- *Incremental hardware upgrades:* One of the main advantages of large multiprocessors is a smooth growth path for customers starting from smaller machines whose applications grow over time. To make this economical, the customer should be able to continue running older processors and boards as new components are added to the system. This suggests that different cells should be able to run with different generations of hardware. The hardware may vary sufficiently that different kernel code in different cells is desirable or necessary.

The key problem with allowing heterogenous cells is that different versions of the kernel software may have different data structure layouts, making it more complicated for one cell to read the

internal state of another. This problem can be solved by limiting the data structures that can be read remotely to those that are invariant across versions, implementing self-describing data structures, or using an object-oriented design scheme where later revisions of data structures are subclasses of earlier revisions. Some combination of all these mechanisms may be required to achieve a complete solution.

Administrative partitions: Hive follows the UNIX tradition of assuming that the resources in the machine form a common pool usable equally by any process. However, large multiprocessors are expensive enough that they may be shared by multiple administrative domains, so it may be desirable to partition the resource pool administratively. For example, if one department contributes 25% of the machine's cost, it may wish to guarantee that its applications are allocated no less than 25% of the hardware resources when the machine is loaded. This suggests that advanced accounting and prioritization mechanisms may be helpful for large multiprocessors to succeed in commercial environments. Wax provides a natural place in which to implement this functionality.

Integration with clustering: Hive assumes that the distributed single-system image it presents to applications and users ends at the boundaries of the multiprocessor. However, it would be natural to integrate multiprocessors running multicellular kernels into a cluster of machines with an overall single-system image. It is an open question how many of the mechanisms required to implement the single-system image can be shared between the inter-cell and the inter-machine cases.

9.3 Comparison to SMP kernels

It is clear that a multicellular kernel has fundamental reliability advantages compared to an SMP kernel. The cellular structure confines the effects of all errors except those of limited types, whereas reliability mechanisms added to an SMP kernel recover only from the specific types of errors for which they are designed.

However, the tradeoffs in scalability and implementation complexity between multicellular and SMP kernels require more discussion.

Scalability: Multicellular kernels and SMP kernels face completely different scalability challenges. A multicellular kernel faces three performance challenges as the number of cells increases:

- The potential for uneven distribution of kernel work across the cells increases, which can hurt application performance as described in Section 8.1.3, so resource sharing and management mechanisms must be improved to distribute work more evenly.

- RPC delays and the potential for congestion at server cells increase, so the amount of state cacheable in client cells or accessible through shared memory must be steadily increased.
- The simple distinction between cell-local and remote hardware resources becomes less sufficient to achieve locality of resource access, so some higher-level unit containing multiple cells needs to be identified and used for resource allocation.

An SMP kernel faces three different challenges as the number of processors increases:

- The cache lines of widely-shared data structures become memory system hotspots, so data structures must be partitioned into components that are not widely write-shared to reduce memory system costs.
- Kernel locks become contended and cause significant queuing delays, so data structures and algorithms must be improved to use finer-grained locking.
- Locality of resource allocation and process scheduling must be improved to reduce memory system costs incurred by applications.

The key difference between these two sets of challenges is in the number of potential trouble spots. In an SMP kernel every data structure and every kernel lock is a potential trouble spot, which is a much higher number of problems to consider than in the multicellular kernel, where only the operations that cross cell boundaries are potential scalability limitations. Moreover a multicellular kernel naturally exploits memory system locality in a CC-NUMA multiprocessor, while modifying an SMP kernel to exploit locality requires significant effort. These factors argue that multicellular kernels are more easily scalable than SMP kernels.

Complexity: A multicellular kernel is undoubtedly more complex than the SMP kernel on which it is based. The primary complexity differences lie in three areas. First, there are the extensions to the various kernel subsystems required to implement the single-system image. These require creative implementation to reduce overheads and substantial new functionality to preserve correct semantics despite potential failure. Second, there are the mechanisms and policy for resource sharing among cells. Finally, achieving sufficient reliability and performance from the file system requires significant design work.

However, the relative simplicity of SMP kernels may be lost when they are extended to improve reliability and scalability on large multiprocessors. Assuming that software faults remain in the system and therefore software errors must be tolerated, improving reliability requires adding code that checks and repairs internal data structures as in the Tandem Nonstop-UX kernel [Jew91]. If done thoroughly, this approach promises to add comparable or greater complexity than

integrating these same data structures into a distributed single system image, with the disadvantage that only the specific errors checked for can be tolerated.

It is more difficult to judge the impact of improving performance scalability on an SMP kernel. Improving scalability under a simple workload does not increase complexity substantially, as the primary synchronization and communication problems usually lie in a few data structures and algorithms that can be improved without affecting much of the system. However, if the goal is efficient performance under complex or diverse workloads, many or most kernel data structures and algorithms may become substantially more complex to provide finer-grained locking and reduce communication cache misses.

These changes reduce the relative difference in complexity but are unlikely to make an SMP kernel as complex as a multicellular kernel. However, the multicellular architecture justifies its complexity by providing substantially better reliability than is implementable with an SMP architecture, and by making performance scalability easier to achieve.

9.4 Limitations of the architecture

Although the multicellular kernel architecture provides significant reliability and scalability advantages, it faces fundamental limitations in both the level of reliability and the performance that it can provide.

There are three limitations to the reliability achievable with the architecture.

- Multicellular kernels do not provide reliability with respect to non-fail-fast hardware errors. In a tightly-coupled multiprocessor, especially one with a complicated memory system such as FLASH, these errors may be more probable than in the distributed systems where fault containment has previously been implemented.
- Multicellular kernels periodically pause user-level processes while the distributed agreement protocol checks for faults. Even though these interruptions are brief, they may not be acceptable in systems with tight scheduling deadlines such as video servers.
- Multicellular kernels do not provide complete protection against wild writes. In a system running any feasible set of software error detection mechanisms, a modern processor will execute millions of instructions before an error that does not trigger an internal assertion is detected by an external observer. The possibility of data corruption resulting from such errors is a function of the number of pages unprotected by the firewall and the frequency with which the data is accessed by applications.

Additionally, there are two fundamental limitations that prevent a multicellular kernel from achieving performance competitive with a sufficiently-parallelized SMP kernel.

- The policy modules, running in Wax, are “out of the loop” of the allocation decisions made inside the cells and hence respond more slowly to changes in the system state than the policy modules of an SMP operating system. This policy delay reduces resource utilization efficiency.
- Any resource utilization inefficiency, whether from the policy delay or from imperfect resource sharing mechanisms, leads to unbalanced kernel workload among the cells. This increases the run time of tightly-synchronized parallel applications through the convoying effect discussed in Chapter 8.

