Hive: Fault Containment for Shared-Memory Multiprocessors

John Chapin, Mendel Rosenblum, Scott Devine,
Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta

Computer Systems Laboratory
Stanford University, Stanford CA 94305
http://www-flash.stanford.edu

Abstract

Reliability and scalability are major concerns when designing operating systems for large-scale shared-memory multiprocessors. In this paper we describe 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 improves reliability because a hardware or software fault damages only one cell rather than the whole system, and 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.

This paper focuses on Hive's solution to the following key challenges: (1) fault containment, i.e. confining the effects of hardware or software faults to the cell where they occur, and (2) memory sharing among cells, which is required to achieve application performance competitive with other multiprocessor operating systems. Fault containment in a shared-memory multiprocessor requires defending each cell against erroneous writes caused by faults in other cells. Hive prevents such damage by using the FLASH *firewall*, a write permission bit-vector associated with each page of memory, and by discarding potentially corrupt pages when a fault is detected. Memory sharing is provided through a unified file and virtual memory page cache across the cells, and through a unified free page frame pool.

We report early experience with the system, including the results of fault injection and performance experiments using SimOS, an accurate simulator of FLASH. The effects of faults were contained to the cell in which they occurred in all 49 tests where we injected fail-stop hardware faults, and in all 20 tests where we injected kernel data corruption. The Hive prototype executes test workloads on a four-processor four-cell system with between 0% and 11% slowdown as compared to SGI IRIX 5.2 (the version of UNIX on which it is based).

1 Introduction

Shared-memory multiprocessors are becoming an increasingly common server platform because of their excellent performance under dynamic multiprogrammed workloads. However, the symmetric multiprocessor operating systems (SMP OS) commonly used for small-scale machines are difficult to scale to the large shared-memory multiprocessors that can now be built (Stanford DASH [11], MIT Alewife [3], Convex Exemplar [5]).

In this paper we describe Hive, an operating system designed for large-scale shared-memory multiprocessors. Hive is fundamentally different from previous monolithic and microkernel SMP OS implementations: it is structured as an internal distributed system of independent kernels called *cells*. This multicellular kernel architecture has two main advantages:

- *Reliability:* In SMP OS implementations, any significant hardware or software fault causes the entire system to crash. For large-scale machines this can result in an unacceptably low mean time to failure. In Hive, only the cell where the fault occurred crashes, so only the processes using the resources of that cell are affected. This is especially beneficial for compute server workloads where there are multiple independent processes, the predominant situation today. In addition, scheduled hardware maintenance and kernel software upgrades can proceed transparently to applications, one cell at a time.
- *Scalability:* SMP OS implementations are difficult to scale to large machines because all processors directly share all kernel resources. Improving parallelism in a "shared-everything" architecture is an iterative trial-and-error process of identifying and fixing bottlenecks. In contrast, Hive offers a systematic approach to scalability. Few kernel resources are shared by processes running on different cells, so parallelism can be improved by increasing the number of cells.

However, the multicellular architecture of Hive also creates new implementation challenges. These include:

- *Fault containment:* The effects of faults must be confined to the cell in which they occur. This is difficult since a shared-memory multiprocessor allows a faulty cell to issue *wild writes* which can corrupt the memory of other cells.
- *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 shared-memory multiprocessor.
- *Single-system image:* The cells must cooperate to present a standard SMP OS interface to applications and users.

In this paper, we focus on Hive's solution to the fault containment problem and on its solution to a key resource sharing problem, sharing memory across cell boundaries. The solutions rely on hardware as well as software mechanisms: we have designed Hive in conjunction with the Stanford FLASH multiprocessor [10], which has enabled us to add hardware support in a few critical areas.

Hive's fault containment strategy has three main components. Each cell uses *firewall* hardware provided by FLASH to defend most of its memory pages against wild writes. Any pages writable by a failed cell are preemptively discarded when the failure is detected, which prevents any corrupt data from being read subsequently by applications or written to disk. Finally, aggressive failure detection reduces the delay until preemptive discard occurs. Cell failures are detected initially using heuristic checks, then confirmed with a distributed agreement protocol that minimizes the probability of concluding that a functioning cell has failed.

Hive provides two types of memory sharing among cells. First, the file system and the virtual memory system cooperate so processes on multiple cells can use the same memory page for shared data. Second, the page allocation modules on different cells cooperate so a free page belonging to one cell can be loaned to another cell that is under memory pressure. Either type of sharing would cause fault containment problems on current multiprocessors, since a hardware fault in memory or in a processor caching the data could halt some other processor that tries to access that memory. FLASH makes memory sharing safe by providing timeouts and checks on memory accesses.

The current prototype of Hive is based on and remains binary compatible with IRIX 5.2 (a version of UNIX SVR4 from Silicon Graphics, Inc.). Because FLASH is not available yet, we used the SimOS hardware simulator [18] to develop and test Hive. Our early experiments using SimOS demonstrate that:

- Hive can survive the halt of a processor or the failure of a range of memory. In all of 49 experiments where we injected a fail-stop hardware fault, the effects were confined to the cell where the fault occurred.
- Hive can survive kernel software faults. In all of 20 experiments where we randomly corrupted internal operating system data structures, the effects were confined to the cell where the fault occurred.
- Hive can offer reasonable performance while providing fault containment. A four-cell Hive executed three test workloads with between 0% and 11% slowdown as compared to IRIX 5.2 on a four-processor machine.

These results indicate that a multicellular kernel architecture can provide fault containment in a shared-memory multiprocessor. The performance results are also promising, but significant further work is required on resource sharing and the single-system image before we can make definitive conclusions about performance.

We begin this paper by defining fault containment more precisely and describing the fundamental problems that arise when implementing it in multiprocessors. Next we give an overview of the architecture and implementation of Hive. The implementation details follow in three parts: fault containment, memory sharing, and the intercell remote procedure call subsystem. We conclude with an evaluation of the performance and fault containment of the current prototype, a discussion of other applications of the Hive architecture, and a summary of related work.

2 Fault Containment in Shared-Memory Multiprocessors

Fault containment is a general reliability strategy that has been implemented in many distributed systems. It differs from fault tolerance in that partial failures are allowed, which enables the system to avoid the cost of replicating processes and data.

Fault containment is an attractive reliability strategy for multiprocessors used as general-purpose compute servers. The workloads characteristic of this environment frequently contain multiple independent processes, so some processes can continue doing useful work even if others are terminated by a partial system failure.

However, fault containment in a multiprocessor will only have reliability benefits if the operating system manages resources well. Few applications will survive a partial system failure if the operating system allocates resources randomly from all over the machine. Since application reliability is the primary goal, we redefine fault containment to include this resource management requirement:

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

One important consequence of choosing this as the reliability goal is that large applications which use resources from the whole system receive no reliability benefits. For example, some compute server workloads contain parallel applications that run with as many threads as there are processors in the system. However, these large applications have previously used checkpointing to provide their own reliability, so we assume they can continue to do so.

The fault containment strategy can be used in both distributed systems and multiprocessors. However, the problems that arise in implementing fault containment are different in the two environments. In addition to all the problems that arise in distributed systems, the shared-memory hardware of multiprocessors increases vulnerability to both hardware faults and software faults. We describe the problems caused by each of these in turn.

Hardware faults: Consider the architecture of the Stanford FLASH, which is a typical large-scale shared-memory multiprocessor (Figure 2.1). FLASH consists of multiple *nodes*, each with a processor and its caches, a local portion of main memory, and local I/O devices. The nodes communicate through a high-speed low-latency mesh network. Cache coherence is provided by a coherence controller on each node. A machine like this is called a CC-NUMA multiprocessor (cache-coherent with non-uniform memory access time) since accesses to local memory are faster than accesses to the memory of other nodes.

