Hardware Fault Containment in Scalable Shared-Memory Multiprocessors

Dan Teodosiu, Joel Baxter, Kinshuk Govil, John Chapin[*],
Mendel Rosenblum, and Mark Horowitz

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

Abstract

Current shared-memory multiprocessors are inherently vulnerable to faults: any significant hardware or system software fault causes the entire system to fail. Unless provisions are made to limit the impact of faults, users will perceive a decrease in reliability when they entrust their applications to larger machines. This paper shows that fault containment techniques can be effectively applied to scalable shared-memory multiprocessors to reduce the reliability problems created by increased machine size.

The primary goal of our approach is to leave normal-mode performance unaffected. Rather than using expensive fault-tolerance techniques to mask the effects of data and resource loss, our strategy is based on limiting the damage caused by faults to only a portion of the machine. After a hardware fault, we run a distributed recovery algorithm that allows normal operation to be resumed in the functioning parts of the machine.

Our approach is implemented in the Stanford FLASH multiprocessor. Using a detailed hardware simulator, we have performed a number of fault injection experiments on a FLASH system running Hive, an operating system designed to support fault containment. The results we report validate our approach and show that in conjunction with an operating system like Hive, we can improve the reliability seen by unmodified applications without substantial performance cost. Simulation results suggest that our algorithms scale well for systems up to 128 processors.

1 Introduction

Scalable shared-memory multiprocessors are becoming an increasingly common computing platform. Several companies, including HP-Convex [20], Sequent [11], and Silicon Graphics [10], are shipping multiprocessor systems with configurations of up to a few hundred processing nodes.

However, current shared-memory multiprocessors are inherently vulnerable to faults: any significant hardware or system software fault will cause the entire system to fail. Unless provisions are made to limit the impact of faults, users will perceive a decrease in reliability when they entrust their applications to larger machines. This is an important problem for the viability of large-scale shared-memory multiprocessors as general-purpose compute servers.

Fault containment is a reliability technique widely used in distributed systems, in which the effects of a fault are limited to a small portion of a system [15]. In a computing system that provides fault containment, the chance of failure for a task depends only on the amount of resources that the task uses, not on the size of the entire system.

When running on a large-scale shared-memory multiprocessor, applications of small to moderate size will benefit from the increased reliability offered by fault containment. Since these applications only require a limited amount of resources, they will be protected from failures that occur in the parts of the machine they do not use. Fault containment can also provide benefits to large parallel applications that use a substantial fraction of the machine, if these applications are structured to cope with the loss of some of their resources.

Several challenging issues arise when one tries to implement fault containment on a shared-memory machine. The tight coupling provided by shared memory allows the effects of hardware faults to spread much faster and to more nodes than in a traditional distributed system. In addition, many of the fault containment techniques used in distributed systems (such as performing consistency checks on all incoming messages) appear to be prohibitively expensive given the low latency requirements of a high-performance memory system.

This paper describes one method for implementing fault containment in a scalable shared-memory multiprocessor without substantially reducing its performance or increasing its cost. Rather than relying on expensive techniques that mask the effects of data loss, our approach is based on limiting the impact of faults and running a distributed algorithm to bring the machine back to an operational state after a failure. We have added support to the FLASH multiprocessor [7][9] to confine the effects of most hardware and system software faults. Using the SimOS simulation environment [19], we have validated our approach by performing a number of fault injection experiments on a detailed simulation of the machine.

Achieving fault containment in a shared-memory multiprocessor requires careful design of both its hardware and its operating system. Current multiprocessor operating systems are unable to cope with the loss of any essential hardware resource, such as the failure of a processor or a memory board. This issue has been addressed in our previous work on the Hive operating system [3][18] that was developed in conjunction with the FLASH multiprocessor. This paper focuses on the hardware and firmware support required for an operating system such as Hive that provides fault containment. The experimental results reported in the paper show that, in conjunction with Hive, our implementation offers fault containment benefits to unmodified applications running on FLASH.

[*]Author's present address: Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

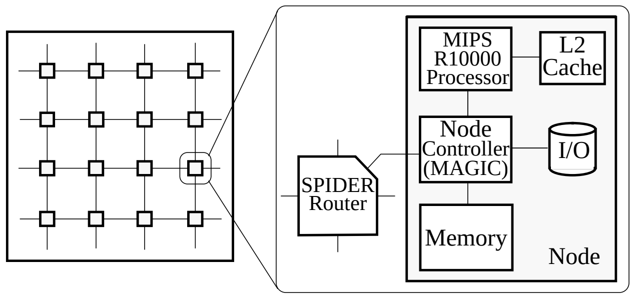


FIGURE 2.1. Overview of the FLASH multiprocessor. FLASH consists of a set of nodes connected through a point-to-point interconnect. Each node contains a portion of the distributed main memory and a node controller that handles cache coherence and other communication within the node and with other nodes.

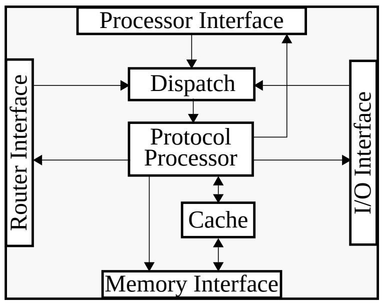


FIGURE 2.2. The MAGIC programmable node controller. The programmable node controller in FLASH contains a RISC processor core that executes handlers to process incoming messages. Handlers are efficiently dispatched by a configurable dispatch mechanism.

The remainder of the paper is organized as follows. In Section 2 we summarize the relevant architectural features of FLASH. Section 3 describes the impact of hardware and firmware faults on the machine, and the features we have added to FLASH to limit this impact. Section 4 presents the distributed recovery algorithm that is run after a failure to restore normal operation on the functioning parts of the machine. We conclude with an experimental evaluation of the effectiveness and scalability of our approach.

2 FLASH Architecture Overview

The FLASH multiprocessor currently being built at Stanford is a CC-NUMA (Cache-Coherent with Non-Uniform Memory Access time) machine that can accommodate up to 512 processors in its largest configuration (Figure 2.1). Each node in the machine contains a MIPS R10000 processor [25] and its second-level cache, a portion of the machine's distributed main memory, a programmable node controller, and attached I/O devices such as disks or network adapters.

The programmable node controller, called MAGIC, manages cache coherence between nodes, as well as all other communication within the node and with other nodes [9]. MAGIC (Figure 2.2) contains a statically scheduled, dual-issue RISC processor core that executes code sequences called *handlers*. The handlers update protocol state, control communication, and manage MAGIC's resources. On receiving a request from the processor, the interconnect, or the node's I/O devices, MAGIC uses a configurable hardware dispatch mechanism to efficiently select the proper handler to respond to that request. The code and data for the protocol processor are stored in a reserved portion of the node's main memory, and may be cached on MAGIC.

By changing the MAGIC handlers it is possible to use a variety of different cache coherence protocols in FLASH. In this paper we assume the use of a directory-based protocol [9] which assigns each 128-byte memory line to a fixed *home node*, where the protocol state for that line is stored. Memory data from one node may be cached by the processor of any other node. Handlers running at the home node for a cache line manage the cache coherence protocol state of that line.

Nodes in FLASH communicate through a high-speed, reliable, flow-controlled interconnect that is based on Silicon Graphics's CrayLink technology and SPIDER router chip [4]. For simplicity, in this paper we shall assume that the nodes are arranged in a two-dimensional mesh. Although such a topology could be realized with the SPIDER routers, FLASH actually uses a hierarchical fat hypercube topology that has a smaller diameter than a mesh with the same number of nodes. The algorithms described in the paper are independent of the actual topology of the machine.

3 Fault Containment in FLASH

Our strategy for coping with hardware failures in FLASH is based on limiting the amount of state lost after a failure and ensuring that the functioning parts of the system are brought back to an operational state. When a fault occurs, the operation of the system is disrupted for a brief period of time to perform the required clean-up and recovery actions. This approach makes it possible to provide undiminished performance during normal operation.

