Communication Across Fault-Containment Firewalls on the SGI Origin*

Kaushik Ghosh and Allan J. Christie
Silicon Graphics Inc.
Mountain View, CA 94043
(reality.sgi.com/[kaushik/ajc])

Abstract

Scalability and reliability are inseparable in high-performance computing. Fault-isolation through hardware is a popular means of providing reliability. Unfortunately, such isolation also increases communication latencies: typically, one has to drop into and out of the kernel to communicate between failure domains. On the other hand, relaxing fault isolation domains allows efficient communication, but at the risk of failure propagation, and thus reduced reliability. We are concerned with finding a middle ground between these extremes.

We first review a few salient aspects of the SGI Origin-2000 architecture, mentioning the hardware features germane to efficient communication, and building protection-firewalls. Then, we describe a mechanism for risk-free, point-to-point communication between processes on distinct failure domains. Quoting performance numbers, we show that the overheads of crossing domains render this mechanism unattractive for small messages. To address this issue, we describe a mechanism for controlled opening of the firewalls, thereby achieving explicit inter-partition shared-memory for communication. We describe the kernel software that addresses the resulting reliability issues, and discuss how familiar IPC mechanisms such as MPI and SysV shared-memory can use the explicit shared-memory to advantage. Finally, based on the lessons learnt, we discuss some future directions, and draw concluding remarks.

1 Introduction

Shared-memory multiprocessors have been becoming increasingly popular due to the ease of programming such machines, as compared to message-passing multicomputers. However, there are two basic problems associated with shared-memory machines: firstly, from a performance viewpoint, it is difficult to scale an operating system beyond a few dozen processors; secondly, from a machine-reliability viewpoint, with the addition of thousands of processors, memory and other hardware/software components, the mean time between failures often reduces to unacceptable limits.

To increase reliability, it is possible to have some fault-containment mechanism in hardware, divide the machine into fault-containing partitions, and run a distinct copy of the operating system kernel in each partition. Usually, such mechanisms are provided through the so-called *firewall* registers. Proper permissions must be set on a firewall register in order for a CPU on one firewall-partition (henceforth, *partition*) to access specific resources on a different partition. The Origin-2000 has provision for such division of a machine into distinct partitions. Thus, a 16-module (128-processor) machine, e.g., might be broken up into 4 partitions of 4 modules (32 processors) each. Each partition behaves as a stand-alone host with its own operating system, console, hostname and IP address. The different processors in a partition, as well as the partitions themselves, are connected by CrayLink[6, 10] – SGI's proprietary interconnect through which coherence and other communication traffic is routed.

The above approach takes care of fault-containment: hardware and/or software faults can panic any kernel. However, since system resets do not propagate across firewalls, other kernels are unaffected. Breaking up a shared-memory machine into multiple autonomous entities can also cater to many of the scalability issues. Most current operating systems don't scale well beyond a few dozen processors. Thus, restricting most accesses to 'local' resources is in general a good strategy.

However, in scaling up, an application might need to span multiple partitions (to access the proper number of processors, amount of memory etc.). Therefore, we need to devise ways to allow the 'threads' of an application to efficiently communicate with their counterparts across a partition boundary. That is what we are concerned with in this paper. The objective of this work is more to provide an efficient communication layer which can be used by libraries such as MPI, or SysV shared memory routines, rather than to provide a user-visible API.

In the first part of this paper, we describe a communication mechanism which does not need to lower the firewalls between partitions. In this respect, the mechanism has attractive safety properties. It uses the hardware firewalls to ensure that

*The authors are solely responsible for the ideas and opinions expressed herein.

faults are unable to propagate from one partition to another, and thus is 'risk-free'. However, the latency of communication with this mechanism is rather high. Precluding the possibility of the propagation of faults by the use of firewalls is attractive, if low communication latencies can also be provided. If not, it might be worthwhile to allow selective opening of firewalls – thereby allowing the propagation of faults – and detect and recover from said faults if and when they occur. Such opening of firewalls, the relevant error-detection/recovery, and resulting low-latency communication is what we will be concerned with in the second part of the paper. We address fail-stop of partitions in this work, rather than more general types of failure – such as transient or Byzantine failures.

The remainder of the paper is organized as follows. In Section 2, we describe the relevant features of the Origin-2000 architecture. Then, in Section 3, we describe the implementation and performance of the risk-free communication mechanism. In Section 4, we point out the drawbacks of the risk-free mechanism, describe a way to handle the drawbacks through inter-partition shared-memory, address the resulting error-handling issues, and report performance. Further in Section 5 we mention how familiar interfaces such as MPI and SysV shared-memory are able to gain from the shared-memory framework. In Section 6, we state our future plans, and in Section 7 mention related work. Conclusions are drawn in Section 8.

2 Origin-2000 Architecture

The Origin-2000 is a cache-coherent, distributed shared-memory NUMA machine consisting of several nodes connected with a scalable interconnect called CrayLink (see Figure 1, and[10]). Each node supports one to two MIPS R10000 CPUs, a memory subsystem and an I/O interface, all of which are connected to a node-controller, or *hub*. The caches use a writeback protocol, and a directory-based, nonblocking protocol with reply-forwarding is used for coherence. There is hardware support for each CPU to interrupt any CPU on a different node in the machine. This is termed the cross-call interrupt. A *region* of an Origin-2000 corresponds to a node if the system is small, or to a set of nodes if the system is large. The receiving processor of a cross-call interrupt can identify the source region of the interrupt.

A part of the physical address space on an Origin-2000 has backdoor access to the page-permission entries. By writing the appropriate permission bits, a processor allows access to any 4Kbyte physical page of the local memory on that node to any set of remote nodes. Another special part of the address space consists of the *region mask*, which states which regions a processor can generate an access to. The region mask entry corresponding to a region can be written only by the owning partition. By default, cross-call interrupts are enabled for all regions. An interrupt from an 'unknown' region triggers software for detection of new partitions.

Each node has a device for asynchronous, high-bandwidth copying of blocks of data, called the Block Transfer Engine (BTE). The BTE can be thought of as a high-speed DMA device; it takes physical addresses, and is available only to the kernel (user-applications have to trap to the kernel to use it).