In a CC-NUMA machine, an important unit of failure is the node. A node failure halts a processor and has two direct effects on the memory of the machine: the portion of main memory assigned to that node becomes inaccessible, and any memory line whose only copy was cached on that node is lost. There may also be indirect effects that cause loss of other data.

For the operating system to survive and recover from hardware faults, the hardware must make several guarantees about the behavior of shared memory after a fault. Accesses to unaffected memory ranges must continue to be satisfied with normal cache coherence. Processors that try to access failed memory or retrieve a cache line from a failed node must not be stalled indefinitely. Also, the set of memory lines that could be affected by a fault on a given node must be limited somehow, since designing recovery algorithms requires knowing what data can be trusted to be correct.

These hardware properties collectively make up a *memory fault model*, analogous to the memory consistency model of a multiprocessor which specifies the behavior of reads and writes. The FLASH memory fault model was developed to match the needs of Hive: it provides the above properties, guarantees that the network remains fully connected with high probability (i.e. the operating system need not work around network partitions), and specifies that only the nodes that have been authorized to write a given memory line (via the firewall) could damage that line due to a hardware fault.

Software faults: The presence of shared memory makes each cell vulnerable to *wild writes* resulting from software faults in other cells. Wild writes are not a negligible problem. Studies have shown that software faults are more common than hardware faults in current systems [7]. When a software fault occurs, a wild write can easily follow. One study found that among 3000 severe bugs reported in IBM operating systems over a five-year period, between 15 and 25 percent caused wild writes [20].

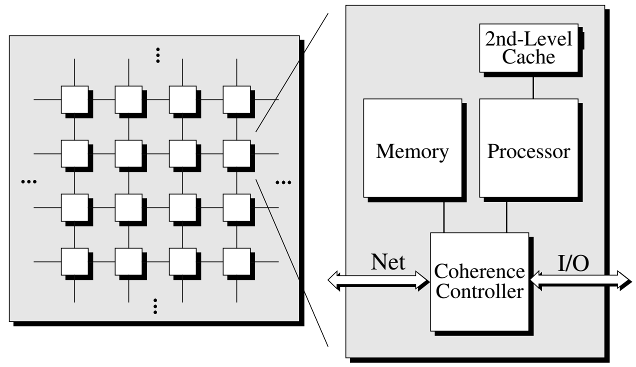

FIGURE 2.1. FLASH architecture.
The machine is structured as a set of nodes in a mesh network. Each node contains a portion of the main memory and a coherence controller which communicates with other nodes to maintain cache coherence. When a hardware fault occurs, the node is a likely unit of failure, so portions of main memory can be lost.

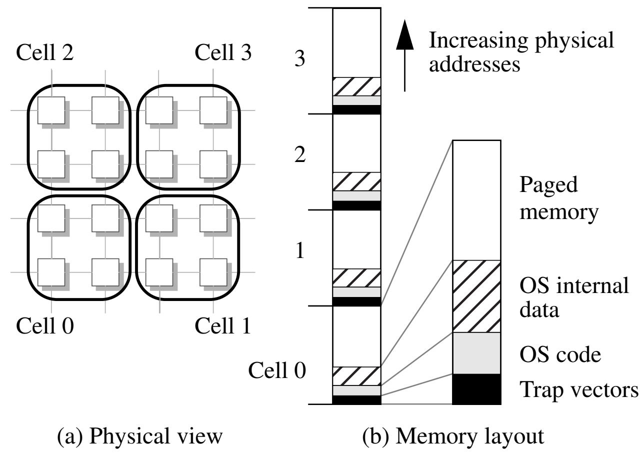

FIGURE 3.1. Partition of a multiprocessor into Hive cells.
Each cell controls a portion of the global physical address space and runs as an independent multiprocessor kernel.

Unfortunately, existing shared-memory multiprocessors do not provide a mechanism to prevent wild writes. The only mechanism that can halt a write request is the virtual address translation hardware present in each processor, which is under the control of the very software whose faults must be protected against.

Therefore an operating system designed to prevent wild writes must either use special-purpose hardware, or rely on a trusted software base that takes control of the existing virtual address translation hardware. For systems which use hardware support, the most natural place to put it is in the coherence controller, which can check permissions attached to each memory block before modifying memory. Systems following a software-only approach could use a microkernel as the trusted base, or could use the lower levels of the operating system's own virtual memory system, on top of which most of the kernel would run in a virtual address space.

The hardware and software-only approaches provide significantly different levels of reliability, at least for an operating system that is partitioned into cells. In the hardware approach, each cell's wild write defense depends only on the hardware and software of that cell. In the software-only approach, each cell's wild write defense depends on the hardware and trusted software layer of all other cells. By reducing the number and complexity of the components that must function correctly to defend each cell against wild writes, the hardware approach provides higher reliability than the software-only approach.

We chose to add firewall hardware, a write permission bit-vector associated with each page of memory, to the FLASH coherence controller. We found that the firewall added little to the cost of FLASH beyond the storage required for the bit vectors. Other large multiprocessors are likely to be similar in this respect, because the hardware required for access permission checking is close to that required for directory-based cache-coherence. The firewall and its performance impact are described in Section 4.2.

3 Hive Architecture

Hive is structured as a set of cells (Figure 3.1). When the system boots, each cell is assigned a range of nodes that it owns throughout execution. Each cell manages the processors, memory, and I/O devices on those nodes as if it were an independent operating system. The cells cooperate to present the required single-system image to user-level processes.

On top of this structure, the architectural features of Hive fall into two broad categories: those related to implementing fault containment, and those related to providing resource sharing despite the fault containment boundaries between cells. After describing both parts of the architecture, we will briefly summarize the implementation status of the Hive prototype.

3.1 Fault containment architecture

Fault containment at the hardware level is a hardware design problem, with requirements specified by the memory fault model that Hive relies on. At the operating system level, there are three channels by which a fault in one cell can damage another cell: by sending a bad message, providing bad data or errors to remote reads, or by causing erroneous remote writes. A cell failure can also deny access to some important resource (such as a common shared library), but that is a different problem which is a subject for further work. We discuss each of the three operating system fault containment problems in turn.

Message exchange: Most communication between cells is done through remote procedure calls (RPCs). 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 fault containment, even though it does not defend against all possible faults.

Remote reads: Cells also read each other's internal data structures directly, which can be substantially faster than exchanging RPCs. 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, it is sanity-checked just as message data is checked.

Remote writes: Cells never write to each other's internal data structures directly, as this would make fault containment impractical. This is enforced by using the FLASH 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 two issues that must be addressed:

- *Choosing which pages to protect:* Each cell always protects the user-level pages that are only used by processes local to that cell. This ensures maximum reliability for most small processes in the system. Each cell also protects as many of the shared pages as possible without causing an excessive number of protection status changes. Protection changes can be expensive: when using the FLASH firewall, revoking remote write permission requires communication with remote nodes to ensure that all valid writes have been delivered to memory. Thus firewall management is a tradeoff between fault containment and performance.
- *Wild writes to unprotected pages:* Wild writes to user pages are a problem because they violate the data integrity expected by users. The chance of data integrity violations must be reduced to near that provided by the memory of the machine, or Hive will not be usable for any important applications.

  Hive attempts to mask corrupt data by preventing corrupted pages from being read by applications or written to disk. However, by the time a cell failure is detected, it is too late to determine which pages have been corrupted. Hive makes the pessimistic assumption that all potentially damaged pages have been corrupted. When a cell failure is detected, all pages writable by the failed cell are preemptively discarded.

Unfortunately, the preemptive discard policy can not prevent all user-visible data integrity violations caused by wild writes. Corrupt data might be used before the cell failure is detected. Alternatively, a faulty cell might corrupt a page, then give up its write permission before the failure is detected, so the page will not be discarded.