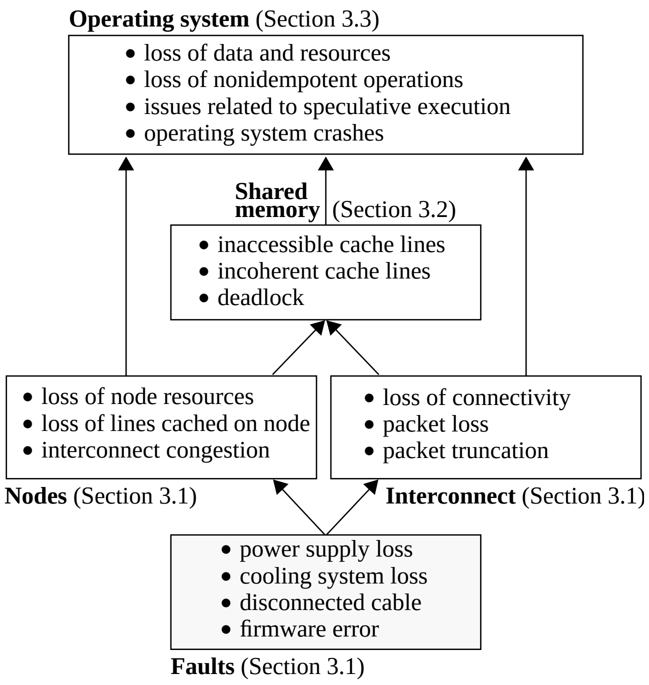
Our fault containment design can thus be broken down into a *set of features* that limit the impact of faults and allow the system to operate despite missing hardware resources, and a *recovery algorithm* that is executed to restore the system to normal operation after a hardware failure. The set of features is introduced in this section, while Section 4 discusses the recovery algorithm.

Figure 3.1 is a roadmap of our presentation of the impact of faults on the system and of the features needed to limit this impact. We start with an overview of the effect of node and interconnect faults on the operation of the machine (a discussion of low-level hardware issues has been relegated to Section 6). We then show how disruptions in the operation of the hardware or firmware can affect the shared memory system, and point out what corrective actions have to be taken. Finally, we discuss how these failures can affect the operating system. The impact of operating system software errors lies beyond the scope of this paper; see [3] and [18] for a detailed treatment of this topic.

3.1 Direct Impact of Faults

As shared-memory multiprocessor systems scale to larger numbers of processors, the probability of a hardware fault increases. The problems we expect to see in practice include partial power supply or cooling system losses, disconnected inter-cabinet cables, node controller firmware errors, and memory failures. For the purpose of this discussion, the direct impact of faults on the machine can be divided into node failures and interconnect failures. Although we discuss the two types of failures separately, a combination of these effects may occur after any given hardware fault.

Node failures: When a node fails, the node's MAGIC controller, processor, portion of main memory, and attached I/O devices all become unavailable. Any information stored in the failed node's processor and MAGIC caches is lost. A node failure may affect the rest of the system by causing other processors to stall and the interconnect to become congested:



Operating system (Section 3.3)
• loss of data and resources
• loss of nonidempotent operations
• issues related to speculative execution
• operating system crashes
Shared memory (Section 3.2)
• inaccessible cache lines
• incoherent cache lines
• deadlock
• loss of node resources
• loss of lines cached on node
• interconnect congestion
Nodes (Section 3.1)
• loss of connectivity
• packet loss
• packet truncation
Interconnect (Section 3.1)
• power supply loss
• cooling system loss
• disconnected cable
• firmware error
Faults (Section 3.1)

FIGURE 3.1. Impact of hardware and firmware faults.
In a scalable shared-memory multiprocessor, hardware and firmware faults may disrupt the functioning of the machine's interconnect, nodes, shared memory, and operating system.

- If a processor sends a coherence request to a failed node, the processor may stall waiting for the request to complete. Note that even if the system software carefully avoids sending any requests to failed nodes, it cannot fully control what references the processors issue. Dynamically scheduled out-of-order execution processors, such as the MIPS R10000 used in FLASH, may speculatively execute code at the predicted target of a branch before it is known whether or not that branch will be taken (the R10000 can speculate through up to four branches). If a branch prediction turns out to be incorrect, the processor discards the results of the speculation, but has already issued any speculatively executed memory references. This means that during incorrect speculation, the processor can issue a memory request for an address that the code does not actually intend to access.
- A node failure has the potential to disrupt the functioning of the whole interconnect. Consider the case in which due to a firmware bug one of the MAGIC handlers enters an infinite loop. After its buffers fill up, the node controller stops accepting packets from the interconnect. If requests continue to be sent to the failed node, traffic will start backing up into the interconnect. As router buffers fill up, the congestion may eventually prevent other interconnect traffic from being delivered. Within a brief time after the failure, coherence traffic throughout the system may come to a standstill.

Limiting the impact of node failures: To avoid sending messages to failed nodes during normal operation, MAGIC uses a configurable hardware table, called the *node map,* that records the availability of all nodes in the system. Each node checks its local node map before sending a request out over the interconnect. The information in the node map is kept up-to-date by the recovery algorithm. The node map checks are implemented entirely in MAGIC hardware and do not add any latency to MAGIC handlers.

Node failures caused by MAGIC firmware faults can cause congestion of the interconnect. If this occurs, it is the responsibility of the recovery algorithm to restore normal network operation by reprogramming the routers bordering the failed area to discard packets routed into it.

Interconnect failures: Having a reliable, flow-controlled interconnect is critical for good performance during normal operation of the multiprocessor. Unfortunately, designing the rest of the system to take advantage of these interconnect features implies that an interconnect failure can disrupt the functioning of the system, by causing loss of connectivity and packet loss or truncation:

- In the interconnect used in FLASH, packets sent between two given nodes travel along a static route that is defined by routing tables programmed into the individual SPIDER routers. Therefore, loss of links and routers (for example after a cabinet power failure) can reduce the connectivity of the interconnect. After a failure, any messages routed through the failed portion of the system will not reach their destinations, even if alternate paths around the faulty area exist.
- During normal operation, the CrayLink interconnect guarantees that no messages are lost or corrupted. However, packets may be dropped or truncated as a result of a failure. For instance, if the power supply to a part of the interconnect goes down, any messages that were in transit through that part of the interconnect will be lost. If the link a message is currently traversing fails, the message will be truncated and a truncated packet will be delivered to the destination.

Limiting the impact of interconnect failures: Connectivity between the functioning nodes can be restored by reprogramming the routing tables so that packets are routed around the failed areas. Loss of connectivity is thus a temporary condition that can be corrected by the recovery algorithm.

For efficiency reasons, we have chosen to let the cache coherence protocol and the operating system deal with packet loss and truncation rather than hiding those effects through the use of a reliable end-to-end transmission protocol. An end-to-end protocol that would buffer messages and perform all necessary retransmissions for coherence traffic would be prohibitively expensive to implement in MAGIC firmware. It takes MAGIC less than 120 nanoseconds, or 24 protocol processor instructions, to execute the handler for processing a read request from a remote node. Any instructions added to this handler for end-to-end checks would slow it down considerably and degrade the performance of the machine. On the other hand, a hardware implementation of end-to-end reliability can be quite effective, as shown in [23].

If a truncated message is received by a node controller, it is not generally possible to discard the message. To reduce the latency of memory operations in FLASH, data is pipelined as it passes through the memory, the interconnect, and MAGIC. When a node controller detects a truncated message, it may have already fed part of that message to the processor. In those cases, MAGIC completes the message, setting the parity error bits on all data words that were lost, and initiates the recovery algorithm.

3.2 Impact of Faults on Shared Memory Operation

Node failures and packet loss or truncation can damage the correctness of the cache coherence protocol. These effects depend on the type of coherence protocol used. In the case of the directory-based protocol currently used in FLASH, a failure can have the following impacts:

- Any cache line whose home is on a failed node becomes *inaccessible*. Even if this line happened to be cached elsewhere in the system at the time its home node failed, it can no longer be used because all cache misses to it are directed to and must be serviced by its home node.