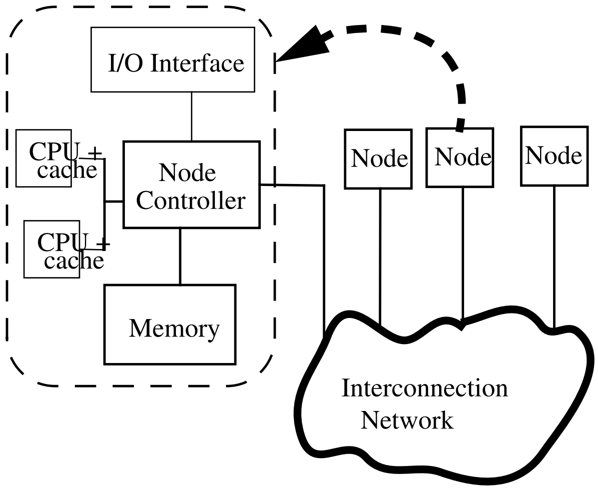

Figure 1: The Origin-2000 Architecture.

In its current implementation, the BTE transfers one or more cache-lines of cache-line aligned data. The BTE operations are fully cache-coherent, but do not pollute any processor's cache (i.e., the sending node's directory-entries do not end up pointing to the receiving node). Further, when the BTE is used to 'pull' bits from a remote partition, the firewalls on the source partition do not have to be lowered. The BTE operations are pipelined; when a BTE operation fails, the error address, type and transaction-id are logged in the hub. Processors can inspect these logs for error status. Each of the two processors on a node has its own set of BTE registers, which control attributes such as the source/destination addresses and length, address to notify upon completion of copy, etc.. However, the two processors share the same physical BTE. Transfer bandwidth is highest when the source address of the BTE-copy is on a different node than the destination: over 500 MBytes/sec of transfer band width for appropriately long messages. Note that this is much higher than what is achievable using the R10000 as such (i.e., via bcopy).

Finally, a few words about the processor: the MIPS R10000 is a 64-bit superscalar RISC processor. A feature of the R10000 is that it uses speculative execution[11]. This is important for boosting performance in the face of high latency loads. However, speculation complicates instruction graduation, and exception handling. For the purpose of this paper, the chief relevance of speculation is that any processor in the system can (asynchronously) generate references to any physical address known to the system. This has serious implications in error-handling, as will become clear in Section 4.2.

3 Point-to-point Communication

In this section, we describe the architecture and performance of a risk-free, point-to-point communication facility based on the BTE. It affords complete isolation of faults by ensuring that firewalls need not be opened to allow communication. Our overriding philosophy in building the point-to-point communication facility was fault-isolation: viz., the failure of one partition should not cause a different partition to panic. Thus, while this communication facility has a few overheads due to operating system intervention, it uses the BTE to safely copy messages from the source partition to the destination, and is guaranteed to not fatally affect any partition due to the failure of any other partition. At worst, a read, write or select operation will return a bad status when such communication happens to a failing or failed partition.

3.1 Control Messages

The key structures of the point-to-point communication device are some ring buffers. Each partition has a control ring for every other partition in the system. Additionally, each of the two endpoints of an inter-partition communication channel has a ring buffer associated with it (see Figure 2). These ring buffers instruct the local BTE about the source-address and length of the data item that has to be copied across from a remote partition. If the data item is small enough to fit in a ring-entry, the actual data is placed in the ring-buffer entry, rather than a pointer to the data.

Each ring has a consume, a produce, and a cached-produce pointer. The produce and consume pointers of all rings are consecutive in memory, so that a single DMA transfer can retrieve the produce/consume pointers of all the rings. The produce pointer points past the latest valid entry in the local ring buffer. When the sender of a message writes a new item in a ring buffer, it updates the produce pointer. The consume pointer points across a partition firewall to the last entry which was obtained from the remote end of the connection. The cached-produce pointer corresponds to a locally-cached value of the produce pointer for the remote end of that control ring. In what follows we shall see why our copying strategies are safe despite having inter-partition pointers.

For expository purposes, let us consider two partitions A and B. Let's also assume that partition A happens to boot up first, then partition B.

The control rings are set up when a partition boots. Each partition allocates a control ring and puts the address of the start of its control ring in a well-known kldir location, defined as follows. The software architecture of the operating system defines a 'directory' of values for each node: the so-called kldir area – which resides at a fixed place in every partition's memory, and can provide pointers to other memory areas as required. Any partition can read a different partition's kldir area through uncached accesses: for our purposes, this means that a different partition can find out where A's control ring resides. Note, however, that the allocated ring buffers themselves are in cached space.

The kernel on A opens the control ring for every other partition (B in this case), which results in sending a cross-call interrupt to the remote end of every control ring. Normally, the interrupt is fielded by the processor on which the *open()* was performed on the destination side of the ring, but the underlying protocol of the point-to-point transfer allows a target processor to be specified. If the cross-call interrupt packet is lost, the open from A fails.

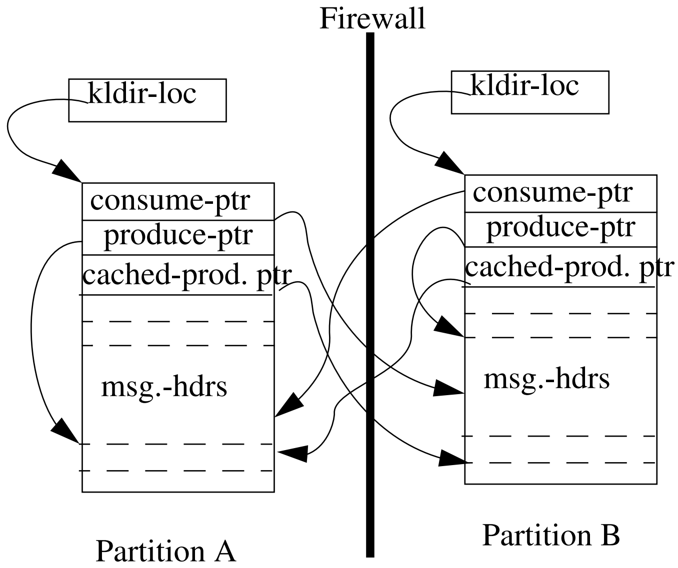

Figure 2: Structure of the control rings.