This problem appears to be fundamental to a multicellular kernel architecture. The only way to prevent all data integrity violations (without excessive hardware overhead to log updates) 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.

It is unclear at present whether the probability of data integrity violations will be higher in a multicellular system than in a current SMP OS implementation. We intend to evaluate this in future studies. One way to reduce the probability is to shorten the time window within which corrupt data might be used, by detecting failures quickly.

Failure detection is a well-studied problem in the context of distributed systems. For Hive, there are two main issues. Although a halted cell is easily recognizable, a cell that is alive but acting erratically can be difficult to distinguish from one that is functioning correctly. Additionally, if one cell could declare that another had failed and cause it to be rebooted, a faulty cell which 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. A failed check provides a hint 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 distributed agreement algorithm. 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 the failure is correctly confirmed by the agreement algorithm). The window of vulnerability can be reduced by increasing the frequency of checks during normal operation. This is another tradeoff between fault containment and performance.

3.2 Resource sharing architecture

The challenge of resource sharing in Hive is to implement the tight sharing expected from a multiprocessor despite the fault containment boundaries between cells. The mechanisms for resource sharing are implemented through the cooperation of the various kernels, but the policy is implemented outside the kernels, in a user-level process called *Wax*.

This approach is feasible because in Hive, unlike in previous distributed systems, cells are not responsible for deciding how to divide their resources between local and remote requests. Making that tradeoff correctly 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.

Resource sharing mechanisms: The resources that need to be shared particularly efficiently across cell boundaries are memory and processors.

Memory sharing occurs at two levels (Figure 3.2). In *logical-level sharing*, a cell that needs to use a data page from a file can access that page no matter where it is stored in the system. Logical-level sharing supports a globally-shared file buffer cache in addition to allowing processes on different cells to share memory. In *physical-level sharing*, a cell that has a free page frame can transfer control over that frame to another cell. Physical-level sharing balances memory pressure across the machine and allows data pages to be placed where required for fast access on a CC-NUMA machine.

To share processors efficiently, Hive extends the UNIX process abstraction to span cell boundaries. A single parallel process can run threads on multiple cells at the same time. Such processes are called *spanning tasks*. Each cell runs a separate local process containing the threads that are local to that cell. Shared process state such as the address space map is kept consistent among the component processes of the spanning task. This mechanism also supports migration of sequential processes among cells for load balancing.

Resource sharing policy: Intercell resource allocation decisions are centralized in Wax, a multithreaded user-level process (Figure 3.3). Table 3.4 lists some of the allocation decisions made by 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 has to 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.

Wax takes advantage of shared memory and the support for spanning tasks to provide efficient resource management. Wax has a complete, up-to-date view of the system state but is not limited to running on a single cell. The threads of Wax running on different cells can synchronize with each other using standard locks and nonblocking data structures, enabling efficient resource management decisions.

Despite its special privileges, Wax is not a special kind of process. It uses resources from all cells, so its pages are discarded and it exits 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

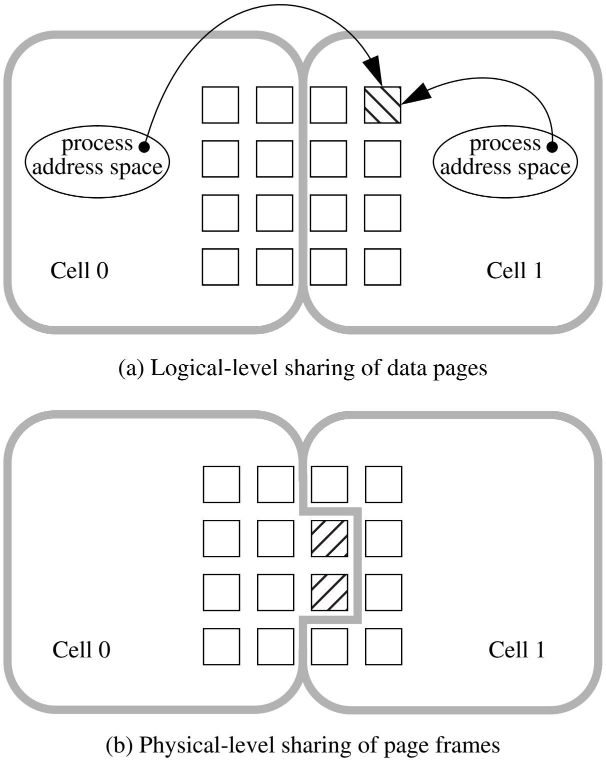

FIGURE 3.2. Types of memory sharing.
In *logical-level* sharing, a process on one cell maps a data page from another cell into its address space. In *physical-level* sharing, one cell transfers control over a page frame to another. One page might be shared in both ways at the same time.

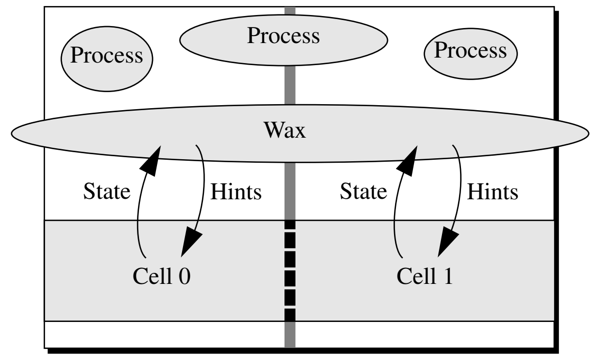

FIGURE 3.3. Intercell optimization using a user-level process. Wax reads state from all cells. Wax provides hints that control the resource management policies that require a global view of the system state. Since Wax is a user-level process, the threads in Wax can use shared memory and synchronize freely without weakening the fault isolation of the cells.

Module	Policy
Page allocator	Which cells to allocate memory from
Virtual memory clock hand	Which cells should be targeted for page deallocation
Scheduler	Gang scheduling, space sharing (granting a set of processors exclusively to a process)
Swapper	Which processes to swap

TABLE 3.4. Examples of policies in each cell driven by Wax.

considerable complexity of trying to recover consistency of Wax's internal data structures after they are damaged by a cell failure.

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 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.

3.3 Implementation status

We have focused development so far on fault containment and memory sharing. Most of the fault containment features of the architecture are implemented and functioning: the internal distributed system, careful reference protocol, wild write defense, and failure hints and recovery. Memory sharing among cells is implemented at both logical-level and physical-level. We have also developed a low-latency intercell RPC subsystem.

The single-system image is only partially complete at present. It provides forks across cell boundaries, distributed process groups and signal delivery, and a shared file system name space. Spanning tasks, Wax, the distributed agreement protocol, and a fault-tolerant file system with single-system semantics remain to be implemented.

The current prototype is sufficient to demonstrate that fault containment is possible in a shared-memory multiprocessor, and that memory sharing can function efficiently without weakening fault containment. Performance results from the current prototype are promising, but further work is required to determine whether a fully-implemented system will perform as well as previous UNIX kernels.

The performance measurements reported in the following sections were obtained using SimOS [18]. We model a machine similar in performance to an SGI Challenge multiprocessor with four 200-MHz MIPS R4000 processors and a 700 nanosecond main memory access latency. We use two types of workloads, characteristic of the two environments we expect to be most common for Hive. For compute-server usage, we use *pmake* (parallel compilation). To model use by large parallel applications, we use *ocean* (scientific simulation) and *raytrace* (graphics rendering). Section 7 describes SimOS, the machine model, and the workloads in detail.

4 Fault Containment Implementation

As described earlier, the three ways one cell can damage another are by sending bad messages, providing bad data to remote reads, and writing to remote addresses. The mechanisms that Hive uses to prevent damage from spreading through messages have proven their effectiveness in previous distributed systems such as NFS. Therefore, we will focus on the novel mechanisms related to remote reads and writes: the careful reference protocol for remote reads, the wild write defense, and aggressive failure detection.