- A cache line can become *incoherent* in two cases: if the line was cached exclusive by a failed node, or if a message containing the only valid copy of the line is dropped or truncated. Although the home node (as well as the coherence protocol state) of an incoherent line remains accessible after a failure, the only valid copy of that line has been lost. For example, a cache line can become incoherent if a cache writeback message is lost, since the cache coherence protocol used in FLASH entrusts the only valid copy of that line to the writeback message for efficiency reasons.
- Memory operations may *deadlock* after a failure. The cache coherence protocol used in FLASH sometimes locks a memory line to prevent access when the line is in a transient state. (This can occur, for instance, when the home node is waiting for invalidate acknowledgments from the nodes sharing a line.) An access to a locked line fails and is retried until it succeeds. If the unlocking message is lost or if the sender of this message fails, a node trying to access the locked line will spin indefinitely waiting for the lock to be released.

Limiting the impact of faults: In order to allow normal shared memory operation to resume after a failure, the recovery algorithm must determine exactly what data has been lost and reset the state of the cache coherence protocol to break any deadlock that may have occurred. After resumption of normal operation, the processors must be prevented from using lost or incoherent data. The following support is needed:

- Any memory references to inaccessible cache lines must be caught and terminated immediately. This is done by having the node map mechanism described previously act as one of the inputs to the MAGIC dispatch table. If a node tries to reference an inaccessible line, an error handler is dispatched on that node's MAGIC. Rather than relaying the memory request to the failed home node, the error handler terminates the reference by raising a bus error.

  Special treatment is required for the set of cache lines corresponding to low physical addresses in FLASH. This is due to the fact that for most processors, including the R10000, exception vectors are located at a fixed range of physical addresses. Requiring all processors in the machine to fetch their vectors from the same memory range would introduce a single point of failure into the system. We have avoided this problem in FLASH by replicating the address range containing the exception vectors. The node controllers contain a hardware *remap* mechanism that converts all references to the vector address range into node-local references.
- Processors must be prevented from using incoherent lines. These lines are identified by the recovery algorithm and marked as incoherent in the cache coherence protocol state. When a processor sends a request to the home node of a line, the handler servicing the request checks the protocol state of that line. If the line is marked incoherent, the handler sends back a special reply that terminates the access by raising a bus error. The additional checks for incoherent lines do not add any latency to common handlers since we were able to insert these checks in unused MAGIC instruction slots.

3.3 Impact of Faults on the Operating System

Traditional multiprocessor operating systems cannot fully benefit from the hardware fault containment support available in FLASH, since in most cases they are unable to recover from the loss of any of their resources. To provide fault containment benefits to applications running on FLASH, we have developed the Hive operating system [3][18], a binary-compatible extension of Silicon Graphics's IRIX 5.2.

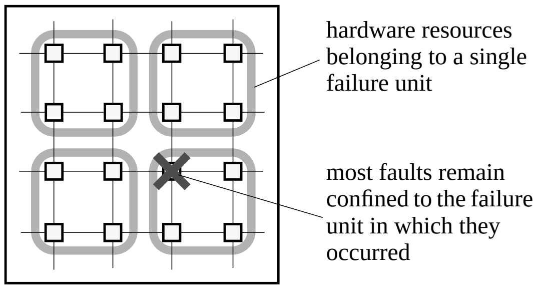

hardware resources belonging to a single failure unit
most faults remain confined to the failure unit in which they occurred

FIGURE 3.2. Partitioning the multiprocessor into Hive cells. The Hive operating system provides fault containment by partitioning the machine into failure units and running a separate kernel (a *cell*) in each unit. Cells are able to survive most hardware or system software faults occurring outside of their failure units.

Hive is structured as an internal distributed system of *cells*, as shown in Figure 3.2. Each cell is a multiprocessor operating system kernel that manages a separate partition of the machine. The cells communicate to provide the user with the usual single system image expected from a shared-memory multiprocessor operating system. Hive provides fault containment by limiting the effects of most faults to one or a few cells.

Hardware faults have several types of impacts on Hive:

- A failure that causes cache lines to become inaccessible or incoherent can cause a cell to crash if the lost data included kernel text or data structures of that cell. Similarly, the failure of a node may cause a cell to deadlock if the failed node was holding one of the cell's system locks (such as the run queue lock) at the time of the failure.
- Traffic loss as the result of a failure can damage I/O operations. If an operation such as an uncached read or write to an I/O device register is lost, it is difficult to determine whether the request has actually been serviced or not. Since those requests are in general nonidempotent, retrying them is not appropriate.
- An incorrectly speculated write may cause a processor to fetch some arbitrary line into its cache in exclusive mode. If that processor fails, the data is lost. Therefore, a node failure could destroy arbitrary data that happened to be cached exclusive on that node as a result of incorrect speculation. This effect can cause multiple cells to crash after a single hardware fault.

Limiting the impact of faults: To limit vulnerability to failures occurring in other parts of the system, each Hive cell must carefully control which resources it uses for its kernel text and data structures, and must defend those resources from outside corruption. FLASH provides the following support:

- The recovery algorithm enforces the abstraction that each partition of the machine behaves as a hardware *failure unit*. The shape of failure units is carefully chosen so that all coherence traffic inside a unit stays within the portion of the interconnect allocated to that unit. This allows the recovery algorithm to guarantee that no intra-cell memory traffic is lost as long as there are no faults in the hardware of the corresponding failure unit. If any component in a failure unit fails, the recovery algorithm stops all the nodes in that failure unit, thereby ensuring a clean shutdown of the affected cell.

  To prevent its critical data from becoming inaccessible or incoherent, a cell only stores this data in memory belonging to its own failure unit. Cells are allowed to read the kernel data structures of other cells, but not to modify them directly. If a cell needs to change the state of some other cell (for instance, when forking a process to the other cell), it makes a remote procedure call (RPC) to ask the other cell to perform the operation.

- Hive cells prevent damage caused by the loss of nonidempotent operations by avoiding uncached accesses to I/O devices across cell boundaries. If a cell needs to access an I/O device belonging to a different cell, it sends an RPC to the other cell and asks it to perform the operation. This inter-cell RPC mechanism is itself vulnerable to packet loss, but the Hive RPC subsystem uses an end-to-end protocol to guarantee exactly-once semantics.

  MAGIC supports this design by terminating with a bus error any uncached accesses to I/O devices that come from outside the local failure unit. This has the additional benefit of defending against software errors in other cells.

- MAGIC provides a hardware access control mechanism called the *firewall* that enables each cell to control precisely which of its data structures can be fetched exclusive by nodes outside the cell. The firewall associates an access control list with every 4KB memory page that specifies which nodes in the machine have write access to that page. If a node tries to fetch a line exclusive from a page to which it does not have write permission, MAGIC will issue a bus error.

  The firewall allows cells to protect their data against speculative writes, and helps defend against wild writes resulting from system software crashes in other cells [3].

- A wild write issued by the processor has the potential to disrupt the operation of that node's MAGIC, as the node controller's code, internal data structures, and cache coherence protocol state are all stored in a region of the node's memory. To defend against operating system software errors, this region is only writable by the protocol processor of the local MAGIC. The node controller prevents accidental modification using a configurable *range limit* implemented for efficiency in dedicated logic and set at protocol firmware boot time. Accesses from any R10000 processor, including the local one, that violate the range limit are terminated with a bus error by MAGIC.

4 Fault Recovery in FLASH

Many of the features described in Section 3 rely on the existence of a recovery algorithm that is executed after a fault has occurred. The purpose of this algorithm is to determine which parts of the machine have remained operational after a fault, and to reconfigure the multiprocessor so that normal operation can be resumed on the functioning parts. Table 4.1 summarizes the conditions that trigger the recovery algorithm and the tasks that it performs.

We start by giving an overview of the recovery algorithm, after which the presentation follows the phases of the algorithm shown in Figure 4.2. For completeness, at the end of the section we have included a brief discussion of operating system recovery.

4.1 Overview of the Recovery Algorithm

The recovery algorithm is divided into the four phases shown in Figure 4.2: recovery initiation, information dissemination, interconnect recovery, and recovery of the cache coherence protocol state. After the recovery algorithm completes it is possible to resume normal operation in the functioning parts of the machine; the operating system is informed that the recovery algorithm has run and performs its own recovery actions [3] before allowing user-level processes to continue execution.