9.5 Open questions

There are several parts of the architecture whose strengths and weaknesses have not yet been evaluated in the Hive prototype.

- *Wax:* Once Wax is implemented, it will be necessary to investigate how rapidly it can respond to changes in the system state, without running continuously and thereby wasting processor resources. It is also unknown how closely a two-level optimization architecture (intracell and intercell decisions made independently) can approximate the resource management efficiency of an SMP kernel.
- *Resource sharing:* Policies such as page migration and intercell memory sharing must work effectively under a wide range of workloads for a multicellular operating system to be a viable replacement for a current SMP operating system. Spanning tasks and lightweight process migration must be implemented and their efficiency measured. The resource sharing policies must be systematically extended to consider the fault containment implications of sharing decisions. More sophisticated statistical measures are needed to predict the probability of uncontained failures and data integrity violations in production operation.
- *Networking:* Efficient networking is an important requirement for large multiprocessors. New intercell sharing mechanisms will be necessary to satisfy the high bandwidth and low latency demands of applications such as web and multimedia servers.
- *File system:* A multicellular architecture requires a fault-tolerant high performance file system that preserves single-system semantics. This is a major design challenge and could prevent the system from achieving either its reliability or its scalability goals.

These questions remain open because the research reported in this dissertation is only a first step. Now that the fundamental cell isolation and resource sharing mechanisms have been shown to work, providing fault containment with roughly competitive performance, it is necessary to investigate whether the higher-level aspects of the architecture can preserve these advantages in large-scale systems.

Chapter 10
Related work

Reliability and scalability have long been major concerns of researchers and developers. Work on dependable operating systems is the primary source for mechanisms that improve reliability beyond that of SMP operating systems, while work on massively parallel computers and distributed systems provides mechanisms for improving scalability.

Space constraints limit this survey to the work most relevant to the problems of large shared-memory multiprocessors. The first two sections discuss systems and techniques for improving the reliability and scalability of multiprocessors. The second two survey work on tightly-coupled distributed systems and on failure models for multiprocessors.

10.1 Improving SMP kernels

Reliability: There are many techniques for reducing the rate or impact of software errors in SMP kernels. All of these techniques could be applied to the individual cells of a multicellular kernel to improve reliability.

One large area of work has been software engineering techniques. Object-oriented designs such as Choices [CIR+93] and Spring [MGH+94] improve the modularity of the system by adding strong internal interfaces, so the operating system is easier to understand and maintain and hence should have fewer software faults over time than a traditional monolithic kernel. An orthogonal approach has been to partition the system into a microkernel and a set of services running in independent address spaces, making the system easier to test and debug. Examples of this approach include Hydra [WLH81], Mach [RJO+89], and Chorus [RAA+88]. Finally, automated testing [SaH94] can increase the fraction of faults that are found and fixed during operating system development.

Another area of work has been techniques for avoiding the reboot that is required to recover from software errors in standard SMP kernels. Both the Nonstop-UX kernel for Tandem Integrity S2 [Jew91] and the IBM MVS/XA system [MoA87] provide data-structure-specific repair routines invoked when an inconsistency is detected. Another approach, implemented in MVS/XA and Fault-Tolerant Mach [RSS93], is to abort the operation in progress when an error is detected and

retry it. [HJK93] provides an excellent analysis of the reasons why an aborted operation frequently succeeds when retried.

Finally, there has been significant work on improving the reliability of applications despite the potential failure of the systems they run on. Common approaches include checkpointing [LiS92, LNP94, PBK+95] and replication of processes on top of microkernels [LiR93, ACC+93].

Scalability: Few researchers have investigated the techniques required to support general-purpose workloads on large-scale shared-memory multiprocessors, because these machines have only recently become available for general-purpose use. [CDV+94] and [VDG+96] investigate algorithms for page replication and migration and how these must work together with processor scheduling to reduce memory system costs. [CHR+95] characterizes the performance bottlenecks of an SMP kernel running on a CC-NUMA system. [SDH+96] describes a new file system created by Silicon Graphics to support general-purpose workloads that access very large amounts of data.

10.2 Multiprocessor operating systems

Multicellular architecture: The scalability benefits of the multicellular architecture for multiprocessor operating systems have been investigated in an ongoing project at the University of Toronto, which calls this design the *hierarchical clustering* architecture. Their initial work on the Hurricane operating system [UKG+95, Kri95] running on the Hector multiprocessor [VSL+91] has been followed by current work on the Tornado operating system [PGK+95] running on the NUMAchine multiprocessor [VBS+95].

In [UKG+95], the Toronto researchers identify several previous proposals for improving scalability through approaches similar to multicellular structuring: [AhG91, CGB91, FeR90, ZhB91]. Hurricane was the first complete operating system implementation based on these ideas, but neither it nor the previous proposals investigate the reliability benefits of the architecture as Hive does.

Many of the mechanisms in Hive parallel similar mechanisms in Hurricane and Tornado. Hurricane includes an RPC subsystem for inter-kernel communication, local representative page descriptors with the same function as Hive's extended pfdats, home clusters for files with the same functions as the Hive data home, and so on. There are two key differences. First, Hurricane and Tornado are built from the ground up as multicellular kernels rather than being modified from an existing SMP kernel, so their implementations are both more modular and more flexible. For example, Hurricane includes a single system-wide locking protocol that makes the interaction between kernels more regular [UKG+94]. Second, since reliability is not an issue, the boundaries

between kernels are much more flexible. For example, different resources such as memory and processors can be managed at different cluster sizes simultaneously.

Current work on Tornado focuses on efficient support for applications whose data set size is larger than main memory, and on providing predictable physical resource allocation to applications so compilers and applications can be optimized with predictable performance results. These techniques and the overall set of scalability techniques developed for Hurricane and Tornado are clearly complementary to the mechanisms developed to improve reliability in Hive.

Reliability: The first large-scale general-purpose shared-memory multiprocessor was the C.mmp, developed at Carnegie-Mellon University in the 1970s. Previous general-purpose multiprocessors such as the Burroughs D825, IBM 360/67, and the Honeywell 645 that ran Multics were limited to four processors. C.mmp scaled to 16 processors, which was large from the perspective of reliability issues given the low-integration technology from which it was constructed. The developers found it necessary to add novel reliability-oriented features to the Hydra operating system to achieve adequate MTBF [WLH81].