4.1 Careful reference protocol

One cell reads another's internal data structures 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.

An access to the memory of a remote cell can result in a hardware exception. For example, a bus error will occur if the remote node has failed. Cells normally panic (shut themselves down) if they detect such hardware exceptions during kernel execution, because this indicates internal kernel corruption. Some mechanism is needed to prevent errors that occur during remote reads from causing a kernel panic.

Hive uses a simple *careful reference* protocol to avoid these problems, as well as to handle data errors such as linked data structures with loops and values that change unexpectedly. 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 while reading the memory of that cell, the trap handler restores to the saved function context.
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 written by the memory allocator and removed by the memory deallocator. 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 in the reading cell will correctly cause the kernel to panic.

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 (Section 4.3). With simulated 200-MHz processors, the average latency from the initial call to `careful_on` until the terminating `careful_off` call finishes is 1.16 µs (232 cycles), of which 0.7 µs (140 cycles) is the latency we model for 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 µs (Section 6) and requires interrupting a processor on the remote cell.

4.2 Wild write defense

Hive defends against wild writes using a two-part strategy. First, it manages the FLASH hardware 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.

FLASH firewall: The firewall controls which processors are allowed to modify each region of main memory. FLASH provides a separate firewall for each 4 KB of memory, specified as a 64-bit vector where each bit grants write permission to a processor. 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. Only the local processor can change the firewall bits for the memory of its node.

The coherence controller of each node stores and checks the firewall bits for the memory of that node. It checks the firewall on each request for cache line ownership (read misses do not count as ownership requests) and on most cache line writebacks. 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.

We chose a 4 KB firewall granularity to match the operating system page size. Anything larger would constrain operating system memory allocation, whereas it is unclear whether a finer granularity would be useful.

We chose a bit vector per page after rejecting two options that would require less storage. A single bit per page, granting global write access, would provide no fault containment for processes that use any remote memory. A byte or halfword per page, naming a processor with write access, would prevent the scheduler in each cell from balancing the load on its processors.

The performance cost of the firewall is minimal. We ran several of the test workloads twice using a cycle-accurate FLASH memory system model, once with firewall checking enabled and once with it disabled. The firewall check increases the average remote write cache miss latency under *pmake* by 6.3% and under *ocean* by 4.4%. This increase has little overall effect since write cache misses are a small fraction of the workload run time.

Firewall management policy: Firewall management is a tradeoff between fault containment and performance. The only time remote write access to a page is required is when a write-enabled mapping to the page is present in a processor of another cell. However, the set of active hardware mappings changes on each TLB miss, a rate that is far too high to send RPCs requesting firewall status changes. Some other policy is needed to decide when firewall write permission should be granted.

Choosing the correct policy requires careful evaluation under various workloads. At present we use a policy that was straightforward to implement and keeps the number of writable pages fairly small.

Write access to a page is granted to all processors of a cell as a group, 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 it to freely reschedule the process on any of its processors without sending RPCs to remote cells. Write permission remains granted as long as any process on that cell has the page mapped.

The address space region is marked writable only if the process had explicitly requested a writable mapping to the file. Thus this policy ensures that a fault in a cell can only corrupt remote pages to which a process running on that cell had requested write access.

To measure the effectiveness of this policy we used *pmake*, which shares few writable pages between the separate compile processes, and *ocean*, which shares its data segment among all its threads. We observed that, over 5.0 seconds of execution sampled at 20 millisecond 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 behavior of the firewall under *pmake* shows that the current policy should provide good wild write protection to a system used predominately 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`).

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

Preemptive discard: It is difficult to efficiently determine 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 one cell knows the precise firewall status of that page (the *data home* cell, defined in Section 5). Distributing firewall status information during recovery to all cells using the page would require significant communication. Instead, all TLBs are flushed and all remote mappings are removed during recovery. This ensures that a future access to a discarded page will fault and send an RPC to the owner of the page, where it can be checked.

The accesses need to be checked because discarding a page can violate the expected stable write semantics of the file system, if the page was dirty with respect to disk. Processes that attempt to access a discarded dirty page should receive an error. However, the accesses might occur arbitrarily far in the future, making it quite expensive to record exactly which pages of each file have been discarded. We solve this problem by relaxing the process-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.

We take advantage of this in Hive and allow 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 will receive I/O errors. 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 dirty page of a file is discarded, the file's generation number is incremented. An access via a file descriptor or address space region with a mismatched generation number generates an error.

4.3 Failure detection and recovery

Hive attempts to detect the failure of a cell quickly in order to reduce the probability that wild writes will cause user-visible data corruption. This is implemented with consistency checks that run regularly in normal operation. When one of the checks fails, it is confirmed by a distributed agreement algorithm.

Just as in previous distributed systems, a cell is considered potentially failed if an RPC sent to it times out. Additionally, a cell is considered potentially failed if:

- An attempt to access its memory causes a bus error. This will occur if there is a serious hardware failure.
- A shared memory location which it updates on every clock interrupt fails to increment. Clock monitoring detects hardware failures that halt processors but not entire nodes, as well as operating system errors that lead to deadlocks or the inability to respond to interrupts.
- Data or pointers read from its memory fail the consistency checks that are part of the careful reference protocol. This detects software faults.

To prevent a corrupt cell from repeatedly broadcasting alerts and damaging system performance over a long period, a cell that broadcasts the same alert twice but is voted down by the distributed agreement algorithm both times is considered corrupt by the other cells.

The distributed agreement algorithm is an instance of the well-studied group membership problem, so Hive will use a standard algorithm (probably [16]). This is not implemented yet and is simulated by an oracle for the experiments reported in this paper.

Recovery algorithms: Given consensus on the live set of cells, each cell runs recovery algorithms to clean up dangling references and determine which processes must be killed. One interesting aspect of these algorithms is the use of a double global barrier to synchronize the preemptive discard operation. The double barrier in recovery is part of a strategy to increase the speed of page faults that hit in the file cache, an extremely common intercell operation.

When a cell exits distributed agreement and enters recovery, it is not guaranteed that all page faults and accesses to its memory from other cells have finished. User-level processes will be suspended, but processes running at kernel level will not be suspended. (Allowing kernel-level processes to continue during recovery permits the recovery algorithms to grab kernel locks and modify kernel data structures.) Each cell only joins the first global barrier when it has flushed its processor TLBs and removed any remote mappings from process address spaces. A page fault that occurs after a cell has joined the first barrier is held up on the client side.

After the first barrier completes, each cell knows that no further valid page faults or remote accesses are pending. This allows it to revoke any firewall write permission it has granted to other cells and clean up its virtual memory data structures. It is during this operation that the virtual memory subsystem detects pages that were writable by a failed cell and notifies the file system, which increments its generation count on the file to record the loss.

Each cell joins the second global barrier after it has finished virtual memory cleanup. Cells that exit the second barrier can safely resume normal operation, including sending page faults to other cells.

Given this design, the server-side implementation of a page fault RPC need not grab any blocking locks to synchronize with the recovery algorithms. This allows page faults that hit in the file cache to be serviced entirely in an interrupt handler, which has significant performance benefits (Section 5.2).

At the end of every recovery round, a recovery master is elected from the new live set. The recovery master runs hardware diagnostics on the nodes belonging to the failed cells. If the diagnostic checks succeed, the failed cells are automatically rebooted and reintegrated into the system. Reintegration is not yet implemented but appears straightforward.

5 Memory Sharing Implementation

Given the fault containment provided by the features described in the previous section, the next challenge is to share resources flexibly across cell boundaries without weakening fault containment. This section describes Hive's solution to one of the major resource sharing problems, memory sharing among cells.

