

OPERATING SYSTEMS (CS 384)

FINAL PAPER

FAULT CONTAINMENT FOR SHARED-MEMORY

MULTIPROCESSORS

BY: SALIM HARUNANI

FEBRUARY 2, 2003

TABLE OF CONTENTS

ABSTRACT.....	2
INTRODUCTION	3
FAULT CONTAINMENT IN SHARED-MEMORY MULTIPROCESSORS	5
Hardware faults:.....	5
Software faults:	7
HIVE ARCHITECTURE	8
Fault Containment Architecture:.....	8
Resource Sharing Architecture:	10
Resource sharing mechanisms:	11
Resource sharing policy:	12
CONCLUSION	14
GLOSSARY.....	15
REFERENCES.....	16

TABLE OF FIGURES

Figure 1: FLASH architecture.....	6
Figure 2: Logical-level sharing of data pages	11
Figure 3: Physical-level sharing of page frames	12
Figure 4: Intercell optimization using a user-level process.....	13

ABSTRACT

The major concerns when designing operating systems for large-scale shared memory multiprocessors are reliability and scalability. To address these issues, I will be referring to Hive, an operating system prototype developed in Stanford University. The structure of Hive contains 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 entire system. Scalability is improved because processes running on different cells share few kernel resources. The Hive prototype is a complete implementation of UNIX SVR4 and is targeted to run on the Stanford FLASH multiprocessor, which is a multiprocessor developed in Stanford University.

This paper focuses on Hive's solution to fault containment and memory sharing among cells. 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.

INTRODUCTION

Due to the excellent performance of shared-memory multiprocessors under dynamic multiprogrammed workloads, they are becoming an increasingly common server platform. 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, such as Stanford DASH, MIT Alewife, Convex Exemplar, etc.

Hive is structured as an internal distributed system of independent kernels called *cells*. This multicellular kernel architecture has two main advantages:

Reliability - A hardware or software fault in SMP OS implementations causes the entire system to crash, thus resulting in an unacceptably low mean time to failure for large-scale machines. Since Hive is made up of cells, only the cell where the fault occurs crashes, affecting only the processes using the resources of that shell. This is beneficial for computer server workloads where there are multiple independent processes. Also, scheduled hardware maintenance and kernel software upgrades can be done one cell at a time.

Scalability - Since all processors in SMP OS implementations share all kernel resources, these systems are difficult to scale to large machines. With Hive, this is not the case since processes running on different cells share few kernel resources. Therefore, scaling can be done by increasing the number of cells.

However, the multicellular architecture of Hive also creates new implementation challenges, one of which is 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.

The solutions to this problem rely on hardware as well as software mechanisms. Hive was designed in conjunction with the Stanford FLASH multiprocessor, which enabled addition of 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, preventing any corrupt data from being read subsequently by applications or written to disk.
- Aggressive failure detection reduces the delay until preemptive discard occurs.

FAULT CONTAINMENT IN SHARED-MEMORY MULTIPROCESSORS

Fault Containment is a general reliability strategy that is implemented in many distributed systems. It is a good reliability strategy for multiprocessors used as general-purpose compute servers. The workloads characteristic of this environment frequently contains multiple independent processes, so some processes can continue doing useful work even if others are terminated by a partial system failure.

If the operating system manages resources well, fault containment will have reliability benefits. If an operating system allocates resources randomly, most applications are prone to failure. Therefore, resource management is important for fault containment. It can therefore be said that 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.

The fault containment strategy can be applied in distributed systems as well as multiprocessors. In addition to the problems that arise in distributed systems, the shared-memory hardware of multiprocessors increases vulnerability to both hardware faults and software faults.

Hardware faults:

To determine hardware faults, I will be referring to the Stanford FLASH, which is shown below:

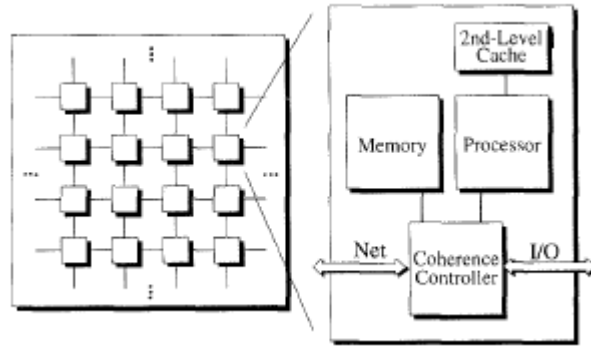


Figure 1: FLASH architecture

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. This type of a machine 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 this kind of a system, the node is the most important unit of failure. A node failure halts a processor, due to which 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.