Hydra includes mechanisms to tolerate both hardware and software faults. At the hardware level, it has a watchdog mechanism to detect fail-stop processor faults and a resource exploration phase at reboot time that can avoid bad memory pages. It tolerates certain non-fail-stop hardware faults, including lost interrupts and all-zeros or all-ones corruption of memory words. For software faults, the primary goal is to ensure that the system will reboot quickly and cleanly. This is difficult because the virtual memory system is integrated with the file system into a two-level object store, so integrity of the database must be recovered in order to reboot. Hydra improves the reliability of the database using techniques such as storing an object identifier along with each pointer to provide a redundant check, deliberately setting reference counts too high during sensitive operations, and delaying the deallocation of objects whose reference counters had reached zero. These techniques could be applied to the internal data structures of a modern SMP kernel to improve its reliability with respect to software faults.

At the same time that the C.mmp project was active at Carnegie-Mellon University, BBN built the Pluribus multiprocessor as a network switch for the ARPANET [KEM+78]. Although not a general-purpose multiprocessor, Pluribus is notable for its focus on fault containment implemented in software as a reliability mechanism. The operating system partitions the machine, run a separate instance of the operating system in each partition, and uses a consensus mechanism to integrate or exclude partitions. Heuristic self-checks run in each partition detect software and hardware errors and shut down the partition when activated. Pluribus shows the upper bound of

reliability achievable with fault containment techniques if the operating system is written from scratch for maximum fault containment rather than modified from an existing SMP kernel.

Finally, the transaction processing multiprocessors built by Tandem set the standard for commercial high-availability systems, combining redundant fail-fast hardware with fault-tolerant software. Tandem systems use two operating systems. The Guardian system [SiS92] runs a distributed system internal to a non-shared-memory multiprocessor. The separate kernels use a heartbeat protocol and a distributed consensus mechanism to agree on which processors are alive. Reliability with respect to software faults is provided by a high number of assertions that shut down a kernel when it detects any internal inconsistency.

The Tandem Nonstop-UX kernel [Jew91, TIY+95] runs as a uniprocessor UNIX kernel on top of triply-redundant hardware. The redundant hardware masks most hardware failures, while the operating system detects software errors using assertions and recovers using forward recovery routines. The system provides a fault containment rather than a fault tolerance abstraction to applications: a user process will be killed if the kernel state that supports it is corrupted by a software error. System reliability is improved using write protection implemented by the memory system of the Tandem Integrity S2, coincidentally called a firewall. The Integrity S2 firewall prevents malfunctioning device controllers from modifying incorrect memory pages but does not support operating system software error containment.

Scalability: In addition to the University of Toronto research and other proposals for scalable multicellular kernels already described, there has been significant work on parallelizing UNIX for performance on small-scale multiprocessors and implementing the UNIX semantics on large-scale non-shared-memory multicomputers.

Mechanisms used to achieve efficient performance from UNIX SVR4 on small-scale parallel systems are described in [SPY+93, Pea92, CBB+91, CHS91]. Mechanisms in the Sun Microsystems versions of UNIX SVR4 are described in [EKB+92]. Work on the Silicon Graphics version is described in [BaB95]. These efforts are all focused on achieving sufficient parallelization to avoid synchronization bottlenecks on four- or eight-processor systems.

The largest-scale systems constructed to date that provide standard SMP semantics are non-shared-memory multicomputers. The Mach SMP kernel has been combined with distributed system techniques from Locus to create OSF/1 AD TNC [ZRB+93], which scales to hundreds of nodes. This operating system uses pure message passing for interkernel communication and a token-based scheme for transferring ownership of kernel objects. Although it successfully manages the hardware resources of hundreds of nodes, it is designed to support scientific

applications that make few kernel requests other than I/O and so is unlikely to provide scalable performance to general-purpose applications.

10.3 Distributed systems

The multicellular architecture achieves reliability and scalability by applying distributed system techniques inside a multiprocessor.

Single-system image: Locus [PoW85] pioneered single-system image support for distributed systems. In addition to a shared filesystem and distributed process management, it demonstrates techniques for coping with network partitions. The Locus work was initially commercialized in AIX/TCF from IBM [WaP89] and later as a portable SSI layer that is one component of OSF/1 AD TNC [ZRB+93]. The later versions include process migration and distributed shared memory.

Sprite [OCD+88] implements process migration and a high-performance SSI distributed file system. Work on Sprite includes mechanisms for reducing the recovery time of distributed file system state [Bak94] which can be applied to reducing the pause times required to recover consistency of distributed state in a multicellular kernel.

Other single-system image distributed systems include MOSIX [BGW93], Chorus/MIX [AGH+89], and Solaris MC [KBM+96]. Solaris MC demonstrates techniques for efficiently implementing a single network identity for the separate kernels with respect to internal processes and external clients.

Resource sharing: The classic resource sharing problem for distributed systems is process migration, implemented in the above single-system image systems and others. Condor [LiS92] provides checkpointing and process migration on top of UNIX without kernel changes, while [MZD+93] describes a process migration mechanism on top of Mach that is transparent to applications. [Nut94] surveys systems that provide migration.

The high amount and cost of memory in desktop workstations has also stimulated development of systems that use the memory of other systems on a local-area network as paging devices. Apollo DOMAIN was an early system to implement this functionality [LLD+83]. More recent systems, including cooperative caching [DWA+94] and GMS [FMP+95], have implemented policies to globally optimize the file pages cached across the machines of the system. These techniques are less relevant to multicellular kernels than process migration mechanisms because the ability to directly access remote pages through shared memory fundamentally changes the performance tradeoffs of page placement decisions.

Error diagnosis: Researchers working on distributed system failure recovery have encountered many of the same problems that multicellular kernels must solve in order to recover correctly after a cell failure. Two key areas are determining the liveset and achieving consensus among all correct cells.

The challenge of determining the liveset is the classic *system-level diagnosis* problem: given a set of test results from cells testing each other, compute the set of faulty cells. Theoretical work on this problem is surveyed in [Dah87] and [LeS94]. Distributed implementations, which do not require a separate fault-free control processor to execute the diagnosis algorithm, include [Dah86] and [BuB93]. The algorithm used in Hive is simpler and less efficient than some known algorithms, so it should be possible to reduce the null recovery latency and the chance of incorrect diagnosis by using a more sophisticated approach.

The diagnosis algorithm used in Hive relies on reliable all-to-all broadcast to achieve consensus on the new live set. Even without changing the overall algorithm, the simple flood algorithm used for reliable broadcast could be improved to terminate early if no cells have failed. Mechanisms for doing this are described in [CDV+94].

The Hive diagnosis algorithm makes strong assumptions about the observability of errors. In particular, it does not behave predictably with respect to software faults that cause intermittent errors. Algorithms are known that can tolerate a wider range of faults. [LeS94] surveys system-level diagnosis algorithms in which the test results are only probabilistically correct, while [ChT96] considers the general problem of consensus and voting when only weak failure detection is possible.