As described earlier (Figure 3.2) there are two types of memory sharing: logical-level sharing and physical-level sharing. The two types require different data structure management and are implemented at different places in the system.

We found it useful to give names to the three roles that cells can play in sharing a memory page:

- *Client cell:* A cell running a process that is accessing the data.
- *Memory home:* The cell that owns the physical storage for the page. Cell 1 is the memory home in both parts of Figure 3.2.
- *Data home:* The cell that owns the data stored in the page. Cell 1 is the data home in Figure 3.2a, but cell 0 is the data home in Figure 3.2b.

The data home provides name resolution, 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 current prototype the

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)`

TABLE 5.1. Virtual memory primitives for memory sharing.

Total local page fault latency	6.9 µsec	
Total remote page fault latency	50.7 µsec	
Client cell	28.0	
File system		9.0
Locking overhead		5.5
Miscellaneous VM		8.7
Import page		4.8
Data home	5.4	
Miscellaneous VM		3.4
Export page		2.0
RPC	17.3	
Stubs and RPC subsystem		4.9
Hardware message and interrupts		4.7
Arg/result copy through shared memory		4.0
Allocate/free arg and result memory		3.7

TABLE 5.2. Components of the remote page fault latency. Times are averaged across 1024 faults that hit in the data home page cache.

data home for a given page is always the cell that owns the backing store for that page.

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

5.1 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.

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.

5.2 Logical-level sharing of file pages

In Hive, when one cell needs to access a data page cached by another, 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 5.1). These functions are most frequently called as part of page fault processing, which proceeds as follows.

A page fault to a remote file 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. This is a shadow vnode which indicates that the file is remote. The file system uses information stored in the vnode to determine the data home for the file and the vnode tag on the data home, and sends an RPC to the data home. The server side of the file system 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 previous systems. The file system on the data home calls `export` on the page. This records the client cell in the data home's pfdat, which prevents the page from being deallocated and provides information necessary for the failure recovery algorithms. `export` also modifies the firewall state of the page if write access is requested.

The server-side file system returns the address of the data 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 and avoid sending an RPC to the data home. The page also remains in the data home's pfdat hash table, allowing processes on other cells to find and share it. Figure 5.3a 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` rather than putting the page on the local free list. `release` frees the extended pfdat and sends an RPC to the data home, which places the page on the data home free list if no other references remain. Keeping the page on the data home free list rather than client free lists increases memory allocation flexibility for the data home. The data page remains in memory until the page frame is reallocated, providing fast access if the client cell faults to it again.

We measure the overhead of the entire mechanism described in this section by comparing the minimal cost of a page 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 6.9 µs while the remote case averages 50.7 µs in microbenchmarks run on SimOS. Table 5.2 shows a detailed breakdown of the remote page fault latency. 17.3 µs of the remote case is due to RPC costs which are explained in Section 6. Another 14.2 µs (listed in the table as client cell locking overhead and miscellaneous VM) is due to an

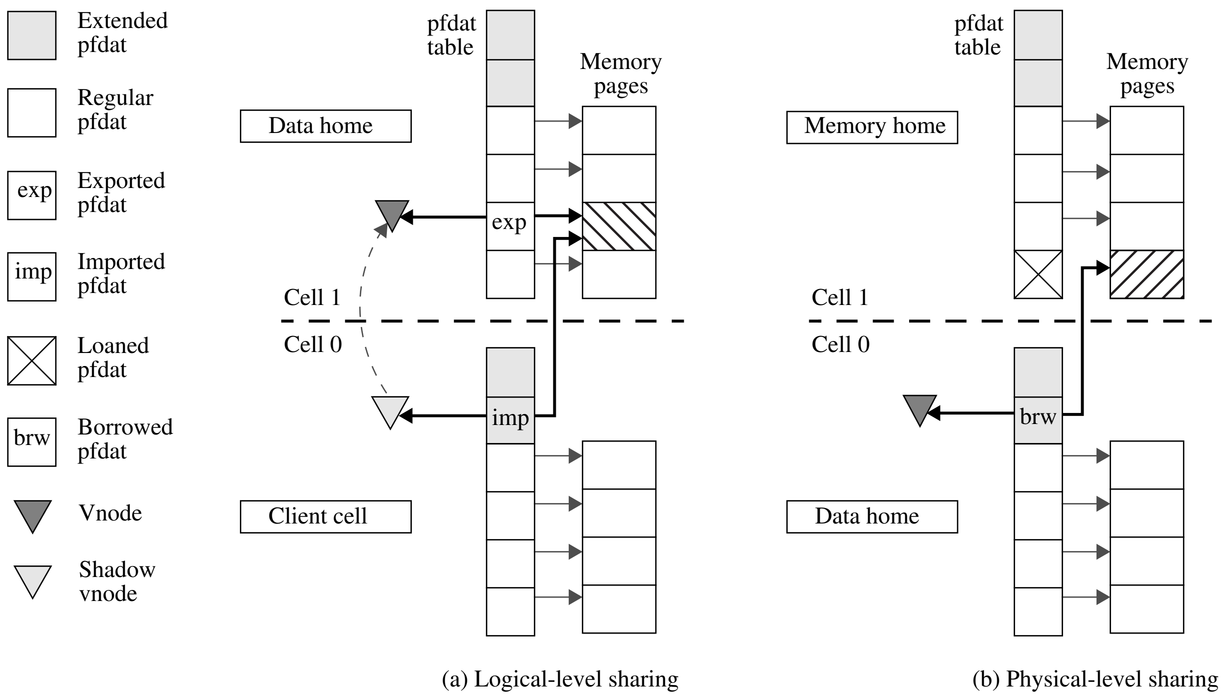

FIGURE 5.3. Implementation of memory sharing.
The entries in the pfdat table bind a logical page id (file, offset) to a physical page frame. In logical-level sharing (a), the data home (cell 1) marks its pfdat as exported and records the identity of the client (cell 0). The data home continues to manage the page. In physical-level sharing (b), the memory home (cell 1) marks its pfdat as loaned to the data home (cell 0) and ignores the page until the data home returns it.

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 practice the remote costs can be somewhat higher, because some of the remote faults cannot be serviced at interrupt level. Faults which encounter certain synchronization conditions at the data home must be queued for an RPC server process, which adds substantial latency (Section 6). To check the overall effect of remote faults, we measured their contribution to the slowdown of *pmake* on a four-cell system compared to a one-cell system. During about six seconds of execution on four processors, there are 8935 page faults that hit in the page cache, of which 4946 are remote on the four-cell system. This increases the time spent in these faults from 117 to 455 milliseconds (cumulative across the processors), which is about 13% of the overall slowdown of *pmake* from a one-cell to a four-cell system. This time is worth optimizing but is not a dominant effect on system performance.

5.3 Logical-level sharing of anonymous pages

The virtual memory system uses nearly the same mechanisms to share anonymous pages (those whose backing store is in the swap partition) as it uses to share file data pages. The interesting difference is the mechanism for finding the requested page when a process takes a page fault.

In IRIX, anonymous pages are managed in copy-on-write trees, similar to the MACH approach [15]. 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. 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. 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, the parent and child processes might be on different cells. There are several different ways to change anonymous page management to respond to this. We chose this issue as the subject for an experiment on the effectiveness of building distributed kernel data structures.

We keep the existing tree structure nearly intact, and allow the pointers in the tree to cross cell boundaries. The leaf node corresponding to a process is always local to a process. Other nodes might be remote. This does not create a wild write vulnerability because the lookup algorithms do not need to modify the interior nodes of the tree or synchronize access to them.

When a child read-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 to set up the export/import binding. The cell that owns the node is always the data home for the anonymous page.