If the remote partition is up and running, it responds to the interrupt; if it is not physically reachable (power/interconnection failure), a timeout occurs, and the interrupting partition knows that the remote end is not available yet. If B is not up and running yet, A sets a kernel flag denoting that there's a pending open on control ring of B, and returns. When B boots up, it allocates its own control ring, and puts the address of its ring in its well-known kldir location. Thereafter, B opens the control ring to A – sending a cross-call interrupt to A. Since we assume here that A is already up, this interrupt will go through. On getting this cross-call interrupt, A marks the pending open on B to be complete, and sends a cross-call interrupt to B. On receiving this interrupt, B marks its open to be complete and returns. At this point, both A and B know that they are connected via a control ring, and where the other partition's control ring is situated (through the location in kldir space, which any partition can read).

Message sends over the control channel take place in the following manner. Control messages are small enough to be 'immediate': that is, the body of the message is sent together with the header. The sending partition writes the message header and contents of the control-ring buffer entry corresponding to its produce pointer, and sends a cross-call interrupt to the destination partition of the message[1] The link-layer protocol (LLP) hardware on CrayLink ensures that truncated packets are re-sent. However, if the cross-call interrupt packet is *lost*, the message is essentially dropped. If the interrupt is received, the destination partition uses the BTE to retrieve the sender's produce pointer, and stores this pointer in its own cached-produce pointer. If there is a pending read on the control ring, and the consume pointer and cached-produce pointer

[1] The cross-call interrupt does not carry data arguments; thus, all channels opened from a particular processor have to be inspected for incoming messages on a cross-call interrupt. However, the number of channels is typically low enough that this isn't a bottleneck.

are different, there is at least one message that can be received. Under these circumstances, the destination uses the BTE to retrieve all the items between the consume pointer and the cached-produce pointer from the sender's control ring, updates its consume pointer to equal the cached-produce pointer and returns the read data. If there is no pending read, the consume pointer is not updated, nor the message retrieved from the sending partition; both of these activities will be performed when the read actually occurs.

Note that all retrieval of pointers and message headers/body from a remote partition is done through copying over by the BTE. As stated in Section 2, if the source of a BTE-copy fails, the BTE returns an error. Hence, checking for the proper status at the end of a BTE-copy is enough to ensure that failure of the sender of a message does not cause the kernel on the receiver to fail.

3.2 Regular Messages

Once the control rings are set up, a user process can open a point-to-point connection to any other user process on a different partition. When the user opens a channel between partitions A and B from partition A (say), the kernel on A allocates a local ring buffer for the channel, marks the status of the open as 'pending', and sends a message to B over the control ring. Among the contents of this message are the channel-id and the address of the allocated ring buffer. At this point, the open at A blocks (actually, sleeps for a rendezvous with the corresponding open on the other end of the channel at B). When the corresponding open takes place at B, B likewise locally allocates a ring buffer for the channel, and sends the channel-id and physical address of the start of the buffer to A. On receiving this message, A returns from the pending open. At this point, the channel is established, and user processes can read/write/select from A to B and vice-versa over this channel.

Sends/receives of messages on a regular channel follow the same procedure as on the control channel, except that message sizes on regular channel are unbounded. If the message is too large to be immediate, the sending end puts a number of 'address-length' pairs on the channel. The receiving end retrieves the header and several $< address, length >$ tuples corresponding to this transfer. Thereafter, the BTE on the receiving hub retrieves the particular number of bytes from the physical address specified. Note that the user-level read/write interface need not be concerned about the address-length pairs: they are constructed by the kernel when the user virtual addresses involved in the transfer aren't physically contiguous.

3.3 A read/write/select Interface

It is relatively simple to build the familiar read/write/select API over this point-to-point messaging system. A write translates to message sends, a read to message receives, and a select to a comparison of consume and cached-produce pointers. The crucial point for synchronous read/write/select is that on the arrival of a cross-call interrupt we need to check whether there is a pending read, a pending write or a pending select from which we can continue. It should be clear why a read or a select can be pending. A write can be pending if we run out of space to put a new message in the ring buffer for a channel.

One needs a mechanism to ensure that a synchronous write does not return prematurely. Upon returning from a synchronous write, the user expects that modifying the write-buffer will not affect the corresponding read. For performance reasons, we try to minimize large-buffer copying as far as practicable. If the buffer size of a write is small enough to fit in an immediate message, the data is copied into the header of the corresponding message and the write returns. However, if the size is larger, the appropriate address-length pairs are inserted into the appropriate channel's ring-buffer. If the write is flagged as synchronous, a bit is set in the header, and the sender goes to sleep. On seeing the synchronous write bit set, the receiver retrieves all the address-length pairs, and sends a cross-call interrupt to the sender. The sender wakes up, observes that the the consume pointer of the receiver has become equal to the sender's produce pointer – implying that the message has been retrieved. At this point, the synchronous write can return.

An added motivation behind synchronous writes is that we need to pin down the source and destination buffers used for long messages before the read can proceed. The BTE is essentially a DMA device, and we don't want the page-mappings to change while DMA is in progress. Synchronous writes ensure that we don't have to pin down a large number of pages because of many simultaneous long-writes.

3.4 Overheads of the Mechanism

In the best case – immediate messages, with the read being posted before the write – a write has to acquire a semaphore. The message headers are prepared and the cross-call interrupt associated with the send is then performed. The reader would be sleeping (waiting on a semaphore) for data to arrive. We'd incur a context switch to the reader once the cross-call interrupt is handled, the reader would need to acquire a spin lock to gain sole control of the BTE, the transfer would be performed, and the read would then complete. In our prototype implementation, sending a cache-line (128 bytes) and receiving it at the destination requires a total of about 90 microseconds.

For the best case with a large (i.e., non-immediate) message, a few more steps are required: the appropriate number of BTE transfers for the body of the message (a single transfer for each address-length pair), and the overhead of waiting until completion if the write is synchronous. The write would acquire the semaphore, fill up the address-length pairs, and go to sleep. The read would BTE-copy the body of the message – in several installments if the message is not physically contiguous – and then send the cross-call interrupt corresponding to the write-completion. The kernel on the writing side would field this interrupt, context-switch to the write, the write would wake up, and only then would the synchronous write complete.