10.4 Error model and reliability prediction

Field failure data: Studies on the errors that affect systems in the field help the design of Hive and similar systems in two ways. First, they validate the assumptions made in Hive's error model about which errors occur and their relative frequencies. Second, they support the design of more sophisticated fault and error injection studies, which should make it possible to predict the reliability of the completed system more accurately.

Systematic efforts to understand the types of errors that affect operating systems started with studies of IBM mainframes, since these were the dominant computing platform for commercial use. An ongoing study of systems at Stanford produced [Bea79] [VeI84, MoA87]. The later studies confirm the earlier ones and make three key observations: (1) storage management and addressing errors are the major immediate effects of software errors, accounting for almost half the entries in system error logs; (2) the recovery routines in the operating system successfully handle errors

about half the time, with the exception of timing and I/O errors that are rarely recoverable; and (3) the rate of software errors is closely linked to the intensity of the interactive workload of the system.

[SuC91] is a more recent study on IBM mainframes that uses error logs to study the probability of wild writes. This study finds that 15–25% of significant operating system software faults cause wild writes, and that only 40% of the wild writes that occur in the field cause immediate addressing errors. Thus around 10% of software faults have the potential to cause data corruption. However, the study also observes that about 50% of the wild writes damage memory that is close to the intended target of the write, which is not the type of wild write that causes data integrity problems for Hive.

Tandem operating systems have also been studied extensively because of their intended use in environments with high availability requirements. [LeI92] and [LeI93] examine the Guardian90 operating system, while [TIY+95] examines the Nonstop-UX operating system. [LeI93] and [TIY+95] are particularly interesting because they provide detailed information on the types and locations of programming mistakes in the operating system code. [LeI93] notes that timing errors are the most significant single cause of system failures in the field, while missing operations (pointer or variable initialization, data update or message send) are the most common type of low-level programming mistake. [TIY+95] observes that pointer manipulation errors and missing checks for illegal data values are the most common low-level mistakes in Nonstop-UX.

The most relevant field failure data for Hive's error model is provided by [ChC96]. This study attempts to develop a statistically valid error model, using field data from "a large IBM operating system." Given 408 operating system software defects that caused field failures over a two-year period, the study finds that 20% of defects cause the operating system to use an incorrect address, while an additional 26% corrupt memory in a non-deterministic way and hence could potentially cause a wild write. The study breaks these and other coarse categories down into a joint distribution of specific fault and error types.

Finally, several studies examine the frequency of types of errors that are outside Hive's error model. [HJL93] considers the possibility of non-fail-fast errors in microprocessors, concluding that a data integrity violation occurs once per month in a population of 10,000 processors built with 1990-era technology. This suggests that more sophisticated data checking mechanisms may be required for very large multiprocessors, although future microprocessors are likely to incorporate more internal redundancy and integrity checks which reduce the error rate. [MaF90] examines cases in which an operating system begins "babbling," that is, flooding the network and thus degrading the performance of the system. Such a failure would bypass the recovery mechanisms

in Hive. Once cases caused by installation of new operating system software, installation of hardware, and network testing are eliminated, the study found only two cases of babbling in seven months of observation of a network containing 2000 machines. This rate would only become significant for very large multiprocessors that are partitioned into minimally-sized cells.

Fault injection: In addition to field failure data, researchers have also used fault injection studies to examine the detailed effects of errors. Work in the RIO project reported in [CNC+96] uses instruction corruption to study the probability of data integrity violations due to wild writes in Digital UNIX. As described in Section 6.5.4, these studies match the results of similar experiments on Hive quite closely. [KIT93] reports the results of similar experiments using SunOS 4.1.2.

The Esprit project on Fault Tolerant Massively Parallel Systems has supported several error injection studies that considered questions relevant to improving the reliability of operating systems. [RMS96] uses pin-level error injection, while [SCM+96] uses internal debugging features of the PowerPC 601 to emulate errors in architecturally hidden processor functional units. These studies demonstrate a significant chance that naive applications will produce incorrect results when errors are injected into one of the system's processors. However, the studies use scientific workloads that can be presumed to spend most of their time executing application code rather than operating system code, so this does not necessarily indicate that application data integrity violations are likely to follow from errors that affect the operating system.

Reliability prediction: Many of the papers discussed above, especially those written or advised by R. K. Iyer, provide mathematical models that assist in predicting reliability from failure data. Notable examples include [LeI92, LeI93, TaI92b, TaI93]. [ChC96] considers how to design experiments so that reliability prediction based on fault injection studies will be statistically valid, while [EIP+91] uses data from system test of a UNIX-based AT&T network management system to analyze the validity of various mathematical models. Finally, [Ham92] provides a non-mathematical introduction to the requirements for effectively predicting software reliability, while [Ham96] provides a more detailed methodology.

Chapter 11
Conclusions

The goal of this research has been to improve the reliability and scalability of large multiprocessors used as general-purpose compute servers.

Contributions: This dissertation has made three contributions towards this goal.

- Demonstration that fault containment is possible inside a shared-memory multiprocessor.

The fault injection experiments convincingly demonstrate that the multicellular kernel architecture can provide fault containment in a multiprocessor. The best estimate available from the experiments in this dissertation is that Hive's uncontained failure rate is around 4% when presented with random faults that damage the operating system. This is far better than the 100% failure rate of an SMP operating system, and could be improved significantly through further testing and debugging.

- Specification of a set of hardware features for FLASH, generalizable to other multiprocessors, that is sufficient to support hardware and software fault containment.

Section 9.1 summarizes the feature set that Hive uses and gives pointers to their precise specifications throughout the dissertation. The key features are the memory fault model that supports hardware fault containment and the firewall that supports software fault containment. I do not argue that all of the features added to FLASH to support Hive are necessary for fault containment, but Hive's success at fault containment (at least in simulation) demonstrates that they are sufficient.

- Demonstration that cells can take advantage of shared-memory hardware across cell boundaries at both application and kernel level while preserving fault containment.

The parallel make workload running in the instruction corruption experiments stresses the file system, which uses user-level shared memory, and the remote fork implementation, which uses kernel-level shared memory. Therefore the experiments demonstrate that both types of memory sharing can be used without weakening fault containment.

Performance: Although Hive appears to perform relatively well, this research has not progressed to the point that it demonstrates that the multicellular architecture is competitive with SMP

operating systems. In particular, the small system sizes simulated and the limited functionality of the Hive prototype suggest that many performance problems may remain undetected.