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. Therefore, processes 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.

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.

Software faults:

Software faults may make a cell vulnerable to wild writes, due to the presence of shared memory. According to recent studies, software faults are more common than hardware faults. It has been reported that out of 3000 severe bugs reported in IBM operating systems over a five-year period, 15 to 25 percent caused wild writes. Current shared-memory multiprocessors do not provide a mechanism to prevent wild writes. Currently, write requests can be halted by virtual address translation hardware present in each processor, which is under the control of the same software whose faults must be protected against.

The operating system must therefore use special-purpose hardware, or rely on a trusted software base that takes control of the existing virtual address translation hardware.

It can therefore be seen that for hardware faults, each cell's wild write defense depends only on the hardware and software of that cell. For software faults, each cell's wild write defense depends only on the hardware and trusted software layer of all other cells.

HIVE ARCHITECTURE

Hive is structured as a set of cells. 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 in an independent operating system. The cells cooperate to present the required single system image to user-level processes.

The Hive architecture is divided into two parts. The fault containment architecture and the resource sharing architecture:

Fault Containment Architecture:

A fault in one cell can damage another cell by the following channels:

- Sending a bad message.
- Providing bad data or errors to remote reads.
- Causing erroneous remote writes.

I will now discuss each of the above operating system fault containment problems.

Message exchange - Since most communication between cells is done by Remote Procedure calls(RPCs), each cell sanity-checks all information received from other cells and sets timeouts when it is waiting for a reply. This approach provides good fault containment, but does not defend against all possible faults.

Remote Reads - Sometimes, cells read each other's internal data structures directly, which can be a lot faster than RPCs. The reading cell has 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 defense is implemented in Hive with a careful reference protocol that includes checks for the various possible error conditions. After the data has been safely read, it is sanity-checked like the way message data is checked.

Remote writes: In Hive, cells never write to each other's internal data structures directly, since this makes fault containment impractical. Therefore the FLASH firewall was used to protect kernel code and data against remote writes. However, cells frequently write to each other's user-level pages since processes running on different cells can share pages. This creates two issues:

- Choosing which pages to protect – Each cell protects the user-level pages that are only used by the processes local to that cell.
- Wild writes to unprotected pages – Wild writes to user pages are a problem because they violate the data integrity expected by users. For Hive to be used for important applications, the chance of data integrity violations must be reduced to near that provided by the memory of the machine.

Hive masks the corrupt data by preventing corrupted pages from being read by applications or written to disk. Since it is too late to determine which pages have been corrupted by the time a fault occurs, Hive assumes that all potentially

damaged pages have been corrupted. Thus, when a cell failure is detected, all pages writable by the failed cell are preemptively discarded.

The preemptive discard policy mentioned above cannot prevent all user-visible data integrity violations caused by wild writes and there is a possibility of corrupt data being 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. This way, the page will not be discarded.

To detect cell failure, Hive uses a two-part solution. First, cells monitor each other during normal operation with a number of heuristic checks. Then, 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 failed, user processes remain suspended until the system has been restored to a consistent state and all potentially corrupt pages have been discarded.

This ensures that vulnerability to wild writes lasts only until the first check fails. To reduce vulnerability, the frequency of checks can be increased.

Resource Sharing Architecture:

There should be tight sharing of resources despite the fault containment boundaries between cells. The mechanisms for resource sharing in Hive 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 possible since in Hive, cells are not responsible for deciding how to divide their resources between local and remote requests. Each cell is responsible only for maintaining its internal correctness and for optimizing performance within the resources it has been allocated.

Resource sharing mechanisms:

The resources that need to be shared efficiently across cell boundaries are memory and processors.

Memory sharing occurs at two levels, logical level and physical level. 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. This level sharing supports a globally-shared file buffer cache in addition to allowing processes on different cells to share memory.