Detailed performance measurements of the point-to-point device are reported in Section 4.3, and a graph of comparative performance is presented in Figure 4.

3.5 Dependence on the BTE

One might ask how dependent the point-to-point communication mechanism is on the BTE. The only feature of the BTE used for correctness is the safe-copy feature, as mentioned in Section 3.1. For performance in the case of large messages, the high-bandwidth afforded by the BTE is also invaluable.

In other similar architectures, where there may not be a BTE-like device, it may be possible to mimic the effects of safe-copy. There are two crucial requirements. Firstly, there must be some kind of backdoor-access method to allow copies to occur without opening the firewalls, and without polluting the caches (i.e., without having the sending partition's directory-entries point to nodes on the receiving partition; the problems due to cache-pollution are detailed in Section 4.2). Secondly, there should be support to synchronously inform the operating system if an error occurs. Several architectures provide an error-barrier. In general, at an error-barrier, faults that had occurred in the instruction sequence up to the error-barrier are reported synchronously (i.e., execution stalls to report these faults). If the cross-partition-copy mechanism could be put between two successive error-barrier calls, the effects would be much the same as the safe-copy feature of the BTE. Of course, for good performance with such a scheme, the error-barrier must be very efficient in the case when there is no error to be reported.

4 Shared-Memory

The point-to-point communication facility mentioned in the previous section offers very attractive fault-containment. The BTE pulls the message header and body across a partition boundary. If the source partition goes down for any reason before or during such a pull operation, the BTE-copy returns with the proper error status. The kernel takes appropriate action to gracefully handle the error. At worst, an application process will be informed about an incomplete message send or receive. Thus, the propagation of faults is essentially precluded.

However, there are certain drawbacks in with the aforementioned scheme. It has no support for efficient broadcast/multicast – which message-passing codes often need (e.g., in collective/ global operations). Further, communication over the point-to-point device requires the application to cross several layers of the operating system before it can interact with its counterpart in a different failure domain. The overhead includes dropping into the kernel at the sending partition, creating the appropriate header for the communication, acquiring locks/semaphores, sending a cross-call interrupt in hardware, the destination partition fielding this interrupt, a context switch, pulling across the header, pulling across the message body if the data is too large to fit into the header, and finally checking for pending signals when the read system call returns. This leads to a somewhat high communication latency of 90 microseconds, which is clearly not good enough for certain operations: e.g., synchronization between the processes of an application that spans partitions.

An alternative approach is to open up coherent shared-memory between partitions. Architecturally, this is possible since the underlying hardware maintains cache coherence across all the processors and physical memory in the machine irrespective of partitioning. With such shared-memory, communication latencies are limited not by the operating system but by the speed of the underlying hardware: how many network hops are required to reach the remote memory, how congested the interconnection currently is, etc. The drawback of this scheme is in fault-containment: since processors on a partition A will be allowed to access memory on a different partition B, care must be taken that the failure of A does not cause B to panic, or vice-versa. In order for memory that is physically on a different partition to appear like any other virtual memory to a processor, we set up the appropriate virtual-to-physical mapping at the virtual address requested by a process. Thereafter, accesses to the range of virtual addresses in the mmapped part will transparently access remote physical memory. This is a two-step process. First, an application must make a range of virtual addresses available to others and the OS on that partition must set the appropriate protection mask in the firewall registers. This allows the exported range of physical addresses to be accessed by specific remote partitions. After this, the second step can occur: a process in a remote partition can map in the memory for communication to take place.

4.1 Cross-partition Mappings

The infrastructure for the shared-memory interface consists of a driver which manages kernel aspects locally, and a daemon which communicates with the local kernel and with the corresponding daemon on remote partitions. This daemon allows distinct kernels to communicate via an RPC-like mechanism, and uses the point-to-point device for such communication. We use the point-to-point device to communicate between distinct kernels in spite of its high latency because the daemon (and driver) are involved only in setting up and tearing down of the channels. Once the channels are set up, the OS is not involved in communication unless there is an error in the system. Since lowering steady-state communication latency (rather than speed of setting up shared-memory) is our aim, the point-to-point device was an appropriate communication mechanism. Another important objective was to not have distinct kernels to read/write each others' memory in order to communicate – the importance of which will become clear when we discuss error-handling (Section 4.2).

On each partition, there are a number of channels, which can be used for shared-memory communication with a different partition. Each channel is represented as a device in the filesystem. At an mmap on such a device, the 'backing partition' associated with the channel allocates the physical memory for the channel. Thereafter, an mmap on the channel from any other partition causes the protection firewall on the allocated physical memory to be opened for the accessing partition, and this physical memory to be mapped to user-space on the accessing partition – provided the user processes performing the mmap has the appropriate access permissions. This is shown in Figure 3; the partition that allocates the physical memory for a channel is called the allocating partition for that channel; all other partitions are non-allocating partitions. The

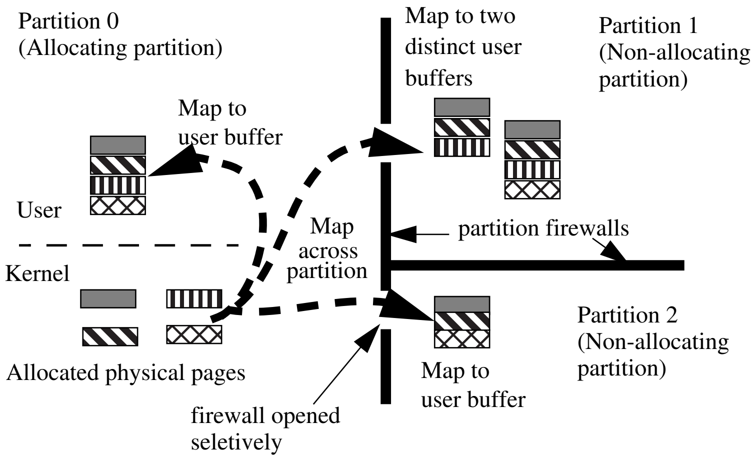

Figure 3: Mapping memory across partitions (single channel shown).

user deals with device handles, rather than virtual/physical addresses. After the mappings are performed on the devices, an application just performs user-level reads/writes into the buffer at which the mmap was performed. Since the pages on any channel can be mapped to multiple partitions by multiple user processes, we immediately get an efficient multicast type of operation through a single-writer-multiple-reader paradigm.