The prototype lacks four main features that must be implemented before the performance results are conclusive: a complete single-system image, Wax, spanning tasks, and an advanced file system. Completing the single-system image is unlikely to cause performance problems except in the networking subsystem. However, the other missing features have greater performance impacts. Wax must function efficiently if it is to keep the system balanced under rapidly-changing dynamic workloads. Spanning tasks will require creative implementation, perhaps using kernel-level shared memory extensively to achieve reasonable performance.

The most significant of the four missing features is the file system. The file system for a multicellular kernel must provide a globally shared namespace, replication of critical directories and files, striping and software RAID, and takeover of dual-ported disks by a backup cell after the primary cell fails. It must do all this while tolerating the loss of any cells in the system. File systems of this nature are just now emerging in the research community [ADN+96] and will require significant further development before they can be widely used.

Implications: If multicellular kernels turn out to have competitive performance at system sizes that are commercially viable, the fault containment that they provide will open much larger markets to multiprocessors than would be available with SMP operating systems. This will enable many more users to benefit from the excellent resource sharing and ease of administration provided by multiprocessors. By increasing the sales volume of these machines, multicellular kernels also have the potential to reduce their cost, which will give benefits to users that already use large multiprocessors.

More fundamentally, this research demonstrates that the traditional assumptions about the inherent unreliability of shared memory systems are incorrect. Hive draws a fault containment boundary inside the shared memory boundary and gains reliability without sacrificing resource sharing. In this regard, Hive is part of a widespread research effort [CNC+96, Gil96, WLA+93] to reevaluate the fundamental ways in which shared memory can be used.

References

[ACC+93] S. Arevalo, J. Carretero, J.L. Castellanos, and F. Barco. “A fault-tolerant server on MACH.” *Nineteenth EUROMICRO Symposium on Microprocessing and Microprogramming*, pp. 793–800 (Barcelona, Spain, September 6–9, 1993). Available as *Microprocessing & Microprogramming*, vol. 38, no. 1, September 1993.

[ADN+96] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, and R.Y. Wang. “Serverless network file systems.” *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 41–79, February 1996.

[AGH+89] F. Armand, M. Gien, F. Herrmann, and M. Rozier. “Distributing UNIX brings it back to its original virtues.” *USENIX Workshop on Distributed and Multiprocessor Systems*, pp. 153–174 (Fort Lauderdale, FL, October 5–6, 1989). Berkeley, CA: USENIX Association, 1989.

[AhG91] I. Ahmad and A. Ghafoor. “Semi-distributed load balancing for massively parallel multicomputer systems.” *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 987–1004, October 1991.

[BaB95] J.M. Barton and N. Bitar. “A scalable multi-discipline, multiple-processor scheduling framework for IRIX.” *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 45–69 (Santa Barbara, CA, April 25, 1995). Berlin, Germany: Springer-Verlag, 1995.

[Bak94] M. Baker. *Fast Crash Recovery in Distributed File Systems.* Doctoral dissertation, technical report CSD-94-787, Computer Science Division, University of California, Berkeley, January 1994.

[Bar81] J.F. Bartlett. “A NonStop kernel.” *Eighth ACM Symposium on Operating Systems Principles*, pp. 22–29 (Pacific Grove, CA, December 1981). Available as *Operating Systems Review*, vol. 15, no. 5, December 1981.

[Bea79] M.D. Beaudry. “A statistical analysis of failures in the SLAC computing center.” *Digest of papers, COMPCON, Spring ’79*, pp. 49–52 (San Francisco, February 26–March 1, 1979). New York: IEEE, 1979.

[BGW93] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. New York: Springer-Verlag, 1993.

[BSS+96] D.L. Black, R.D. Smith, S.J. Sears, and R.W. Dean. “FLIPC: a low latency messaging system for distributed real time environments.” *USENIX 1996 Annual Technical Conference*, pp. 229–238 (San Diego, CA, January 22–26, 1996). Berkeley, CA: USENIX Association, 1996.

[BuB93] R.W. Buskens and R.P. Bianchini. “Distributed on-line diagnosis in the presence of arbitrary faults.” *Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 470–479 (Toulouse, France, June 22–24, 1993). Los Alamitos, CA: IEEE Computer Society Press, 1993.

[CaS82] X. Castillo and D.P. Siewiorek. “A workload dependent software reliability prediction model.” *Twelfth International Symposium on Fault-Tolerant Computing*, pp. 279–286 (Santa Monica, CA, June 1982). Long Beach, CA: IEEE Computer Society, 1982.

[CBB+91] M. Campbell, R. Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith, and R. Wescott. “The parallelization of UNIX System V Release 4.0.” *Winter 1991 USENIX Conference*, pp. 307–323 (Dallas, TX, January 21–25, 1991). Berkeley, CA: USENIX Association, 1991.

[CBR95] R. Chillarege, S. Biyani, and J. Rosenthal. “Measurement of failure rate in widely distributed software.” *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 424–433 (Pasadena, CA, June 27–30, 1995). Los Alamitos, CA: IEEE Computer Society Press, 1995.

[CDV+94] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. “Scheduling and page migration for multiprocessor compute servers.” *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–24 (San Jose, CA, October 4–7, 1994). Available as *SIGPLAN Notices*, vol. 29, no. 11, November 1994.

[CGB91] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. “Paradigm: a highly scalable shared-memory multicomputer architecture.” *Computer*, vol. 24, no. 2, pp. 33–46, February 1991.

[ChB94] R. Chillarege and S. Biyani. “Identifying risk using ODC based growth models.” *Fifth International Symposium on Software Reliability Engineering*, pp. 282–288 (Monterey, CA, November 6–9, 1994). Los Alamitos, CA: IEEE Computer Society Press, 1994.

[ChC96] J. Christmansson and R. Chillarege. “Generation of an error set that emulates software faults based on field data.” *Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pp. 304–313 (Sendai, Japan, June 25–27, 1996). Los Alamitos, CA: IEEE Computer Society Press, 1996.

[CHR+95] J. Chapin, S.A. Herrod, M. Rosenblum, and A. Gupta. “Memory system performance of UNIX on CC-NUMA multiprocessors.” *1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 1–13 (Ottawa, Ontario, Canada, May 15–19, 1995). Available as *Performance Evaluation Review*, vol. 23, no. 1, May 1995.

[CHS91] M. Campbell, R. Holt, and J. Slice. “Lock granularity tuning mechanisms in SVR4/MP.” *Symposium on Experiences with Distributed and Multiprocessor Systems, SEDMS II*, pp. 221–228 (Atlanta, GA, March 21–22, 1991). Berkeley, CA: USENIX Association, 1991.

[ChT96] T.D. Chandra and S. Toueg. “Unreliable failure detectors for reliable distributed systems.” *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, March 1996.