TABLE 4.1. Recovery algorithm triggers and tasks.

Events that trigger hardware recovery	
memory operation timeout	Section 4.2
NAK counter overflow	Section 4.2
MAGIC firmware assertion failure	Section 4.2
reception of truncated interconnect packet	Section 4.2
Tasks performed by the recovery algorithm	
update node map	Section 4.3
implement hardware failure units	Section 4.3
recover interconnect	Section 4.4
mark incoherent cache lines	Section 4.5
resolve coherence protocol deadlock	Section 4.5

To avoid single points of failure, the recovery algorithm runs as a distributed algorithm over all the functioning nodes in the machine. The algorithm is able to cope with additional hardware failures that occur during its execution by restarting whenever a new fault is detected.

Under some pathological overload conditions, recovery might be triggered even in the absence of a hardware fault. The algorithm has been carefully designed so that normal execution can be resumed after a false alarm without any data loss. The sole effect of a false alarm is a brief interruption in the operation of the machine.

The remainder of this subsection provides some background for the presentation of the recovery phases by discussing the types of failures recognized by the recovery algorithm, the way nodes communicate during recovery, and the implementation of the algorithm.

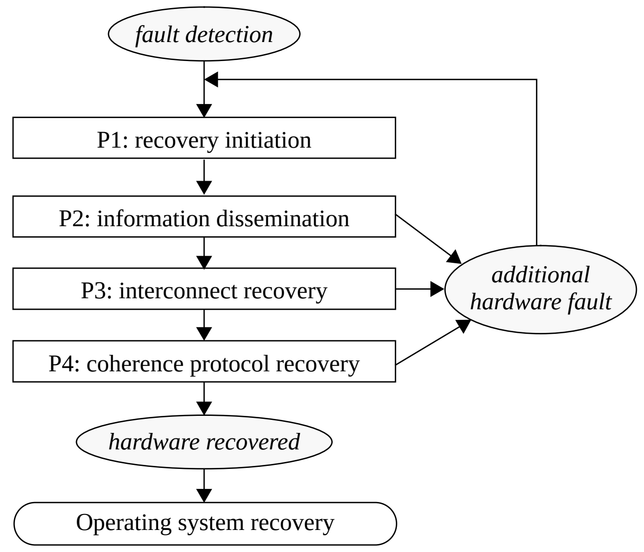
Failure types: We simplify the design of the recovery algorithm by considering only two abstract classes of failures, and by mapping all actual faults that can occur in the real system onto a combination of those two abstractions. A comprehensive diagnosis of the cause of the failure need not be done by the recovery algorithm, but can instead be performed off-line. The two abstract failure classes are node failures and link failures:

- A *node failure* affects the node's controller, processor, and memory, but not the router to which the failed node is connected. For most node failures, any packets that are sent to the failed node are discarded. The only exception is the case of a MAGIC firmware error in which a handler enters an infinite loop and causes the node controller to stop accepting packets.
- A *link failure* prevents transmission of interconnect traffic. In the CrayLink interconnect, failed links act as "black holes" by sinking any traffic that is directed through them. The recovery algorithm does not distinguish between the failure of a router and the failure of all the links connected to that router.

A hardware failure in the real machine will usually result in several link or node failures that occur almost simultaneously. For instance, a power supply loss may manifest itself as the failure of a set of routers and all of their connecting links; a disconnected cable between two cabinets appears as multiple link failures.

Inter-node communication during recovery: For the reasons discussed in Section 3.1, the recovery algorithm cannot use the normal CrayLink communication mechanisms when sending information between nodes. To avoid these problems, FLASH dedicates two virtual lanes of the interconnect for recovery traffic. When it starts, the recovery algorithm can assume that the dedicated lanes are not clogged with backed-up traffic.

Recovery packets use a source-routing mechanism provided by the CrayLink interconnect that allows the sender of a message to



fault detection
P1: recovery initiation
P2: information dissemination
P3: interconnect recovery
P4: coherence protocol recovery
additional hardware fault
hardware recovered
Operating system recovery

FIGURE 4.2. Hardware fault recovery algorithm in FLASH. After a hardware fault is detected, a distributed algorithm is executed on all the functioning nodes in the machine. The algorithm diagnoses the system, reconfigures the interconnect and resets the state of the cache coherence protocol.

specify on a link-by-link basis the exact route the message will take to its destination. This allows nodes to route messages around the failed areas. The number of hops that a source-routed packet can traverse is limited, so the initial phases of recovery use only local communication (exchange of messages between neighboring functioning nodes).

An important characteristic of the source-routing delivery mechanism is that packets do not back up indefinitely. If a source-routed packet has been stalled in a router for more than a specified time, subsequent source-routed packets sent to that router are automatically discarded. This prevents the virtual lanes used for recovery from becoming congested.

Recovery algorithm implementation: Given that the node controllers in FLASH are programmable, one could imagine implementing most of the recovery algorithm in node controller firmware. However, this would be a poor system design decision, since keeping the node controller as simple as possible helps increase the level of confidence in its correctness. Instead, most of the recovery algorithm, about 6,000 lines of C code, executes on the R10000 processors. Only the strictly necessary support was added to MAGIC firmware, contributing about 2,000 new lines to the existing 12,000 lines of node controller code.

Since the recovery algorithm temporarily suspends all normal memory system operations, it is essential that the code executed on the R10000 does not cause any coherence traffic. Therefore, for the duration of hardware recovery the processors fetch their code and data from uncached space only. In this mode, the overhead of performing a bus transaction and running a MAGIC handler for every instruction fetch slows the R10000 processor down to under 2.5 MIPS.

4.2 Recovery Initiation Phase

The first phase of the recovery algorithm is initiated when a failure is detected in the system. After starting the recovery code on the R10000, a node diagnoses the resources in its immediate vicinity and triggers recovery on the surrounding nodes.

Failure detection: As shown in Table 4.1, there are four mechanisms in FLASH that can trigger the recovery algorithm: memory operation timeouts, NAK counter overflows, firmware assertion failures, and reception of truncated packets.

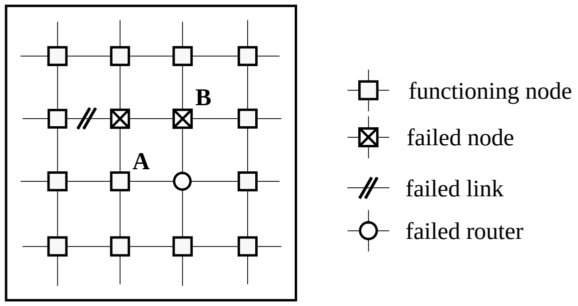


FIGURE 4.3. Failure detection in FLASH. In this example, *A* receives no reply to a memory request it has sent to *B*, since *B* has failed. Eventually, *A* times out and triggers the recovery algorithm.

Memory operation timeouts signal that the home node to which a memory request has been sent has failed, or that one of the request or reply messages has been delayed or lost. Consider the example shown in Figure 4.3. Node *A* has sent a cache coherence request to node *B*, but node *B* has failed and hence cannot satisfy the request and send back a reply. Eventually, node *A*'s controller notices that the request has timed out and triggers recovery.

The second failure detection mechanism is implemented with NAK counters. Requests to a memory line are answered with a NAK reply whenever that line is locked by some other memory operation. Each node controller maintains a hardware NAK counter that keeps track of the number of times a given memory request has been unsuccessfully retried. A *NAK counter overflow* is an indication that deadlock may have occurred as the result of a failure. The node controller implements NAK counters as dedicated logic in its processor interface. Updating the counters does not add any latency to MAGIC operation.

The third mechanism is implemented with *assertions* in the MAGIC firmware that check for firmware errors. Assertions can only be used sparingly, since it is important to avoid slowing down the normal operation of the machine. The primary use of assertions is during the debugging of the cache coherence protocol implementation.

Since the CrayLink interconnect guarantees the integrity of packets during normal operation, the *reception of a truncated packet* by a node controller indicates that a failure has occurred. The next handler that is dispatched on that MAGIC is a special error handler that triggers recovery.