Some information, used during error-recovery, is recorded at an mmap. The kernel on each mapping partition records the process-group which performed the map, and the physical address at which the backing physical memory is located. This data-structure is called the map-table. An allocating partition stores similar information when an mmap happens for a non-allocating partition. That data structure is called the reverse-map-table. If a partition goes down gracefully (e.g., administrative shutdown), the reverse-map-table is consulted for all other partitions that have mapped a channel with physical memory on the partition going down. Each of these remote partitions is then asked to send an appropriate signal to the relevant process groups on those partitions. Conversely, the partition going down consults its map-table to find which allocating-partitions it had mapped memory from. All such partitions are informed of the demise of the partition being shut down, so that they can update their reverse-map-tables and reference counts associated with the cross-partition physical pages appropriately. When a partition comes up, it broadcasts a message to all the other partitions, which then unmap any memory they might have previously mapped from the upcoming partition (i.e., before that partition went down), and update their reverse-map-tables and reference counts.

The above approach takes care of orderly shutdown of partitions. The physical addresses in the map-tables were not used in handling shutdowns. Error handling in non-shutdown cases are discussed next, and that is where the physical addresses obtained from the map tables become important.

4.2 Error Handling

Broadly, three basic error scenarios have to be handled for cross-partition pages: firstly, failure of the home partition of a cache-line; secondly, failure of a non-home partition that has an exclusive copy of the cache-line; and thirdly, failure of a non-home partition that has a shared copy of the cache-line[12]. In the first two cases, the latest copy of the cache-line cannot be obtained. The third case produces cache-coherence protocol errors, since the failed partition will not acknowledge invalidations/interventions sent to it[2]. Kernel software has to handle these errors in order to ensure that the failure of one partition does not cause a different partition to fail.

Non-speculative errors: If the home partition of a cache-line fails, a user process' request for a read access will timeout (the requesting processor waits for an exclusive, shared, or speculative reply to the read request from the home, which will never arrive). A similar timeout arises with write accesses (the requesting processor waits for either a writeback exclusive or a writeback busy signal from the home, which will never arrive). If a non-home partition which has a copy of a cache-line fails, several types of errors might arise. In the case of a read-shared request, the requester waits indefinitely for either a shared-response or a shared-ack, and the home for either a shared writeback or a sharing transfer from the (failed) owner. In the case of a read-exclusive access, the requester waits indefinitely for either an exclusive response, or an exclusive-ack from the (failed) owner; the home waits indefinitely for a dirty-transfer, which never arrives. The requester gets a timeout in each of the above 'infinite wait' scenarios. A more severe failure is the case of a cache-line becoming stuck in a busy state. This can happen if a partition requests read access to a cache-line which is in exclusive state at another owner, and the owner fails before being able to send a dirty-transfer, or shared/exclusive writeback to the home in response to the request. The cache-line will be essentially locked under these conditions. If the directory state is observed to be busy by a requester, the request is usually re-tried. To handle the case of a locked cache-line, a count is maintained in the hub, and a timeout returned after the configured number of retries has been performed. Such a failed access results in the hub logging the type of access, and the physical address. The processor can later check these logs to inspect the failed access. The error-logging region of the hub is a ring-buffer, configurable in size, up to 64 MBytes.

When an access fails, the kernel inspects the physical address at which the failure has occurred. If the map-tables indicate that the address is a cross-partition address, the mapping is removed, the TLB entry flushed and a SIGBUS is sent to the application. Note that if a read/write from the kernel causes such a failed access, we can't take the simple contingency of sending bus-errors to the kernel! As mentioned earlier, the shared-memory interface is currently used by 'trusted' subsystems such as MPI and other system libraries, rather than directly by user applications[3]. In our current implementation,

[2] Details of the cache-coherence protocol on Origin can be found in[10].

[3] copyin/copyout checks to ensure that addresses passed to the kernel in syscalls are from the local partition.

there is no on-demand cached access to any cross-partition page by the kernel. However, if the need for such access arises, it is possible to use a nofault mechanism when there is no chance of processor-speculation: save the kernel state before any such access. If a bus error arises, and it is found that the kernel had caused the offending access, restore the kernel state and set a flag such that the access is not retried. A similar strategy was mentioned in Hive, where the kernel stack frame is saved/restored[3].

Though our kernel is programmed to never touch a cross-partition region, due to speculation in the MIPS R10000 processor, essentially *any* physical address might be (erroneously) generated, because of the unmapped K0 segment available in kernel mode. Thus, speculation while in kernel mode can generate an address corresponding to a cross-partition region. While a speculative access will not be graduated prematurely, the cache-line on which said speculation occurs might migrate as a side-effect. Two things can go wrong as a result. Firstly, the partition corresponding to the current owner of the cache-line being accessed, might have failed. This would cause the memory-controller on the accessing node to log the access as erroneous. If the processor graduates the speculative access later, it will get a bus-error. Secondly, the access might go through, causing the cache-line to migrate; after this the accessing partition might fail. In either case, as far as cache-coherence is concerned, the results would be identical to the cases discussed under 'Non-speculative errors' earlier in this section. These speculative errors are handled in the same way as those due to on-demand accesses. The user processes which had mapped the offending cross-partition address are notified, the mappings are removed, TLBs for those processes flushed, and the kernel continues. It makes sense to remove the mappings, because had the application touched these pages, it would have received a bus error anyway.

In addition to detection of failed partitions due to access failures, a coarse heartbeat goes on between partitions. The heartbeat merely increments counters through uncached accesses in the kernel. In addition, each partition keeps a mask of all other partitions that it knows about. If a partition A misses heartbeats from another partition B, or if A notices that B has removed A from its mask of known partitions, A declares that B has failed. On detection of failure, the map-tables are used to unmap any memory that might have been resident on the failed partition.