[CIR+93] R.H. Campbell, N. Islam, D. Raila, and P. Madany. “Designing and implementing Choices: an object-oriented system in C++.” *Communications of the ACM*, vol. 36, no. 9, pp. 117–126, September 1993.

[CNC+96] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. “The Rio file cache: surviving operating system crashes.” *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 74–83 (Cambridge, MA, October 1–5, 1996). Available as *Operating Systems Review*, vol. 30, no. 5, October 1996.

[CVJ92] R. Cramp, M.A. Vouk, and W. Jones. “On operational availability of a large software-based telecommunications system.” *Third International Symposium on Software Reliability Engineering*, pp. 358–366 (Research Triangle Park, NC, October 7–10, 1992). Los Alamitos, CA: IEEE Computer Society Press, 1992.

[Dah86] A.T. Dahbura. “An efficient algorithm for identifying the most likely fault set in a probabilistically diagnosable system.” *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 354–356, April 1986.

[Dah87] A.T. Dahbura. “System-level diagnosis: a perspective for the third decade.” *Princeton Workshop on Algorithm, Architecture, and Technology Issues for Models of Concurrent Computation*, pp. 411–434 (Princeton, NJ, September 30–October 1, 1987). Available as S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, eds., *Concurrent Computations: Algorithms, Architecture, and Technology.* New York: Plenum Press, 1987.

[Dat96] Data General Corporation. “NUMA: delivering the next level of commodity SMP performance.” Document 4855, *VIEWPOINT* no. 5, 1996.

[DWA+94] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. “Cooperative caching: using remote client memory to improve file system performance.” *First USENIX Symposium on Operating Systems Design and Implementation*, pp. 267–280 (Monterey, CA, November 14–17, 1994). Berkeley, CA: USENIX Association, 1994.

[EIP+91] W.K. Ehrlich, A. Iannino, B.S. Prasanna, J.P. Stampfel, and J.R. Wu. “How faults cause software failures: implications for software reliability engineering.” *Second International Symposium on Software Reliability Engineering*, pp. 233–241 (Austin, TX, May 17–18, 1991). Los Alamitos, CA: IEEE Computer Society Press, 1991.

[EKB+92] J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. “Beyond multiprocessing: multithreading the SunOS kernel.” *Summer 1992 USENIX Conference*, pp. 11–18 (San Antonio, TX, June 8–12, 1992). Berkeley, CA: USENIX Association, 1992.

[FeR90] D.G. Feitelson and L. Rudolph. “Distributed hierarchical control for parallel processing.” *Computer*, vol. 23, no. 5, pp. 65–77, May 1990.

[FMP+95] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy, and C.A. Thekkath. “Implementing global memory management in a workstation cluster.” *Fifteenth ACM Symposium on Operating Systems Principles*, pp. 201–212 (Copper Mountain Resort, CO, December 3–6, 1995). Available as *Operating Systems Review*, vol. 29, no. 5, December 1995.

[Gal96] M. Galles. “Scalable pipelined interconnect for distributed endpoint routing: the SGI SPIDER chip.” *Hot Interconnects Symposium IV* (Stanford, CA, August 1996).

[Gil96] R.B. Gillett. “Memory channel network for PCI.” *IEEE Micro*, vol. 16, no. 1, pp. 12–18, February 1996.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. “Memory consistency and event ordering in scalable shared-memory multiprocessors.” *Seventeenth Annual International Symposium on Computer Architecture*, pp. 15–26 (Seattle, WA, May 28–31, 1990). Los Alamitos, CA: IEEE Computer Society Press, 1990.

[Gra90] J. Gray. “A census of Tandem system availability between 1985 and 1990.” *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409–418, October 1990.

[GrC96] M. Greenwald and D. Cheriton. “The synergy between non-blocking synchronization and operating system structure.” *Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 123–136 (Seattle, WA, October 28–31, 1996). Berkeley, CA: USENIX Association, 1996.

[Ham92] D. Hamlet. “Are we testing for true reliability?” *IEEE Software*, vol. 9, no. 4, pp. 21–27, July 1992.

[Ham96] D. Hamlet. “Predicting dependability by testing.” *1996 International Symposium on Software Testing and Analysis*, pp. 84–91 (San Diego, CA, January 8–10, 1996). Available as *SIGSOFT Software Engineering Notes*, vol. 21, no. 3, May 1996.

[HJK93] Y. Huang, P. Jalote, and C. Kintala. “Two techniques for transient software error recovery.” *Hardware and Software Architectures for Fault Tolerance*, pp. 159–170 (Le Mont Saint Michel, France, June 1993). Berlin, Germany: Springer-Verlag, 1994.

[HJL93] R. Horst, D. Jewett, and D. Lenoski. “The risk of data corruption in microprocessor-based systems.” *Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 576–585 (Toulouse, France, June 22–24, 1993). Los Alamitos, CA: IEEE Computer Society Press, 1993.

[HKO+94] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The performance impact of flexibility in the Stanford FLASH multiprocessor.” *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 274–284 (San Jose, CA, October 4–7, 1994). Available as *SIGPLAN Notices*, vol. 29, no. 11, November 1994.

[HP95] HP-Convex Technology Center. “Convex Examplar scalable computing systems feature message passing and shared memory for programming flexibility.” Press release, November 22, 1995.

[Iye95] R. Iyer. “Experimental evaluation.” *Twenty-Fifth International Symposium on Fault-Tolerant Computing, Special Issue*, pp. 115–132 (Pasadena, CA, June 27–30, 1995). Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Jew91] D. Jewett. “Integrity S2: a fault-tolerant Unix platform.” *Twenty-First International Symposium on Fault-Tolerant Computing*, pp. 512–519 (Montreal, Quebec, Canada, June 25–27, 1991). Los Alamitos, CA: IEEE Computer Society Press, 1991.

[Joh89] B. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Reading, MA: Addison-Wesley, 1989.

[KBM+96] Y.A. Khalidi, J.M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani. “Solaris MC: a multicomputer OS.” *USENIX 1996 Annual Technical Conference*, pp. 191–203 (San Diego, CA, January 22–26, 1996). Berkeley, CA: USENIX Association, 1996.

[KEM+78] D. Katsuki, E. Elsam, W. Mann, E. Roberts, J. Robinson, F. Skowronski, and E. Wolf. “Pluribus—an operational fault-tolerant multiprocessor.” *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1146–1159, October 1978.

[KIT93] W.-I. Kao, R.K. Iyer, and D. Tang. “FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults.” *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, November 1993.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The Stanford FLASH multiprocessor.” *Twenty-First Annual International Symposium on Computer Architecture*, pp. 302–313 (Chicago, April 18–21, 1994). Los Alamitos, CA: IEEE Computer Society Press, 1994.