The fact that this implementation appears to work reliably in the face of fault injection experiments (Section 7) indicates that distributed data structures can be built without weakening fault containment. However, we do not observe any substantial performance benefit in this case. 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. A more conventional RPC-based approach would be simpler and probably just as fast, at least for the workloads we evaluated.

5.4 Physical-level sharing

The logical-level design just described for both file data and anonymous data has a major constraint: all pages must be in their data home's page cache. If this design constrained all pages to be stored in the data home's 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 a cell, the memory home, to loan one of its page frames to another cell, which becomes the data home (Figure 5.3b). 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 (except it must send an RPC to the memory home when it needs to change the firewall state).

Frame loaning is usually demand-driven by the page allocator. When the page allocator receives a request, it may decide to allocate a remote frame. Wax will eventually provide the policy support for remote allocation. If a cell decides to allocate remotely, it sends an RPC to the memory home asking for a set of pages.

Borrowed frames are not acceptable for all requests. For example, frames allocated for internal kernel use must be local, since the firewall does not defend against wild writes by the memory home. The page allocator supports constraints by taking two new arguments, 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 in some cases because it results in immediately flushing the data. We have not yet developed a better policy.

5.5 Logical/physical interactions

In general, the two types of 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 can occur when the data home places a page in the memory of the client cell that has faulted to it, which helps 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. This is possible because the logical-level and physical-level state machines use separate storage within each pfdat.

5.6 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.
- The generation number strategy used for preemptive discard (Section 4.2) 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.

The tradeoffs in page allocation between fault containment and performance are complex; we have not yet studied them in enough detail to recommend effective allocation strategies.

5.7 Summary of memory sharing implementation

The key organizing principle of Hive 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 for file data pages and by the copy-on-write manager for anonymous pages.

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.

6 RPC Performance Optimization

We have focused development so far on the fault containment and memory sharing functionality of Hive. However, it was clear from the start that intercell RPC latency would be a critical factor in system performance. RPCs could be implemented on top of normal cache-coherent memory reads and writes, but we chose to add hardware message support to FLASH in order to minimize latency.

Without hardware support, intercell communication would have to be layered on interprocessor interrupts (IPIs) and producer-consumer buffers in shared memory. This approach is expensive if the IPI carries no argument data, as on current multiprocessors. The receiving cell would have to poll per-sender queues to determine which cell sent the IPI. (Shared per-receiver queues are not an option as this would require granting global write permission to the queues, allowing a faulty cell to corrupt any message in the system.) Data in the queues would also ping-pong between the processor caches of the sending and receiving cells.

We added a short interprocessor send facility (SIPS) to the FLASH coherence controller. We combine the standard cache-line delivery mechanism used by the cache-coherence protocol with the interprocessor interrupt mechanism and a pair of short receive queues on each node. Each SIPS delivers one cache line of data (128 bytes) in about the latency of a cache miss to remote memory, with the reliability and hardware flow control characteristic of a cache miss. Separate receive queues are provided on each node for request and reply messages, making deadlock avoidance easy. An early version of the message send primitive is described in detail in [8].

The Hive RPC subsystem built on top of SIPS is much leaner than the ones in previous distributed systems. No retransmission or duplicate suppression is required because the primitive is reliable. 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). 128 bytes is large enough for the argument and result data of most RPCs. The RPC subsystem is also simplified because it supports only kernel-to-kernel communication. User-level RPCs are implemented at the library level using direct access to the message send primitive.

The base RPC system only supports requests that are serviced at interrupt level. The minimum end-to-end null RPC latency measured using SimOS is 7.2 µs (1440 cycles), of which 2 µs is SIPS latency. This time is fast enough that the client processor spins waiting for the reply. The client processor only context-switches after a timeout of 50 µsec, which almost never occurs.

In practice the RPC system can add somewhat more overhead than measured with the null RPC. As shown in Table 5.2, we measured an average of 9.6 µs (1920 cycles) for the RPC component of commonly-used interrupt-level request (excluding the time shown in that table to allocate and copy memory for arguments beyond 128 bytes). The extra time above the null RPC latency is primarily due to stub execution.

Layered on top of the base interrupt-level RPC mechanism is a queuing service and server process pool to handle longer-latency requests (for example, those that cause I/O). A queued request is structured as an initial interrupt-level RPC which launches the operation, then a completion RPC sent from the server back to the client to return the result. The minimum end-to-end null queued RPC latency is 34 µsec, due primarily to context switch and synchronization costs. In practice the latency can be much higher because of scheduling delays.

The significant difference in latency between interrupt-level and queued RPCs had two effects on the structure of Hive. First, we reorganized data structures and locking to make it possible to service common RPCs at interrupt level. Second, common services that may need to block are structured as initial best-effort interrupt-level service routines that fall back to queued service routines only if required.

7 Experimental Results

In this section we report the results of experiments on the Hive prototype. First we describe SimOS and the machine model used for our experiments in more detail. Next we present the results of performance experiments, fail-stop hardware fault experiments, and software fault experiments.

7.1 SimOS environment

SimOS [18] is a complete machine simulator detailed enough to provide an accurate model of the FLASH hardware. It can also run in a less-accurate mode where it is fast enough (on an SGI Challenge) to boot the operating system quickly and execute interactive applications in real time. The ability to dynamically switch between these modes allows both detailed performance studies and extensive testing.

Operating systems run on SimOS as they would run on a real machine. The primary changes required to enable IRIX and Hive to run on SimOS are to the lowest level of the SCSI driver, ethernet and console interfaces. Fewer than 100 lines of code outside the device drivers needed modification.

Running on SimOS exposes an operating system to all the concurrency and all the resource stresses it would experience on a real machine. Unmodified binaries taken from SGI machines execute normally on top of IRIX and Hive running under SimOS. We believe that this environment is a good way to develop an operating system that requires hardware features not available on current machines. It is also an excellent performance measurement and debugging environment [17].

7.2 Simulated machine

We simulate a machine similar in performance to an SGI Challenge multiprocessor, with four 200-MHz MIPS R4000-class processors, 128 MB of memory, four disk controllers each with one attached disk, four ethernet interfaces, and four consoles. The machine is divided into four nodes, each with one processor, 32 MB of memory, and one of each of the I/O devices. This allows Hive to be booted with either one, two or four cells.

Each processor has a 32 KB two-way-associative primary instruction cache with 64-byte lines, a 32 KB two-way-associative primary data cache with 32-byte lines, and a 1 MB two-way-associative unified secondary cache with 128-byte lines. The simulator executes one instruction per cycle when the processor is not stalled on a cache miss.

A first-level cache miss that hits in the second-level cache stalls the processor for 50 ns. The second-level cache miss latency is fixed at the FLASH average miss latency of 700 ns. An interprocessor interrupt (IPI) is delivered 700 ns after it is requested, while a SIPS message requires an IPI latency plus 300 ns when the receiving processor accesses the data.

Disk latency is computed for each access using an experimentally-validated model of an HP 97560 disk drive [9]. SimOS models both DMA latency and the memory controller occupancy required to transfer data from the disk controller to main memory.

There are two inaccuracies in the machine model that affect our performance numbers. We model the cost of a firewall status change as the cost of the uncached writes required to communicate with the coherence controller. In FLASH, additional latency will be required when revoking write permission to ensure that all pending valid writebacks have completed. The cost of this operation depends on network design details that have not yet been finalized. Also, the machine model provides an oracle that indicates unambiguously to each cell the set of cells that have failed after a fault. This performs the function of the distributed agreement protocol described in Section 4.3, which has not yet been implemented.