4.3 Performance

As mentioned earlier, the performance figures of interest are not the time required to set up mappings across partitions, but the time taken to communicate once the mappings have been set up. Since each cached access brings an entire cache-line (128 bytes) over to the accessing partition, we report the performance numbers in cache-line granularity. The experiments reported here were performed on a two module Origin-2000 machine, partitioned into two systems. Each partition had a single module consisting of two nodes (four CPUs). Each CPU was a 195 MHz R10000, with 32 KBytes each of primary instruction and data cache, and 4 MByte of secondary data cache. The hubs were clocked at 97.5 MHz.

Number of cache-lines	Shared-memory latency (μsecs)	Point-to-point latency (μsecs)
1	3	90
2	7	126
4	14	126
8	20	126
16	32	130
32	55	133
64	100	144
128	173	161

Table 1: Latencies of ping-pong test for zero offset. Each cache-line consists of 128 bytes.

Table 1 shows communication latencies for ping-pong tests of the stated number of cache-lines for the shared-memory and point-to-point devices. In the shared-memory case, each processor bzero-s the ping-pong buffer – the length of which is the stated number of cache-lines. It then updates a sequence number which is a part of the buffer. Further, the processor waits for the sequence number to be updated by its counterpart in the remote partition, and then bcopies the entire buffer. The order of reading and writing is reversed in the two processors (one reads first, the other writes first). For the point-to-point device case, ping-pong consisted of a synchronous write followed by a read (and vice-versa). The two processors participating in the ping-pong were on two distinct partitions. Performance numbers reported here are an average of between ten thousand and one million repetitions and are a half of the round-trip latency in each case.

A couple of effects ensure that performance improves with increases in message size. In the case of the shared-memory device, the write is a one-shot bzero, instead of writes of individual bytes. Further, bcopy had been optimized to prefetch later cache-lines, so that after the first few cache-lines are read, data is found in the cache instead of having to fetch from remote memory. For the point-to-point device, the large increase in latency in going from 1 to 2 cache-lines is because immediate messages were configured to be up to 1 cache-line in size. For larger messages, the receiver has to get the header first, and then perform another copy for the message-payload. Once primed, the actual transfer over the BTE is fast enough that the increase in latency in going from 2 to 8 cache-lines is negligible. Note that a single ping-pong test going on on the system shows an unamortized case of the point-to-point device. Recall that the produce/consume pointers of all the channels are consecutive in memory, and latency of transfer over the BTE increases slowly. Therefore, to amortize costs, the cache-lines corresponding to the produce/consume pointers of all the channels are retrieved at each message receive. Thus, the design scales well when multiple channels are active simultaneously.

It should be noted that the BTE can perform high-bandwidth transfers of long messages which are cache-line aligned. The shared-memory based paradigm used the processor to bcopy the corresponding messages. However, if the

Number of cache-lines	Shared-memory latency (μsecs)	Point-to-point latency (μsecs)
1	6	104
2	10	150
4	19	157
8	22	165
16	35	202
32	60	239
64	111	339
128	211	515
256	410	940

Table 2: Latencies of ping-pong test for four-byte offset. Each cache-line consists of 128 bytes.

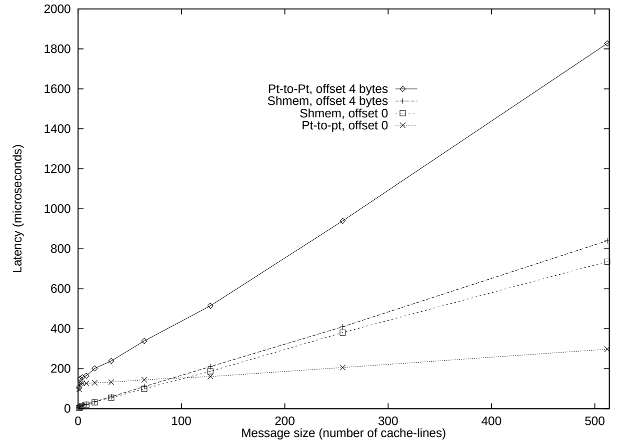

Figure 4: Latencies of inter-partition communication

message body is not cache-line aligned, and if the deviations from cache-line alignment (i.e., the 'offsets') at the source and destination are different, the point-to-point device must first bcopy the message into an intermediate aligned buffer before it is able to use the BTE-based copy for transferring this message. This slows down the transfer considerably, as shown in Table 2 for an offset of 4 bytes[4]. Performance for other non-zero offsets is much the same as what's shown below for the 4-byte case. A graph of comparative performance for these and larger messages is presented in Figure 4.

5 Interfaces to Shared Memory

Here we mention how familiar interfaces such as MPI and SysV shared-memory can be efficiently built over the cross-partition shared memory facility.

[4]Note that bcopy bandwidth is higher when the source and destination buffers are on different nodes than when they are on the same node.

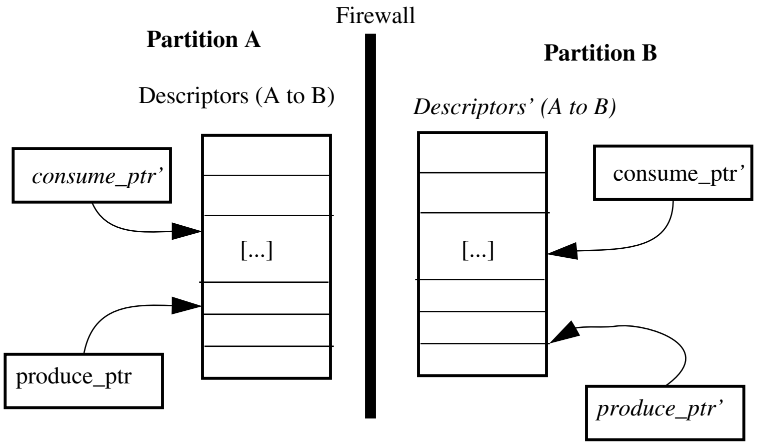

Figure 5: Message passing from Partition A to Partition B using inter-partition shared-memory.