[Kri95] O. Krieger. *HFS: A Flexible File System for Shared-Memory Multiprocessors.* Doctoral dissertation, Department of Electrical and Computer Engineering, University of Toronto, 1995.

[KTR94] D. Kotz, S. Toh, and S. Radhakrishnan. “A detailed simulation of the HP 97560 disk drive.” Technical report PCS-TR94-20, Dartmouth College, 1994.

[LeI92] I. Lee and K. Iyer. “Analysis of software halts in the Tandem GUARDIAN operating system.” *Third International Symposium on Software Reliability Engineering*, pp. 227–236 (Research Triangle Park, NC, October 7–10, 1992). Los Alamitos, CA: IEEE Computer Society Press, 1992.

[LeI93] I. Lee and R.K. Iyer. “Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system.” *Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 20–29 (Toulouse, France, June 22–24, 1993). Los Alamitos, CA: IEEE Computer Society Press, 1993.

[LeS94] S. Lee and K. G. Shin. “Probabilistic diagnosis of multiprocessor systems.” *ACM Computing Surveys*, vol. 26, no. 1, pp. 121–139, March 1994.

[LiR93] J. Lipkis and M. Rozier. “Fault tolerance enablers in the CHORUS microkernel.” *Hardware and Software Architectures for Fault Tolerance*, pp. 182–190 (Le Mont Saint Michel, France, June 1993). Berlin, Germany: Springer-Verlag, 1994.

[LiS92] M. Litzkow and M. Solomon. “Supporting checkpointing and process migration outside the UNIX kernel.” *Winter 1992 USENIX Conference*, pp. 283–290 (San Francisco, January 20–24, 1992). Berkeley, CA: USENIX Association, 1992.

[LLD+83] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. “The architecture of an integrated local network.” *IEEE Journal on Selected Areas in Communications*, vol. 1, no. 5, pp. 842–857, November 1983.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. “The Stanford Dash multiprocessor.” *Computer*, vol. 25, no. 3, pp. 63–79, March 1992.

[LNP94] K. Li, J.F. Naughton, and J.S. Plank. “Low-latency, concurrent checkpointing for parallel programs.” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, August 1994.

[LoC96] T. Lovett and R. Clapp. “STiNG: a CC-NUMA computer system for the commercial marketplace.” *Twenty-Third Annual International Symposium on Computer Architecture*, pp. 308–317 (Philadelphia, PA, May 22–24, 1996). Available as *Computer Architecture News*, vol. 24, no. 2, May 1996.

[LSP82] L. Lamport, R. Shostak, and M. Pease. “The Byzantine Generals problem.” *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[Lyn96] N. Lynch. *Distributed Algorithms*. San Francisco: Morgan Kaufmann, 1996.

[MaF90] R.A. Maxion and F.E. Feather. “A case study of Ethernet anomalies in a distributed computing environment.” *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 433–443, October 1990.

[MGH+94] J.G. Mitchell, J.J. Gibbons, G. Hamilton, P.B. Kessler, Y.A. Khalidi, P. Kougiouris, P.W. Madany, M.N. Nelson, M.L. Powell, and S.R. Radia. “An overview of the Spring system.” *COMPCON, Spring ’94*, pp. 122–131 (San Francisco, February 28–March 4, 1994). Los Alamitos, CA: IEEE Computer Society Press, 1994.

[MoA87] S. Mourad and D. Andrews. “On the reliability of the IBM MVS/XA operating system.” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 10, pp. 1135–1139, October 1987.

[MZD+93] D.S. Milojicic, W. Zint, A. Dangel, and P. Giese. “Task migration on the top of the Mach microkernel.” *USENIX Mach III Symposium*, pp. 273–289 (Santa Fe, NM, April 19–21, 1993). Berkley, CA: USENIX Association, 1993.

[Nut94] M. Nuttall. “A brief survey of systems providing process or object migration facilities.” *Operating Systems Review*, vol. 28, no. 4, pp. 64–80, October 1994.

[OCD+88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. “The Sprite network operating system.” *Computer*, vol. 21, no. 2, pp. 23–36, February 1988.

[PBK+95] J.S. Plank, M. Beck, G. Kingsley, and K. Li. “Libckpt: transparent checkpointing under Unix.” *1995 USENIX Technical Conference*, pp. 213–224 (New Orleans, LA, January 16–20, 1995). Berkeley, CA: USENIX Association, 1995.

[Pea92] J.K. Peacock. “File system multithreading in System V Release 4 MP.” *Summer 1992 USENIX Conference*, pp. 19–29 (San Antonio, TX, June 8–12, 1992). Berkeley, CA: USENIX Association, 1992.

[PGK+95] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. “(De-)clustering objects for multiprocessor system software.” *Fourth International Workshop on Object-Orientation in Operating Systems*, pp. 72–81 (Lund, Sweden, August 14–15, 1995). Los Alamitos, CA: IEEE Computer Society Press, 1995.

[PoW85] G. Popek and B. Walker, eds. *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press, 1985.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. “CHORUS distributed operating systems.” *Computing Systems*, vol. 1, no. 4, pp. 305–370, Fall 1988.

[RBD+] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. “Using the SimOS machine simulator to study complex computer systems.” *ACM TOMACS*, in press.

[RHW+95] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. “Complete computer system simulation: the SimOS approach.” *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 3, no. 4, pp. 34–43, Winter 1995.

[RJO+89] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. “Mach: a system software kernel.” *COMPCON, Spring ’89*, pp. 176–178 (San Francisco, February 27–March 3, 1989). Washington, DC: IEEE Computer Society Press, 1989.

[RMS96] M.Z. Rela, H. Madeira, and J.G. Silva. “Experimental evaluation of the fail-silent behaviour in programs with consistency checks.” *Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pp. 394–403 (Sendai, Japan, June 25–27, 1996). Los Alamitos, CA: IEEE Computer Society Press, 1996.

[RSS93] M. Russinovich, Z. Segall, and D.P. Siewiorek. “Application transparent fault management in fault tolerant Mach.” *Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 10–19 (Toulouse, France, June 22–24, 1993). Los Alamitos, CA: IEEE Computer Society Press, 1993.

[SaH94] S. Sankar and R. Hayes. “ADL—an interface definition language for specifying and testing software.” *ACM Workshop on Interface Definition Languages*, pp. 13–21 (Portland, OR, January 20, 1994). Available as *SIGPLAN Notices*, vol. 29, no. 8, August 1994.

[SCM+96] J.G. Silva, J. Carreira, H. Madeira, D. Costa, and P. Moreira. “Experimental assessment of parallel systems.” *Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pp. 415–424 (Sendai, Japan, June 25–27, 1996). Los Alamitos, CA: IEEE Computer Society Press, 1996.