Initiating recovery code on the R10000: After a failure is detected, the node controller causes the node's R10000 processor to start executing the recovery code. The initiation of the recovery code must not depend on the correctness of the system software running on that processor, given that the operating system is far too complex to be trusted to react promptly and reliably. This creates two problems. First, initiation must use a non-maskable interrupt (NMI). Not all processors provide a restartable NMI[1] (in particular, the R10000 NMI is not always restartable). Second, posting the non-maskable interrupt is not by itself sufficient for dropping the processor into recovery, as the processor may be waiting for some memory requests to complete.

Rather than using an NMI on the R10000, MAGIC forces a processor interface bus parity error which raises a Cache Error that has most of the necessary properties. This solution is not optimal,

1. An example of appropriate support is the RED_state found in the UltraSPARC architecture [22].

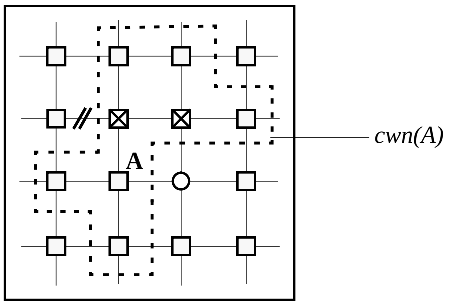


FIGURE 4.4. The set of closest working neighbors.
Every node exchanges information on the state of the system with all other functioning nodes that are closest to it. The set of closest working neighbors (*cwn*) is determined during recovery initiation.

since fielding of the exception may be delayed if another (genuine) Cache Error is being handled; we rely on the fact that the latter is a low-probability event. To make sure that the processor starts executing the recovery code immediately, any pending requests to cacheable addresses are NAK'ed by MAGIC. These requests will be reissued by the processor after recovery completes.

Outstanding uncached operations require a different treatment as they have exactly-once semantics and must not be retried. If an uncached read operation is pending at the time of recovery initiation, the recovery code running on MAGIC terminates the read by returning a NAK, but also allocates an internal buffer in MAGIC memory to save the result of the read when it arrives from the network. (As discussed in Section 3.3, a remote uncached read can only be an intra-cell one. Thus, either the read succeeds, or the failure unit corresponding to the cell goes down entirely.) Prior to resuming normal operation, the recovery code emulates the read instruction and advances the saved program counter past it.

Determining the set of closest working neighbors: After the processor has started executing the recovery code, it probes its neighboring routers and node controllers to determine which resources in its vicinity are reachable and functioning. Each node *A* determines its set of *closest working neighbors, cwn(A)*, which includes all the functioning nodes that can be reached from *A* through a path that does not contain other nodes in *cwn(A)*. An example *cwn* set is shown in Figure 4.4.

The determination of *cwn(A)* is an iterative process that probes nodes located further and further away from *A* until every path starting at *A* either ends in a failed link or contains a functioning node. At each step, *A* first probes the unexplored links by interrogating the routers at the end of those links. *A* then sends ping packets to the node controllers at the end of the unexplored links. A ping reply will only be received if the pinged node has successfully started executing the recovery code on the R10000 processor. While arbitrarily complex self-diagnostics could be run on a node before responding to a ping reply, successfully dropping the node's processor into recovery is complicated enough to provide evidence that the node is functioning correctly.

Triggering recovery on all good nodes: Typically, an anomalous condition in the operation of the machine is brought to the attention of a node by one of the mechanisms shown in Table 4.1. After a node has detected a hardware problem and started executing the recovery algorithm, it will send ping packets to all the nodes in its *cwn* set, dropping those nodes into recovery, too. A wave of ping packets spreading from each node to its immediate functioning neighbors will eventually cause all the good nodes in the system to enter recovery.

As an optimization to speed up recovery triggering, nodes speculatively send ping packets to their immediate neighbors before performing the *cwn* exploration. We have found that in FLASH this heuristic can lead to a fivefold increase in the speed at which recovery is triggered.

We assume the redundancy in the machine's interconnect is high enough that a partitioning of the system ("split-brain syndrome") is unlikely to occur. A heuristic such as shutting down the whole machine if more than a given fraction of its nodes have failed may help avoid split-brain behavior. The subsequent phases of the recovery algorithm assume that the interconnect remains connected.

4.3 Information Dissemination Phase

At the end of the initiation phase, each node *A* knows only the state of the links and nodes in its immediate vicinity, namely *cwn(A)*. The purpose of the dissemination phase is to ensure that every functioning node in the machine learns the global system state.

The dissemination phase proceeds in a number of rounds. During each round, each node *A* exchanges its current knowledge about the state of the machine with all the nodes in *cwn(A)*. As information propagates between neighboring nodes, each node gains a more complete picture of the state of the system.

```
LState = {info about links in cwn(A)}
NState = {info about nodes in cwn(A)}
round = 1
repeat
    for all n in cwn(A) do
        send (LState, NState) to n
        get (LS, NS) from n
        LState = merge(LState, LS)
        NState = merge(NState, NS)
    done
    round = round+1
until COND
```

A node gains full knowledge about the system after a number of rounds equal to the height of the BFT (breadth-first tree) rooted at that node. From this point on, `LState` and `NState` remain unchanged even if additional rounds are executed. However, the BFT height is not the same for all nodes, meaning that for different nodes, `LState` and `NState` may stabilize after a different number of rounds. To keep the nodes synchronized, we need a condition *`COND`* that terminates the above loop after the same number of rounds on all nodes, while guaranteeing that their information has stabilized.

One such loop termination condition could check that the number of rounds executed equals the maximum height of a BFT rooted at any node in the graph. This maximum height corresponds to the diameter of the interconnection graph. Note that the diameter depends on what parts of the system are working, and may change as the result of a failure. Unfortunately, the diameter is expensive to compute precisely since the best known algorithm is quadratic in the number of nodes.

As a simple approximation to the diameter, all nodes use the same algorithm to choose one of the functioning nodes, compute the height *h* of the BFT rooted at that node (in linear time), and terminate the dissemination algorithm after *2h* rounds (*2h* is an upper bound to the diameter). Still better approximations to the diameter can be computed in linear time, as shown in [1].

The BFT computations have to be carefully scheduled in order to avoid slowing down the dissemination phase. If every node computes the BFT as soon as its knowledge about the state of the system has stabilized, BFT computations on neighboring nodes will be chained during consecutive rounds instead of proceeding in parallel. To avoid the serialization of those computations, during the dissemination phase we only compute the BFT on a few nodes

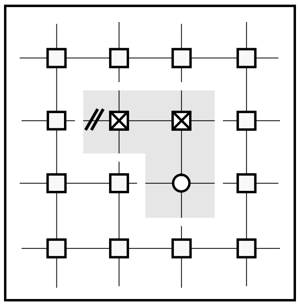


FIGURE 4.5. Interconnect recovery.
Each faulty region is isolated by discarding any traffic that tries to enter the region. After the interconnect has drained, the routing tables are reprogrammed such that normal coherence traffic will be routed around the failed regions.

for which `LState` and `NState` stabilize first. Those nodes send the resulting diameter estimation as a hint to their neighbors during subsequent rounds. Nodes that receive a hint defer their BFT computation until the end of the dissemination phase, when all the deferred computations occur in parallel.

Nodes use the knowledge gained during the information dissemination phase to update the node maps of their individual node controllers. If a node finds out that it is part of a failure unit in which some other resources have failed, the node shuts itself down after the cache coherence protocol recovery phase completes.

4.4 Interconnect Recovery Phase

The common knowledge about the state of the system gained during the dissemination phase allows the nodes to recover the interconnect with only minimal communication between them. Recovering the interconnect (Figure 4.5) involves three steps: isolating the failed regions, draining any stalled interconnect traffic, and reprogramming the routing tables.

In the first step all the routers bordering the failed parts of the system are reprogrammed to discard coherence traffic that is stalled trying to enter the failed region. As a side effect of discarding these packets, other stalled traffic makes forward progress and eventually reaches its destination. During this step the node controllers keep fielding coherence messages until the interconnect has drained. To avoid generating new traffic, the node controllers switch to a mode where incoming requests do not generate the replies or invalidates that would normally be sent.