5.1 A Message-Passing Interface

For various reasons it might be worthwhile to build a message-passing interface on a shared-memory machine. The comparative overheads of shared-memory vs. message passing differ from one application to another[2, 8]. Moreover, there are important legacy codes which require a message-passing style of communication – one which fits the point-to-point device perfectly. We contend that a message-passing library can be built efficiently atop the inter-partition shared-memory. Strategies for building such libraries over shared-memory have been documented[5, 7]. Therefore, instead of going into the specifics of building such an interface, we will briefly mention how the inter-partition shared-memory architecture can be used to advantage in a generic message-passing library.

As is evident from Table 1, it is more efficient to send short inter-partition 'messages' – control messages as well as headers/descriptors – over shared-memory. Transfers of cache-aligned long messages will be faster if the point-to-point device is used, because the BTE is a very efficient copying device.

Figure 5 shows how descriptors for sending messages from one partition A to a different partition B can be efficiently transferred over shared-memory. In order to send a message from A to B, A writes a descriptor, which B reads. Typically, if the message is 'small', the payload is sent as a part of the descriptor. For larger messages, the descriptor has 'pointers' to where the payload can be obtained from. In either case, the sender writes the descriptor and payload, while the receiver reads it. For an implementation over inter-partition shared memory, the message-passing library on the sending side allocates the backing physical memory for the descriptors, and maps it to the user-space, while on each receiving partition, the library just maps the allocated descriptor-space to user-space. The sending side also allocates a produce-pointer – which is an index into the descriptor-space of where the next descriptor can be written. The receiving side maps this produce-pointer into user-space. The receiving side, similarly, allocates a consume-pointer which denotes how many descriptors have been read. The sending side maps the consume pointer into user-space. In case a broadcast/multicast type of operation is required, each receiver must map the descriptor-space and produce-pointer, and the sender must map the consume-pointer of each receiver. Only a single receiver is shown in Figure 5. The items in italics represent the mapped versions of the corresponding allocated items, which are shown in normal-font. The receiving processes poll their mapped produce-pointer to check for a new message, while the sending process polls its mapped consume pointers to determine how much memory is left in the descriptor-space. Since the descriptor space, as well as the produce-pointer are mapped across partitions, when the user-process on partition A writes a new descriptor, the receiving process on partition B sees that the produce-pointer's value has changed, reads the new descriptor, and updates its consume-pointer. On seeing the updated consume-pointer, the sender knows that space has been freed up for an additional descriptor. Note that whatever has been just stated about descriptors also works for any non-immediate message which can be received via a mailbox type of interface.

There are several attractive features of this scheme. Firstly, in the absence of new messages, polling of the produce and consume pointers will read in-cache versions, which ensures that the reads are fast, and extraneous traffic is not generated on the interconnect. It is important to make these polls fast. Message-passing libraries often internally 'peek' at incoming queues to test whether there is a waiting message, without actually consuming it; thus, the processor might not be otherwise idle when polling for incoming descriptors. Secondly, since the sender allocates space for the descriptors and produce-pointer, and the receiver allocates the space for the consume-pointer, it is assured that only the home of a cache-line writes that cache-line; all other partitions merely read the cache-line. Hence, this strategy stands the best chance for recovery from partition failure. If the sending partition fails, there will be no message to be received (thus, the receivers can unmap the produce-pointers and descriptor space); if any receiving partition fails, it is assured that there is no loss of descriptor-data. The sender has to detect the failure of the receiver, and disregard its consume-pointer while accounting for descriptor-space[5]. Thirdly, receivers can join a communication group on-the-fly, since all the state required to start receiving messages is maintained at the sender. The descriptor-space can be mapped at any time, not necessarily at channel initialization; when a new receiver joins a communication-group, its consume-pointer should be set equal to the sender's produce-pointer, since it hasn't received any previous message. As a corollary, after a temporary failure, a receiver can re-join a multicast group if it is ready to forego the messages transmitted while it was down. Finally, broadcast/multicast operations are almost as cheap as unicast. The sender just has to keep track of more than one consume-pointer while accounting for descriptor-space.

[5] That being said, we should point out that on processors which perform speculative execution – such as the R10000 – a read access might be (mis)predicted as a read-modify-write access, in which case the receiver can load the cache-line in exclusive mode.This would annul the aforementioned advantage, unless the processor supports a strict 'read-only' mode of mapping, which takes precedence over (mis)prediction.

5.2 SysV Shared Memory

There are essentially two orthogonal problems in implementing the SysV model of shared memory in a distributed environment – such as a partitioned Origin. The first is that of key-space management, and the second is that of coherence. In what follows, we have assumed that key-space management is performed outside of inter-partition shared memory. The processes involved in the shmget() and other SysV key-handling operations can agree on a key ahead of time, or exchange messages at runtime to agree on a key.

We define a new bit, IP_XP, in the flag associated with shmget(). If a new shared-memory region is being created (i.e., the key is IPC_CREAT), and the user wants this shared-memory region to be visible across partitions, the IP_XP bit should be set. If this bit is set on a shmget() call, the kernel allocates an inter-partition shared-memory buffer, and returns a shmid for the allocation. The upper bits of the shmid returned contain the partition number on which the allocation was performed. This shmid is submitted at a later shmat() call to attach the denoted region to the caller's address space. The kernel on the calling partition performs an RPC (see Section 4.1) with the kernel on the partition denoted in the shmid. If the caller has the proper credentials, the allocating kernel increments a reference count associated with this buffer, and opens the firewall for the calling partition. At this point, the calling kernel maps the inter-partition shared-memory region into user-space. At a shmdt() call, the calling kernel performs a similar RPC with the allocating kernel. The reference count associated with the cross-partition buffer is decremented. If the reference count drops to zero as a result, the buffer is freed and the firewall for that region of memory is closed. The other relevant shmctl() operations of IPC_STAT, IPC_SET and IPC_RMID can be performed similarly by consulting the allocating kernel, and returning the relevant information if the user has the proper credentials. On the other hand, the inter-partition buffers being kernel buffers, they would never be swapped out. Therefore, the shmctl() operations of SHM_LOCK and SHM_UNLOCK can be silently ignored.