7.3 Performance tests

For performance measurements, we selected the workloads shown in Table 7.1. These workloads are characteristic of the two ways we expect Hive to be used. *Raytrace* and *ocean* (taken from the Splash-2 suite [21]) are parallel scientific applications that use the system in ways characteristic of supercomputer environments. *Pmake* (parallel make) is characteristic of use as a multi-programmed compute server. In all cases the file cache was warmed up before running the workloads.

We measured the time to completion of the workloads for Hive configurations of one, two, and four cells. For comparison purposes, we also measured the time under IRIX 5.2 on the same four-processor machine model. Table 7.2 gives the performance of the workloads on the various system configurations.

As we expected, the overall impact of Hive's multicellular architecture is negligible for the parallel scientific applications. After a relatively short initialization phase which uses the file system services, most of the execution time is spent in user mode.

Even for a parallel make, which stresses operating system services heavily, Hive is within 11% of IRIX performance when configured for maximum fault containment with one cell per processor. The overhead is spread over many different kernel operations. We would expect the overhead to be higher on operations which are highly optimized in IRIX. To illustrate the range of overheads, we ran a set of microbenchmarks on representative kernel operations and compared the latency when crossing cell boundaries with the latency in the local case.

Table 7.3 gives the results of these microbenchmarks. The overhead is quite small on complex operations such as large file reads and writes. It ranges up to 7.4 times for simple operations such as a page fault that hits in the page cache. These overheads could be significant for some workloads, but the overall performance of pmake shows that they are mostly masked by other

Name	Description
ocean	simulation; 130 by 130 grid, 900 second interval
raytrace	rendering a teapot; 6 antialias rays per pixel
pmake	compilation; 11 files of GnuChess 3.1, four at a time

TABLE 7.1. Workloads and datasets used for tests.

Workload	IRIX 5.2 time (sec)	Slowdowns on Hive		
		1 cell 4 CPUs/cell	2 cells 2 CPUs/cell	4 cells 1 CPU/cell
ocean	6.07	1 %	1 %	–1 %
raytrace	4.35	0 %	0 %	1 %
pmake	5.77	1 %	10 %	11 %

TABLE 7.2. Workload timings on a four-processor machine. As expected, the partition into cells has little effect on the performance of parallel scientific applications. It has a larger effect on compilation, which uses operating system services intensively.

Operation	Local	Remote	Remote/ local
4 MB file read (msec)	65.0	76.2	1.2
4 MB file write/extend (msec)	83.7	87.3	1.1
open file (μsec)	148	580	3.9
page fault that hits in file cache (μsec)	6.9	50.7	7.4

TABLE 7.3. Local vs. remote latency for kernel operations. The overhead of crossing cell boundaries is low for complex operations such as file read and write, but high for operations which are highly optimized in the local case such as quick fault. Times were measured on a two-processor two-cell system using microbenchmarks, with the file cache warmed up.

effects (such as disk access costs) which are common to both SMP and multicellular operating systems.

7.4 Fault injection tests

It is difficult to predict the reliability of a complex system before it has been used extensively, and probably impossible to demonstrate reliability through fault injection tests. Still, fault injection tests can provide an initial indication that reliability mechanisms are functioning correctly.

For fault injection tests in Hive, we selected a few situations that stress the intercell resource sharing mechanisms. These are the parts of the architecture where the cells cooperate most closely, so they are the places where it seems most likely that a fault in one cell could corrupt another. We also injected faults into other kernel data structures and at random times to stress the wild write defense mechanism.

When a fault occurs, the important parts of the system's response are the latency until the fault is detected, whether the damage is successfully confined to the cell in which the fault occurred, and how long it takes to recover and return to normal operation. The latency until detection is an important part of the wild write defense, while time required for recovery is relatively

Injected fault type and workload (P = *pmake*, R = *raytrace*)		# tests	Latency until last cell enters recovery(msec)	
			Avg	Max
Node failure:				
during process creation	P	20	16	21
during copy-on-write search	R	9	10	11
at random time	P	20	21	45
Corrupt pointer in:				
process address map	P	8	38	65
copy-on-write tree	R	12	401	760

TABLE 7.4. Fault injection test results.
The tests used a four processor system booted with four cells. In all tests Hive successfully contained the effects of the fault to the cell in which it was injected.

unimportant because faults are assumed to be rare. We measured these quantities using both the *pmake* and *raytrace* workloads, because multiprogrammed workloads and parallel applications stress different parts of the wild write defense.

We used a four-processor four-cell Hive configuration for all the tests. After injecting a fault into one cell we measured the latency until recovery had begun on all cells, and observed whether the other cells survived. After the fault injection and completion of the main workload, we ran the *pmake* workload as a system correctness check. Since *pmake* forks processes on all cells, its success is taken as an indication that the surviving cells were not damaged by the effects of the injected fault. To check for data corruption, all files output by the workload run and the correctness check run were compared to reference copies.

Table 7.4 summarizes the results of the fault injection tests. In all tests, the effects of the fault were contained to the cell in which it was injected, and no output files were corrupted.

- *Hardware fault injection tests:* We simulated fail-stop node failures by halting a processor and denying all access to the range of memory assigned to that processor. We observe that the latency until the fault is detected always falls within a narrow range. This is an effect of the clock monitoring algorithm, which puts a narrow upper bound on the time until some cell accesses the failed node's memory, receives a bus error, and triggers recovery.
- *Software fault injection tests:* Each software fault injection simulates a kernel bug by corrupting the contents of a kernel data structure. To stress the wild write defense and careful reference protocol, we corrupted pointers in several pathological ways: to address random physical addresses in the same cell or other cells, to point one word away from the original address, and to point back at the data structure itself. Some of the simulated faults resulted in wild writes, but none had any effect beyond the preemptive discard phase of recovery. The careful reference protocol successfully defended cells when they traversed corrupt pointers in other cells.

We also measured the latency of recovery. The latency of recovery varied between 40 and 80 milliseconds, but the use of the failure oracle in these experiments implies that the latency in practice could be substantially higher. We intend to characterize the costs of recovery more accurately in future studies.

Development of the fault containment mechanisms has been substantially simplified through the use of SimOS rather than real hardware. The ability to deterministically recreate execution from a checkpoint of the machine state, provided by SimOS, makes it straightforward to analyze the complex series of events that follow

Feature	Description
Required features:	
Firewall	Access control list per page of memory. This enables each cell to defend against wild writes.
Memory fault model	Interface between the OS and the memory system that specifies how memory behaves when a hardware fault occurs.
Remap region	Range of physical memory addresses that is remapped to access node-local memory. This enables each cell to have its own trap vectors.
Optimizations:	
SIPS	Low-latency interprocessor message send.
Memory cutoff	Coherence controller function that cuts off all remote accesses to the node-local memory. This is used by the cell panic routine to prevent the spread of potentially corrupt data to other cells.

TABLE 8.1. Summary of custom hardware used by Hive.

after a software fault. We expect to continue using SimOS for this type of development even after the FLASH hardware is available.

8 Discussion

The current Hive prototype demonstrates that it is possible to provide significantly better reliability for shared-memory multiprocessors than is achieved by SMP OS implementations. However, there are several issues that must be addressed before we can suggest that production operating systems be constructed using the techniques described in this paper:

Hardware support: Various aspects of the Hive design depend on hardware features that are not standard in current multiprocessors. Table 8.1 summarizes the special-purpose support that we added to FLASH, including a few features not discussed earlier in the paper. Of these features, the firewall requires the most hardware resources (for bit vector storage). The memory fault model requires attention while designing the cache-coherence protocol, but need not have a high hardware cost as long as it does not try to protect against all possible faults.

The hardware features used by Hive appear to allow a range of implementations that trade off among performance, cost, and fault containment. This suggests that a system manufacturer interested in improved reliability could choose an appropriate level of hardware support. We do not see this issue as a barrier to production use of a system like Hive.