To determine when the interconnect has drained, we rely on the existence of a small upper bound τ for the delay between two consecutive deliveries of stalled packets (the packets do not have to be delivered to the same node). The nodes go through a two-phase agreement protocol. During the first phase, all the notes vote to proceed after they have not seen any delivery of stalled traffic to their node controllers for at least τ time units. In the second phase, nodes can either vote to proceed, if they still haven't seen any delivery of stalled traffic, or they can request that the agreement protocol be restarted, if any delivery of stalled traffic took place since their first vote. The two phases are implemented as (fault-tolerant) barriers using the BFT computed during dissemination [6]. In practice, we have not observed any case in which the agreement protocol did not commit in the first round.

After all the traffic has drained out of the interconnect, it is safe to reprogram the routing tables so that new traffic will be routed around the failed regions. Stalled traffic must be drained first since otherwise we may introduce loops and cause traffic deadlock while reprogramming the routers. When computing the new routing tables, the key problem is to avoid introducing potential loops that

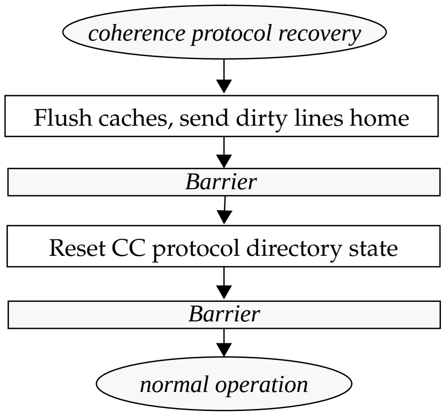

coherence protocol recovery
Flush caches, send dirty lines home
Barrier
Reset CC protocol directory state
Barrier
normal operation

FIGURE 4.6. Cache coherence protocol recovery.
To determine what cache lines have become incoherent, all the functioning nodes flush their processor caches and send dirty lines back home. After the flush completes, each node examines the state of the cache lines it owns. Any line that hasn't made it home is marked as incoherent.

could lead to deadlock. We use the turn method and an approach similar to the one described in [5] and [21] to guarantee that the routing is deadlock free. Although this approach is able to produce a deadlock-free rerouting for many of the common failure cases, a general solution is still an open research issue; the details are outside the scope of this paper.

The reconfiguration phase ends with a global barrier that guarantees that all routers have been reprogrammed before any new coherence traffic is injected into the interconnect. After the barrier, the interconnect is again ready to carry normal coherence traffic; this is essential for the last phase of the recovery algorithm, during which communication between nodes is not local.

4.5 Cache Coherence Protocol Recovery Phase

The purpose of the final recovery step (Figure 4.6) is to reset the state of the cache coherence protocol and to determine which cache lines have become incoherent. Since there is no *a priori* way to know which coherence messages have been lost, the system first returns all dirty remote data to its home node. This is implemented by having all nodes flush their processor caches and send all dirty cache lines back to their homes. (It is not necessary to send any coherence message to the home for lines that are not dirty, since for those lines the copy at the home is valid.) After flushing their caches, the nodes join a global barrier. This barrier is implemented as an all-to-all send and relies on the in-order delivery of packets provided by the interconnect to guarantee that all writebacks have made it to their destinations.

Following the barrier, each node resets the state of its cache coherence protocol data structures. To determine which cache lines have been lost, each node scans its protocol directory state. Any line that still appears to be cached exclusive must have been lost. Incoherent lines are marked so that future accesses to those lines will be bus-errored by MAGIC. For all other lines, the protocol state is reset to "clean and not cached" since after the cache flush step all processor caches in the system are empty.

After the reset of the directory state, the nodes join another global barrier. At this point, the machine's interconnect and shared memory system are ready to resume normal operation, and the node controllers will again dispatch the normal handlers to service coherence requests.

4.6 Operating System Recovery

Prior to resuming normal operation, each node controller raises an interrupt to inform the operating system that hardware recovery has taken place in the machine. This interrupt causes Hive to activate its own recovery mechanisms before allowing user-level processes to continue execution.

The functioning cells adjust their internal kernel data structures to reflect the new machine configuration and to preserve the single system image semantics. User applications that had any essential dependencies on the failed cells are terminated, while other applications are allowed to continue unaffected. Normal system operation resumes after operating system recovery has completed.

The operating system must be able to cope with incoherent lines, since any access to those lines will give rise to a bus error. After Hive removes all the mappings to a page, it uses a special MAGIC service to check whether the page contains any incoherent lines and to reset the state of the cache coherence protocol for those lines before reusing the page.

5 Experimental Results

In this section we report experimental results for our hardware fault containment implementation in FLASH. We start by describing the simulation environment used for our experiments. Next we present the results of 1000 validation runs that we performed in order to verify the correctness of our implementation. To demonstrate that our solution provides end-to-end fault containment, we ran an additional batch of experiments in which we injected a fault during the execution of a workload running on top of the Hive operating system. We conclude this section with scalability results.

5.1 Simulation Environment

We debugged and tested our implementation of fault recovery using the SimOS [19] and FlashLite [7] simulators, since the FLASH hardware is not available at the time of writing. FlashLite is an accurate model of the FLASH memory system that models in detail both the node controllers and the CrayLink interconnect.

In the experiments reported below, we used an R4000 processor model, rather than the R10000 simulator which is significantly slower. The simpler model does not expose any of the effects related to incorrect speculation, but should otherwise demonstrate all the effects seen on the real machine.

The machine configurations used in our experiments are shown in Table 5.1. For the end-to-end recovery experiments we booted the Hive operating system [3] in an 8-cell configuration with one processor and 16 MB of main memory per cell. We ran a parallel make benchmark that compiles eight of the GnuChess 4.0 files, with each compile executing on a different cell. The benchmark generates a large amount of coherence traffic, since one of the cells acts as a file server for the other cells and the Hive file system uses shared memory for all file data transfers across cell boundaries.

Table 5.2 shows the types of faults we have simulated in our experiments. All the faults shown in this table correspond to possible occurrences on a real system.

5.2 Experimental Validation

In the first batch of validation experiments we did not boot an operating system, but used a stand-alone test program to create a controlled sharing pattern before injecting a fault. In each run, all the processors start by filling up their caches with lines chosen at random from the range of valid system addresses. For each line, we randomly decide whether it will be fetched in shared or exclusive mode. After all the processors have filled up at least half of their caches, we inject a fault. Upon completion of the hardware recovery algorithm, the processors read all of the system's memory and check, for each cache line, whether it contains the correct data or has become incoherent. We keep track in the simulator of the lines that may have become incoherent, either because they were cached on a failed node or because they were in a transitional state when we injected the fault. This allows us to verify that our recovery algorithm does not mark more lines as incoherent than necessary. The results of the validation experiments are shown in Table 5.3. We did not observe anomalies in any experiment.

End-to-end recovery experiments: For the end-to-end recovery experiments we injected the types of faults shown in Table 5.2 into an 8-processor system running a parallel make workload on top of the Hive operating system. We performed the experiments listed in Table 5.4, injecting the faults while all of the eight compiles were running, and checked the results of the completed compiles. In all experiments, the system correctly ran the hardware recovery algorithm and underwent operating system recovery. 91.6% of the

TABLE 5.1. Simulated hardware configurations for the fault injection experiments.

Component	Characteristics
Processors	$8 \times$ MIPS R4000 @ 200 MHz
Node controllers	$8 \times$ MAGIC @ 100 MHz
Memory / node	1-16 MB
L2 cache size	1 MB

TABLE 5.2. Fault injection experiment types.

Fault type	Description
Node failure	MAGIC fails, but the router stays up. Any packets sent to the node controller are discarded.
Router failure	Any packets sent to the router are discarded.
Link failure	Packets that try to traverse the link are dropped.
Infinite loop	MAGIC stops accepting packets. Traffic directed to this node backs up into the interconnect.
False alarm	Recovery triggered by an exceptional overload condition in the absence of a fault.

TABLE 5.3. Validation experiments.