6 Discussion and Future Work

Future architectures and operating systems can work synergistically to improve the type inter-partition error-handling mentioned here. As shown in Section 5.1, it is enough for non-allocating partitions in message-passing libraries to access inter-partition shared memory in read-only mode. The chance of data-loss due to partition failure is lower if a non-allocating (thus, non-home) partition does not acquire a cache-line in exclusive mode. However, the cache-coherence protocol on Origin provides a cache-line in exclusive mode to the first processor accessing it – even if the access is read-only. While this optimization enhances application performance, it complicates support for failure recovery. With a read-only mode that precluded misprediction of writes, recovery/availability would be facilitated.

In our current implementation, there is no support to disseminate information about failed partitions to other partitions. Had there been such support, and some additional architectural assistance detailed hereunder, it could be effectively used to reduce error-recovery overhead and false interconnect traffic. The failure-detection algorithm could be run as a distributed algorithm either in hardware or software. As output, it would update the region-mask (see Section 2). The node-controller could check that the destination of a cache-request is indeed running before it forwarded such a request[6]. Under these circumstances, the overhead for error-recovery would be greatly reduced. When a partition failure is detected, we wouldn't have to signal all relevant processes and undo their inter-partition mappings. It would be enough to mark the corresponding bit in the region-map and continue. The node-controller would detect any access to a cache-line on a failed partition, and send bus-errors to the application. Further, if the cache-coherence mechanism can be made aware of the partition-map, it might be possible to drop all the nodes of a failed partition from the coherence domain. This would allow read-exclusives and writebacks to a cache-line which is held in shared mode at a failed partition, without having to wait for invalidate-acknowledgments from the failed nodes.

We did not consider failures of individual interconnect links in the current work, only failure of partitions as a whole. While the Origin architecture supports re-programming of routes around failed links, the current work used static routes. Thus, we have the stipulation that communication between two partitions A and B must never go through a third partition C, because the failure of C would then cause A and B to stop talking with each other. This restricts the topologies of partitioning that are currently supported.

7 Related Work

Building fault-containment partitions from the hardware up appears to be a relatively new phenomenon, at least on shared-memory multiprocessors. Some of the work done in Hive[3] is related to our work in mapping physical memory on one protection-domain to user buffers on a different domain. Nevertheless, several features distinguish our work from Hive. Unlike Hive, we considered distinct kernels directly mapping/accessing each others' data as unacceptable for security and fault containment. Instead, we use a user-level RPC-like mechanism is to communicate with a remote kernel. The implication is that normally, no partition will fault due to a kernel access to a remote, failed partition. Therefore, the so-called careful-reference protocol of Hive was irrelevant in our case.

Perhaps the most closely related to our work is the effort to provide architectural support for hardware fault containment in the FLASH multiprocessor[12]. Several of the issues that we have to contend with in software: administrative shutdown, failure/recovery of partitions, the effects of on-demand and speculative-access-related cache-line migration, etc., also arise during the design of said hardware in FLASH. However, while the FLASH researchers are chiefly concerned with building architectural primitives that facilitate containing hardware faults and making accesses safe, our focus is on efficient communication across partitions with software support to recover from the faults that might arise due to fail-stop of partitions.

Efficient communication at the user-level has been a subject of considerable research for quite some time. Efforts in this area range from the use of clever operating system features that avoid copying as far as practicable [1, 4] to the use of specialized atomic-swap and fetch-and-op hardware[9].

8 Conclusions

We mentioned why it is important to be able to break up a large computer into autonomous partitions, and described features of the Origin-2000 that allow such partitions to be built. Describing the implementation of a point-to-point, risk-free communication device, we showed that it leads to rather high latencies of inter-partition communication. Hence, we built a fast communication mechanism through selective opening of the firewalls. From a fault-containment standpoint, the point-to-point device is the safer means of communication. However, we considered the errors that might arise due to partition failures after firewalls are selectively opened, and added support in the kernel to handle such errors. Error recovery is somewhat crude in the current implementation: on detecting a failed partition, the kernel merely signals the relevant process group that was using memory resident on the failed partition, and leaves further recovery to the application. However, this approach is in keeping with Unix faulting/SIGBUS semantics.

One is often faced with a choice of communication paradigms. There are applications (such as databases) which prefer a somewhat higher latency of communication if it means more attractive fault-isolation. Further, the typical message size in many applications is of the order of a few kilobytes – low latency for small messages may not be of pressing concern. Such applications would want to use the point-to-point communication based paradigm. On the other hand, there are several applications in high-performance computing which exchange short messages, want as fast communication as possible, and can tolerate the effects if a hardware error signals the application. These would stand to gain from the explicit shared-memory paradigm.

Acknowledgments

Eric Salo is perfecting an implementation of MPI over the inter-partition shared memory. Ranga Rajagopalan did the PROM work required to support fault-containing fences. Len Widra, Dan Teodosiu and Ben Fathi provided useful feedback on an earlier revision of this paper. We are grateful for relevant discussions with Curt Schimmel, Bhanu Subramanya, Ajit Dandapani, Uday Hegde, Steve Whitney, Alex Petruncola and Luca Castellano. Finally, the feedback of the anonymous reviewers greatly helped in improving the paper.

[6] FLASH researchers have proposed the use of a similar structure – the node-map – to detect references to inaccessible cache-lines[12].

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, pages 175–198, May 1991.

[2] Satish Chandra, James Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, October 1994.

[3] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Don Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, December 1995.

[4] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–203, December 1993.

[5] Cezary Dubinicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[6] Mike Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip. *Proceedings of Hot Interconnects Symposium IV*, August 1996.

[7] Kaushik Ghosh and Stephen R. Breit. Evaluating the Limits of Message Passing via the Shared Attraction Memory on CC-COMA Machines: Experiences with TCGMSG and PVM. *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[8] Vijay Karamcheti and Andrew Chien. Software Overhead in Messaging Layers: Where Does the Time Go? *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–60, October 1994.

[9] Vijay Karamcheti and Andrew Chien. A Comparison of Architectiral Support of Messaging on the TMC CM-5 and Cray T3D. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, June 1995.

[10] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[11] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, December 1995.

[12] Dan Teodosiu, Joel Baxter, Kinshuk Govil, John Chapin, Mendel Rosenblum, and Mark Horowitz. Hardware Fault Containment in Scalable Shared-Memory Multiprocessors. *Proceedings of 24th Annual International Symposium on Computer Architecture*, pages 73–84, June 1997.