Architectural tradeoffs: Significant further work on the Hive prototype is required to explore the costs of a multicellular architecture.

- *Wax:* There are two open questions to be investigated once Wax is implemented. We must determine whether an optimization module that is “out of the loop” like Wax can respond rapidly to changes in the system state, without running continuously and thereby wasting processor resources. We also need to investigate whether a two-level optimization architecture (intracell and intercell decisions made independently) can compete with the resource management efficiency of a modern UNIX implementation.
- *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 OS. Spanning tasks and process migration must be implemented. The resource sharing policies must be systematically extended to consider the fault containment implications of sharing decisions. Some statistical measure is needed to predict the probability of data integrity violations in production operation.
- *File system:* A multicellular architecture requires a fault-tolerant high performance file system that preserves single-system semantics. This will require mechanisms that support file replication and striping across cells, as well as an efficient implementation of a globally coherent and location independent file name space.

Other advantages of the architecture: We also see several areas, other than the reliability and scalability issues which are the focus of this paper, in which the techniques used in Hive might provide substantial benefits.

- *Heterogenous resource management:* For large diverse workloads, performance may be improved by managing separate resource pools with separate policies and mechanisms. A multicellular operating system can segregate processes by type and use different strategies in different cells. Different cells can even run different kernel code if their resource management mechanisms are incompatible or the machine’s hardware is heterogenous.
- *Support for CC-NOW:* Researchers have proposed workstation add-on cards that will provide cache-coherent shared memory across local-area networks [12]. Also, the FLASH architecture may eventually be distributed to multiple desktops. Both approaches would create a cache-coherent network of workstations (CC-NOW). The goal of a CC-NOW is a system with the fault isolation and administrative independence characteristic of a workstation cluster, but the resource sharing characteristic of a multiprocessor. Hive is a natural starting point for a CC-NOW operating system.

9 Related Work

Fault containment in shared-memory multiprocessor operating systems appears to be a new problem. We know of no other operating systems that try to contain the effects of wild writes without giving up standard multiprocessor resource sharing. Sullivan and Stonebraker considered the problem in the context of database implementations [19], but the strategies they used are focused on a transactional environment and thus are not directly applicable to a standard commercial operating system.

Reliability is one of the goals of microkernel research. A microkernel could support a distributed system like Hive and prevent wild writes, as discussed in Section 2. However, existing microkernels such as Mach [15] are large and complex enough that it is difficult to trust their correctness. New microkernels such as the Exokernel [6] and the Cache Kernel [4] may be small enough to provide reliability.

An alternative reliability strategy would be to use traditional fault-tolerant operating system implementation techniques. Previous systems such as Tandem Guardian [2] provide a much stronger reliability guarantee than fault containment. However, full fault tolerance requires replication of computation, so it uses the available hardware resources inefficiently. While this is appropriate when supporting applications that cannot tolerate partial failures, it is not acceptable for performance-oriented and cost-sensitive multiprocessor environments.

Another way to look at Hive is as a distributed system where memory and other resources are freely shared between the kernels. The NOW project at U.C. Berkeley is studying how to couple a cluster of workstations more tightly for better resource sharing [1].

The hardware they assume for a NOW environment does not provide shared memory, so they do not face the challenge of wild writes or the opportunity of directly accessing remote memory. However, much of their work is directly applicable to improving the resource management policies of a system like Hive.

The internal distributed system of Hive requires it to synthesize a single-system image from multiple kernels. The single-system image problem has been studied in depth by other researchers (Sprite [13], Locus [14], OSF/1 AD TNC [22]). Hive reuses some of the techniques developed in Sprite and Locus.

10 Concluding Remarks

Fault containment is a key technique that will improve the reliability of large-scale shared-memory multiprocessors used as general-purpose compute servers. The challenge is to provide better reliability than current multiprocessor operating systems without reducing performance.

Hive implements fault containment by running an internal distributed system of independent kernels called cells. The basic memory isolation assumed by a distributed system is provided through a combination of write protection hardware (the firewall) and a software strategy that discards all data writable by a failed cell. The success of this approach demonstrates that shared memory is not incompatible with fault containment.

Hive strives for performance competitive with current multiprocessor operating systems through two main strategies. Cells share memory freely, both at a logical level where a process on one cell accesses the data on another, and at a physical level where one cell can transfer control over a page frame to another. Load balancing and resource reallocation are designed to be driven by a user-level process, Wax, which uses shared memory to build a global view of system state and synchronize the actions of various cells. Performance measurements on the current prototype of Hive are encouraging, at least for the limited tests carried out so far.

Finally, the multicellular architecture of Hive makes it inherently scalable to multiprocessors significantly larger than current systems. We believe this makes the architecture promising even for environments where its reliability benefits are not required.

Acknowledgments

We are grateful to Silicon Graphics, Inc. for giving us access to the IRIX source code. Frans Kaashoek, the other program committee members, and the reviewers provided valuable comments that substantially improved this paper. We thank the SimOS development team for supporting this work and adding fault injection functionality to the system.

This work was supported in part by ARPA contract DABT63-94-C-0054. John Chapin is supported by a Fannie and John Hertz Foundation fellowship. Mendel Rosenblum is partially supported by a National Science Foundation Young Investigator award. Anoop Gupta is partially supported by a National Science Foundation Presidential Young Investigator award.

References

[1] T. Anderson, D. Culler, and D. Patterson. "A Case for NOW (Networks of Workstations)." *IEEE Micro* 15(1):54-64, February 1995.

[2] J. Bartlett, J. Gray, and B. Horst. "Fault Tolerance in Tandem Computer Systems." In *Evolution of Fault-Tolerant Computing*, pp. 55-76, Springer-Verlag, 1987.

[3] D. Chaiken and A. Agarwal. "Software-Extended Coherent Shared Memory: Performance and Cost." In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 314-324, April 1994.

[4] D. Cheriton and K. Duda. "A Caching Model of Operating System Kernel Functionality." In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pp. 179-193, November 1994.

[5] Convex Computer Corporation. *Convex Exemplar: System Overview*. Order No. 080-002293-000, 1994.

[6] D. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture For Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor." In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 38-50, October 1994.

[9] D. Kotz, S. Toh, and S. Radhakrishnan. "A Detailed Simulation of the HP 97560 Disk Drive." Technical Report PCS-TR94-20, Dartmouth University, 1994.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford FLASH Multiprocessor." In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 302-313, April 1994.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. "The Stanford DASH Multiprocessor." *Computer* 25(3):63-79, March 1992.

[12] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Park. "The S3.mp Scalable Shared Memory Multiprocessor." In *Proceedings of 27th Hawaii International Conference on Systems Sciences*, pp. 144-153, January 1994.

[13] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. "The Sprite Network Operating System." *Computer* 21(2):23-36, February 1988.

[14] G. Popek and B. Walker (eds.). *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[15] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computers* 37(8):896-908, August 1988.

[16] A. Ricciardi and K. Birman. "Using Process Groups To Implement Failure Detection in Asynchronous Environments." In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 341-353, August 1991.

[17] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, and A. Gupta. "The Impact of Architectural Trends on Operating System Performance." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[18] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. "Fast and Accurate Multiprocessor Simulation: The SimOS Approach." *IEEE Parallel and Distributed Technology* 3(4), Fall 1995.

[19] M. Sullivan and M. Stonebraker. "Improving Software Fault Tolerance in Highly Available Database Systems." Technical reports UCB/ERL M90/11, University of California, Berkeley, 1990, and UCB/ERL M91/56, 1991.

[20] M. Sullivan and R. Chillarege. "Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems." In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pp. 2-9, June 1991.

[21] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations." In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[22] 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." In *Proceedings of the Winter 1993 USENIX Conference*, pp. 449-468, January 1993.