Injected fault type	# of experiments	# of failed experiments
Node failure	200	0
Router failure	200	0
Link failure	200	0
Infinite loop in MAGIC handler	200	0
Recovery triggered by false alarm	200	0

TABLE 5.4. End-to-end recovery experiments.

Injected fault type	# of experiments	# of failed experiments
Node failure	310	29
Router failure	215	20
Link failure	268	22
Infinite loop in MAGIC handler	394	28
Total	1187	99

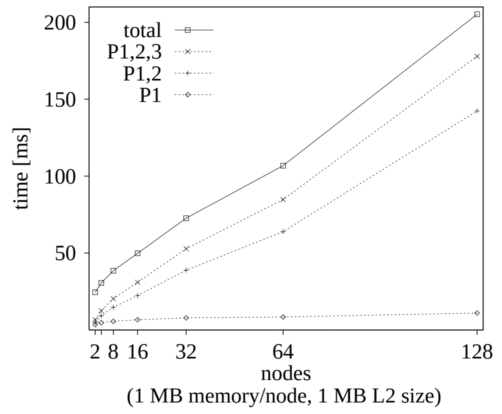


FIGURE 5.5. Total hardware recovery times.
The times shown for the four phases of the recovery algorithm are for a mesh configuration. The time required for the information dissemination phase (*P2*) grows slower for a hypercube topology.

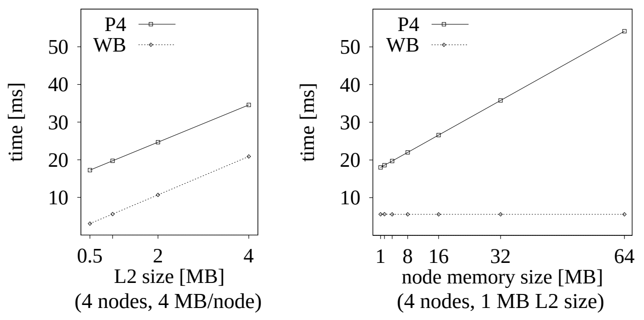


FIGURE 5.6. Cache coherence protocol recovery times.
In the last phase (*P4*) of the hardware recovery algorithm, the processors flush their caches (*WB*) and then reset the state of the cache coherence protocol. Assuming no interconnect contention during the flush, these steps scale linearly with the second level cache size and the amount of memory per node, respectively.

runs correctly finished executing the compiles that were not affected by the fault. Investigation of the failed runs shows that the problems are caused by a few operating system bugs rather than incorrect hardware recovery. All the OS bugs are related to the presence of incoherent lines after a fault; these cases were not exercised in our previous experiments [3][18], which were performed using a less detailed simulator.

5.3 Scalability Results

The time required for the hardware recovery algorithm to complete depends on the number of processors in the system, the second level cache size, and the amount of memory per node.

Figure 5.5 shows how the hardware recovery time scales with the number of processors. For large systems, hardware recovery is dominated by the time required for the dissemination phase. This phase scales better (both asymptotically and for a moderate number of nodes) on the fat hypercube topology used in FLASH than on the mesh interconnect we have simulated in these experiments, since its running time is proportional to the diameter of the interconnect.

Both components of the last recovery phase (flushing the caches and recovering the state of the cache coherence protocol) scale linearly with the size of the second level cache and the amount of memory per node, respectively (see Figure 5.6). On large

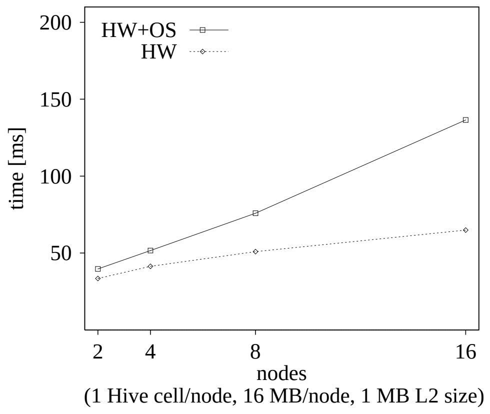


FIGURE 5.7. End-to-end recovery times.
Applications that continue running after a fault will be suspended for the duration of the hardware recovery algorithm (*HW*) and of the Hive operating system recovery phase (*OS*).

machines, the cache flush time may be higher if the flood of writebacks leads to interconnect contention or memory hot spots.

The duration for which user processes are suspended after a hardware fault is shown in Figure 5.7. These times include both hardware and operating system recovery, and should be acceptably fast for most applications. Operating system recovery time should be significantly faster on large machines than this graph suggests, since it scales with the number of cells rather than the number of nodes and large systems are likely to run with multiple nodes per cell. Furthermore, we have identified several ways to reduce Hive recovery time.

To gauge the precision of the above scalability figures, we simulated a few dozen R10000 instruction fetches from uncached space using a cycle-accurate RTL model of the R10000 processor and a Verilog model of the MAGIC chip. The only significant deviation we observed was in the time required for fetching and executing an uncached instruction on the R10000. In the precise simulation this time is 390 ns, while SimOS/FlashLite report only 320 ns. The discrepancy is primarily due to the fact that the R4000 simulator used in our experiments does not accurately simulate the execution pipeline and the bus interface of the R10000. The overall impact of this imprecision on our scalability results is small, since a significant fraction of the recovery algorithm is spent executing protocol processor code for which the time required is accurately simulated by FlashLite.

6 Discussion

We begin this section by discussing some low-level hardware design considerations for providing fault containment. Next we evaluate the performance impact of fault containment on normal operation, and address the issues related to implementing fault containment on a machine with hardwired node controllers. We conclude with an overview of related work.

6.1 Low-Level Support for Fault Containment

To make effective fault containment possible, the hardware of the machine must be designed to provide the following three properties: no single point of failure, *fail-fast* failure behavior, and *monotonic* failures. Components are fail-fast if after a fault they stop working without generating erroneous outputs. Components fail monotonically if they remain halted until repaired or reset.

The requirement concerning the absence of a single point of failure is obvious, and is the main reason that fault containment cannot be easily provided in bus-based shared-memory multiprocessors. Fortunately, scalable shared-memory systems have a structure that lends itself naturally to this design goal.

The fail-fast and monotonic failure properties are important since it is in general impossible to design a recovery algorithm that can cope with arbitrary Byzantine failures. In practice, however, neither the fail-fast nor the monotonic failure property can be guaranteed in all cases. Occasionally, components can be expected to generate bad outputs (due to a firmware error, for instance) or to suffer from intermittent faults (such as a wiring problem). In these cases our fault-containment approach may break down and the whole machine may fail. This is acceptable as long as the probability of a such a fault is small enough to have no significant impact on the overall reliability of the machine. Careful design of the hardware and of the node controller firmware can minimize this probability.

6.2 Costs and Benefits of Flexibility

Table 6.1 summarizes the hardware features that support fault containment in FLASH. We believe that similar mechanisms will be required to provide fault containment in other scalable multiprocessor architectures. This is an area for future research, as only an explicit design of a fault containment solution for an architecture will bring out all the significant details.

It is simple to evaluate the performance cost of most node controller features that were added to FLASH. The node map, truncated message dispatch, remap, range check, memory operation timeout, and NAK counter mechanisms are implemented in dedicated logic in the MAGIC interfaces or dispatch mechanism. Careful design ensures that these features do not increase the number of MAGIC clock cycles required to dispatch and execute handlers during normal operation.

The firewall mechanism is more intrusive, since our implementation requires adding access permission checks to the handlers that service intercell writes. Detailed system simulations show that the average increase in intercell write cache miss latency due to the firewall is less than 7% of the fastest internode write cache miss [3]. Since intercell write cache misses make up a small fraction of the cache coherence operations and since the R10000 processors provide latency-hiding mechanisms, the degradation in application performance lies in the noise for most workloads.

An important issue for fault containment solutions on other scalable multiprocessor architectures is the flexibility of the node controller. Unlike MAGIC, node controllers in most current commercial scalable multiprocessors are hardwired, not programmable. Whether or not it is possible to implement fault containment in a machine with hardwired node controllers without impacting performance is a topic for future research. However, our experience with FLASH provides insights for such a development.