[SDH+96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. “Scalability in the XFS file system.” *USENIX 1996 Annual Technical Conference*, pp. 1–14 (San Diego, CA, January 22–26, 1996). Berkeley, CA: USENIX Association, 1996.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. “Design and implementation of the Sun network filesystem.” *Summer 1985 USENIX Conference*, pp. 119–130 (Portland, OR, June 1985). Berkeley, CA: USENIX Association, 1985.

[Sil96] Silicon Graphics Inc. “Origin family of technical and enterprise servers.” Press release, October 7, 1996.

[SiS92] D. Siewiorek and R. Swarz. *Reliable computer systems: design and evaluation* (second edition). Burlington, MA: Digital Press, 1992.

[SPY+93] S. Saxena, J.K. Peacock, F. Yang, V. Verma, and M. Krishnan. “Pitfalls in multithreading SVR4 STREAMS and other weightless processes.” *Winter 1993 USENIX Conference*, pp. 85–96 (San Diego, CA, January 25–29, 1993). Berkley, CA: USENIX Association, 1993.

[SuC91] M. Sullivan and R. Chillarege. “Software defects and their impact on system availability—a study of field failures in operating systems.” *Twenty-First International Symposium on Fault-Tolerant Computing*, pp. 2–9 (Montreal, Quebec, Canada, June 25–27, 1991). Los Alamitos, CA: IEEE Computer Society Press, 1991.

[TaI92a] D. Tang and R.K. Iyer. “Analysis of the VAX/VMS error logs in multicomputer environments—a case study of software dependability.” *Third International Symposium on Software Reliability Engineering*, pp. 216–226 (Research Triangle Park, NC, October 7–10, 1992). Los Alamitos, CA: IEEE Computer Society Press, 1992.

[TaI92b] D. Tang and R.K. Iyer. “Analysis and modeling of correlated failures in multicomputer systems.” *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 567–577, May 1992.

[TaI93] D. Tang and R.K. Iyer. “Dependability measurement and modeling of a multicomputer system.” *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 62–75, January 1993.

[TBG+] D. Teodosiu, J. Baxter, K. Govil, J. Chapin, M. Rosenblum, and M. Horowitz. “Hardware fault containment in scalable shared-memory multiprocessors.” In press.

[TeT87] A. Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* Doctoral dissertation, technical report CMU-CS-88-106, Department of Computer Science, Carnegie-Mellon University, 1987.

[TGH92] J. Torrellas, A. Gupta, and J. Hennessy. “Characterizing the caching and synchronization performance of a multiprocessor operating system.” *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 162–174 (Boston, October 12–15, 1992). Available as *SIGPLAN Notices*, vol. 27, no. 9, September 1992.

[TIY+95] A. Thakur, R.K. Iyer, L. Young, and I. Lee. “Analysis of failures in the Tandem NonStop-UX Operating System.” *Sixth International Symposium on Software Reliability Engineering*, pp. 40–50 (Toulouse, France, October 24–27, 1995). Los Alamitos, CA: IEEE Computer Society Press, 1995.

[UKG+94] R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. “Experiences with locking in a NUMA multiprocessor operating system kernel.” *First USENIX Symposium on Operating Systems Design and Implementation*, pp. 139–152 (Monterey, CA, November 14–17, 1994). Berkeley, CA: USENIX Association, 1994.

[UKG+95] R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. “Hierarchical clustering: A structure for scalable multiprocessor operating system design.” *Journal of Supercomputing*, vol. 9, no. 1, pp. 105–134, 1995.

[VBS+95] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. “The NUMAchine multiprocessor.” Unpublished white paper, Department of Electrical and Computer Engineering, University of Toronto, June 28, 1995.

[VDG+96] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. “Operating system support for improving data locality on CC-NUMA compute servers.” *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 279–289 (Cambridge, MA, October 1–5, 1996). Available as *Operating Systems Review*, vol. 30, no. 5, October 1996.

[vEB+95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. “U-Net: a user-level network interface for parallel and distributed computing.” *Fifteenth ACM Symposium on Operating Systems Principles*, pp. 40–53 (Copper Mountain Resort, CO, December 3–6, 1995). Available as *Operating Systems Review*, vol. 29, no. 5, December 1995.

[vEC+92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. “Active messages: a mechanism for integrated communication and computation.” *Nineteenth Annual International Symposium on Computer Architecture*, pp. 256–266 (Gold Coast, Queensland, Australia, May 19–21, 1992). Available as *Computer Architecture News*, vol. 20, no. 2, May 1992.

[VeI84] P. Velardi and R.K. Iyer. “A study of software failures and recovery in the MVS operating system.” *IEEE Transactions on Computers*, vol. C-33, no. 7, July 1984.

[VSL+91] Z.G. Vranesic, M. Stumm, D.M. Lewis, and R. White. “Hector: a hierarchically structured shared-memory multiprocessor.” *Computer*, vol. 24, no. 1, pp. 72–79, January 1991.

[WaP89] B. Walker and J. Popek. “Distributed UNIX transparency: goals, benefits, and the TCF example.” *Winter 1989 Uniforum conference* (January 1989). Santa Clara, CA: Uniforum, 1989.

[WiR96] E. Witchel and M. Rosenblum. “Embra: fast and flexible machine simulation.” *1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 68–79 (Philadelphia, PA, May 23–26, 1996). Available as *Performance Evaluation Review*, vol. 24, no. 1, May 1996.

[WLA+93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. “Efficient software-based fault isolation.” *Fourteenth ACM Symposium on Operating Systems Principles*, pp. 203–216 (Ashville, NC, December 5–8, 1993). Available as *Operating Systems Review*, vol. 27, no. 5, December 1993.

[WLH81] W. Wulf, R. Levin, and S. Harbison. *HYDRA/C.mmp, an experimental computer system*. New York: McGraw-Hill, 1981.

[WOT+95] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. “The SPLASH-2 programs: characterization and methodological considerations.” *Twenty-Second Annual International Symposium on Computer Architecture*, pp. 24–36 (Santa Margherita Ligure, Italy, June 22–24, 1995). New York: ACM, 1995.

[ZhB91] S. Zhou and T. Brecht. “Processor pool-based scheduling for large-scale NUMA multiprocessors.” *1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 133–142 (San Diego, CA, May 21–24, 1991). Available as *Performance Evaluation Review*, vol. 19, no. 1, May 1991.

[ZRB+93] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. Lo Verso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. “An OSF/1 Unix for massively parallel multicomputers.” *Winter 1993 USENIX Conference*, pp. 449–468 (San Diego, CA, January 25–29, 1993). Berkley, CA: USENIX Association, 1993.

Index