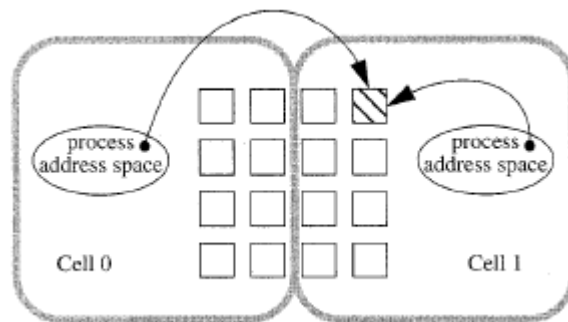


Figure 2: Logical-level sharing of data pages

In physical level sharing, a cell that has a free page frame can transfer control over that frame to another cell. This 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.

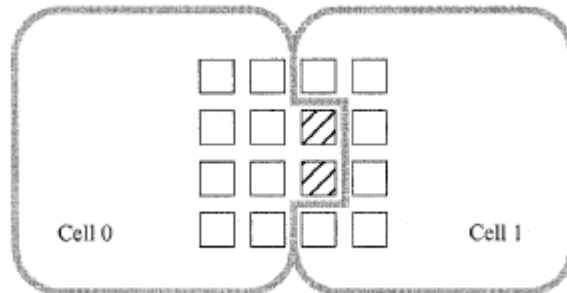


Figure 3: Physical-level sharing of page frames

For efficient sharing of processors, Hive extends the UNIX process abstraction to span cell boundaries. A single process can run threads on multiple cells at the same time. Each cell runs a separate local process containing the threads that are local to that cell. This mechanism supports migration of sequential processes among cells for load balancing.

Resource sharing policy:

The inter cell resource allocation decisions are centralized in Wax, a multithreaded user-level process in Hive.

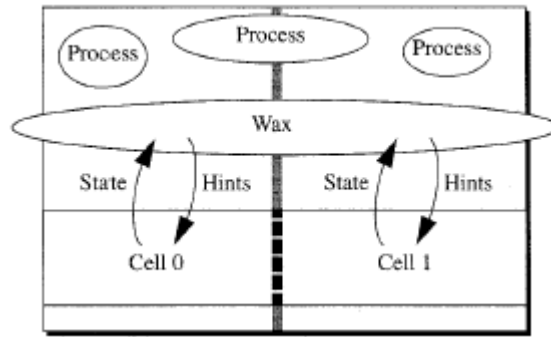


Figure 4: Intercell optimization using a user-level process

Wax addresses a problem faced by previous distributed systems, which were limited to two 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. It can also 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. The threads of Wax running on different cells can synchronize with each other enabling efficient resource management decisions.

Another good thing is that Wax does not weaken the fault containment boundaries between cells. Each cell protects itself by sanity checking the input it receives from Wax. Also, system correction operations are handled through remote procedure calls (RPCs), and not by Wax. Therefore, if a faulty cell damages Wax, it can hurt system performance but not correctness.

CONCLUSION

The test results for fault containment in Hive showed that Hive contained the effects of the fault to the cell in which it was injected. This indicates that to provide fault containment in memory shared multiprocessors, it is better to have cells. This technique promises reliability of large-scale shared memory multiprocessors and does not reduce performance.

Hive is a very good prototype of an operating system that illustrates fault containment for memory-shared multiprocessors, since it takes care of the memory isolation and also performance.

Memory isolation can be provided through a combination of write protection hardware and a software strategy that discards all data writable by a failed cell.

For better performance, cells share memory freely, both at the logical level and the physical level. At the logical level, a process on one cell accesses the data on another. At the physical level, one cell can transfer control over a page to frame to another. Wax, which is a user-level process in Hive, uses shared memory to build a global view of system state and synchronize the actions of various cells.

Finally, such an architecture is easy to scale to large multiprocessors, and thus can be very useful.

GLOSSARY

Hive – An operating system prototype developed in Stanford University

Cells - An internal distributed system of independent kernels

FLASH – A multiprocessor developed in Stanford University

Wax - A multithreaded user-level process in Hive.

REFERENCES

1. 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
2. T. Anderson, D. Culler, and D. Patterson. "A Case for NOW(Networks of Workstations)." *IEEE Micro* 15(1). 54-64. February 1995.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman. 1993.
4. 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.
5. J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu and A. Gupta. "Hive: Fault Containment for Shared Memory Multiprocessors".
<<http://www.ece.cmu.edu/~eno/papers/os/hive.pdf>>. December 1995.

6. Suman Kumar. "Hive: Fault Containment for Shared Memory Multiprocessors Main Points."<

<http://www.ece.cmu.edu/~ganger/712.fall01/summaries/hive.ps>>. November 2001.

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.