The main difference between hardwired and programmable controllers, as far as fault containment is concerned, lies in the support they can offer for the recovery algorithm. Design complexity limits a hardwired controller to a significantly lower level of support than is implemented in MAGIC. The minimum support required is: (1) mechanisms for detecting hardware failures, (2) mechanisms for unstalling the processor and initiating recovery, and (3) a mode in which the processor performs all node controller functions except for processing the uncached reads and writes needed to access the recovery code and data. This solution requires exposing most of the state of the node controller to allow the processor to take control of its operation during recovery.

6.3 Related Work

An early effort at recovering from failures in the C.mmp multiprocessor is described in [24]. C.mmp attempted to diagnose the system after a fault and excluded any failed resources before resuming the unaffected processes. However, C.mmp did not have to deal with most of the issues described in this paper, since it did not use caches and since it had a crossbar interconnect. Furthermore, due to an *ad-hoc* design of the error recovery component, C.mmp was not able to handle many errors.

TABLE 6.1. Summary of features used for hardware fault containment in FLASH.

Node controller	
node map	Section 3.1
handling truncated messages	Section 3.1
remap exception vectors	Section 3.2
firewall	Section 3.3
range check	Section 3.3
memory operation timeouts	Section 4.2
NAK counter overflow	Section 4.2
assertions on protocol state	Section 4.2
Routers and interconnect	
reprogrammable routing tables	Section 3.1
detection of truncated messages	Section 3.1
dedicated virtual lanes for recovery	Section 4.1
source-routed message option	Section 4.1
Processor	
non-maskable restartable interrupt	Section 4.2

The Sequoia shared-memory multiprocessor [2] provided fault tolerance by using a checkpointing scheme. In this scheme, main memory stores two copies of all application data. Any writeback requires the entire cache to be flushed twice (under OS control), once to each memory copy. For a high-performance system like FLASH, such an approach would come at a very high cost, and would run contrary to our driving goal of leaving the normal mode performance of the system intact.

An approach for fault tolerance in COMA (Cache Only Memory Architecture) systems has been proposed in [13], but has not been implemented. The foremost drawback of this scheme is the high overhead for maintaining copies of memory lines. A further difficulty consists in determining whether non-idempotent operations (such as uncached I/O operations) have completed before a fault since rolling back to a previously saved state and reissuing those operations may violate their exactly-once semantics. The approach described in [13] can only cope with the failure of one processor, requires a spare processor to resume operation, and cannot handle interconnect failures.

A general survey of checkpointing techniques for distributed shared memory can be found in [14].

Fault containment has been implemented in several network operating systems such as Locus [16], Sprite [17], and Solaris MC [8]. Due to a much weaker coupling between nodes and to the absence of shared memory, these systems do not have to cope with the problems described in Section 3.1 and Section 3.2. Node failures are detected in a lazy fashion, while transient network problems are handled by an end-to-end transmission protocol.

The HAL multiprocessor provides an efficient hardware implementation of an end-to-end reliable protocol for coherence traffic [23]. The advantage of end-to-end reliability for fault containment consists in eliminating the loss of coherence packets that try to traverse a failed area, since these packets are resent by the hardware after the connectivity of the interconnect is restored. A recovery algorithm similar to the one described in this paper could be used for the HAL system. With a reliable interconnect, the cache flush step could be eliminated, but the directories would still have to be scanned and their state updated to reflect the loss of memory lines cached either shared or exclusive in the failed portion of the machine.

7 Conclusions

Fault containment has been used in many distributed systems, but to our knowledge has not been previously implemented in a high-performance shared-memory multiprocessor. While providing weaker guarantees than fault tolerance, fault containment avoids the expensive duplication of resources required by fault tolerance.

In this paper we have shown that fault containment techniques can be successfully applied to scalable shared-memory multiprocessors. We have added fault containment support to the FLASH multiprocessor without significantly affecting its performance, at least as measured through detailed simulation.

Our approach to providing fault containment in FLASH includes implementing a set of features that limit the impact of faults and a distributed recovery algorithm that is executed after a fault to restore the normal operation of the system. The experimental results reported in the paper indicate that this approach is effective and leads to acceptably short failure recovery times for systems up to 128 nodes.

Adding support for fault containment in FLASH was made easier by the existence of a programmable node controller whose behavior can be easily extended without hardware modifications. Much of the necessary support was added in the form of node controller software extensions. This approach was essential to our design process, since it allowed us to incrementally add functionality to the recovery algorithm without having to change the design of the MAGIC hardware.

The implementation of fault containment in FLASH should offer valuable information to those considering adding similar support to other scalable multiprocessors. Hardware fault containment provides an attractive way to increase the reliability and usability of these machines.

Acknowledgments

Rajeev Motwani contributed to the design of the second phase of the recovery algorithm. Mark Heinrich implemented the simulator for the CrayLink interconnect used in FlashLite. We are grateful to John Heinlein for running the detailed R10000 RTL model and MAGIC Verilog simulation. John Keen kindly provided feedback on an earlier draft of this paper. We thank Silicon Graphics for giving us access to the IRIX 5.2 source code.

This work was supported in part by ARPA contract DABT63-94-C-0054. Mendel Rosenblum is partially supported by a National Science Foundation Young Investigator award.

References

[1] D. Aingworth, C. Chekuri, and R. Motwani. “Fast Estimation of Diameter and Shortest Paths (without Matrix Multiplication).” In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 547-553, 1996.

[2] P. Bernstein. “Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing.” *IEEE Computer* 21(2), February 1988.

[3] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. “Hive: Fault Containment for Shared-Memory Multiprocessors.” In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 12-25, December 1995.

[4] M. Galles. “Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip.” Presented at *Hot Interconnects Symposium IV*, August 1996.

[5] C. Glass and L. Ni. “Fault-Tolerant Wormhole Routing in Meshes without Virtual Channels.” *IEEE Transactions on Parallel and Distributed Systems* 7(6), pp. 620-636, June 1996.

[6] J. R. Goodman, M. K. Vernon, and P. J. Woest. “Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors.” In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64-73, April 1989.

[7] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor.” In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 274-285, October 1994.

[8] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani. “Solaris MC: A Multi Computer OS.” In *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 191-204, January 1996.

[9] 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.

[10] J. Laudon, and D. Lenoski. “The SGI Origin 2000: A ccNUMA Highly Scalable Server.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

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

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1995.

[13] C. Morin, A. Gefflaut, M. Banatre, and A. Kernarrec. “COMA: an Opportunity for Building Fault-tolerant Scalable Shared Memory Multiprocessors.” In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 56-65, May 1996.

[14] C. Morin, and I. Puaut. “A Survey of Recoverable Distributed Shared Memory Systems.” IRISA Technical Report 975, December 1995.

[15] V. P. Nelson. “Fault-tolerant computing: fundamental concepts.” *IEEE Computer* 23(7), pp. 19-25, July 1990.

[16] G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[17] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. “The Sprite network operating system.” *IEEE Computer* 21(2), pp. 23-36, February 1988.

[18] M. Rosenblum, J. Chapin, D. Teodosiu, S. Devine, T. Lahiri, and A. Gupta. “Implementing Efficient Fault Containment for Multiprocessors.” *Communications of the ACM* 39(9), September 1996, pp. 52-61.

[19] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. “Complete Computer Simulation: The SimOS Approach.” *IEEE Parallel and Distributed Technology* 3(4), pp. 34-43, Winter 1995.

[20] T. Sterling, P. Merkey, and D. Savarese. “Improving Application Performance on the HP/Convex Exemplar.” *IEEE Computer* 29(12), pp. 50-55, December 1996.

[21] J. Vounckx et al. “Fault-Tolerant Compact Routing based on Reduced Structural Information in Wormhole-Switching based Networks.” In *Proceedings of the Colloquium on Structural Information and Communication Complexity*, May 1994.

[22] D. Weaver and T. Germond, eds. *The SPARC Architecture Manual, Version 9*. Prentice-Hall, Inc., 1994.

[23] W. Weber et al. “The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-performance Servers.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[24] W. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[25] K. Yeager. “The MIPS R10000 Superscalar Microprocessor.” *IEEE Micro* 16(2), pp. 28-41, April 1996.