ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors [*]

Milos Prvulovic, **Zheng Zhang**[‡ †], **Josep Torrellas**
University of Illinois at Urbana-Champaign
[‡]Hewlett-Packard Laboratories
http://iacoma.cs.uiuc.edu

Abstract

This paper presents ReVive, a novel general-purpose rollback recovery mechanism for shared-memory multiprocessors. ReVive carefully balances the conflicting requirements of availability, performance, and hardware cost. ReVive performs checkpointing, logging, and distributed parity protection, all memory-based. It enables recovery from a wide class of errors, including the permanent loss of an entire node. To maintain high performance, ReVive includes specialized hardware that performs frequent operations in the background, such as log and parity updates. To keep the cost low, more complex checkpointing and recovery functions are performed in software, while the hardware modifications are limited to the directory controllers of the machine. Our simulation results on a 16-processor system indicate that the average error-free execution time overhead of using ReVive is only 6.3%, while the achieved availability is better than 99.999% even when the errors occur as often as once per day.

1 Introduction

Cache-coherent shared-memory multiprocessors are seeing widespread use in commercial, technical, and scientific applications. In recent years, fault-tolerance has become an increasingly important feature of such systems. In some commercial applications, high *availability* is needed, as business transactions are being processed by the system. Some applications execute for a long time and require a highly *reliable* execution environment. Examples of such applications are those that mine large data sets and many simulations. Unfortunately, both availability and reliability are difficult to achieve in modern large systems. Improvements in silicon technology result in smaller feature sizes, while power dissipation constraints result in lower operating voltages. Both of these make modern integrated circuits prone to transient and permanent faults. In large systems the problem is worse, as those systems contain many interacting components that must all operate correctly.

To deal with these problems, much work has been done in error recovery. Typically, error recovery mechanisms are categorized into Forward and Backward Error Recovery (FER and BER). With FER, hardware redundancy is added to the system, which makes it possible to determine the correct outcome of an operation, even if one (or more) of the participating devices fails. It is possible to design cost-effective FER that targets only a single device, such as the processor core [3, 28, 30]. However, general-purpose FER is not cheap. The most popular such method is triple-modular redundancy (TMR), in which each operation is performed by three identical devices and a majority vote decides the correct result. For most systems, the cost of TMR is prohibitively high. BER, also called *rollback recovery* or *checkpointing*, can be used in such systems. With rollback recovery, the system stores information about its past state. When an error is detected, this information allows the system to be restored into a previous error-free state. The main advantage of BER is that no hardware replication is required. However, it has three disadvantages: the performance overhead during error-free execution, storage overhead, and the higher recovery latency.

In this paper, we present ReVive, a novel, cost-effective scheme for rollback recovery in shared-memory multiprocessors with distributed memory. ReVive is compatible with off-the-shelf processors, caches, and memory modules. It only requires modifications to the directory controllers of the machine, to perform memory-based distributed parity protection and logging in the background. Both hardware and storage requirements are very modest.

ReVive has both good error-free performance and quick recovery from a wide class of errors, including permanent loss of an entire node. Our experiments with 12 applications on a simulated 16-processor system show that the average overhead of error-free execution is only 6.3%. When an error occurs, the system is unavailable for less than half a second on average, including the correct work lost due to the rollback. The resulting availability is better than 99.999%, even when errors occur as often as once per day.

This paper is organized as follows: Section 2 presents a novel taxonomy of BER schemes for multiprocessors; Section 3 presents the design of ReVive; Section 4 explains some implementation issues in ReVive; Section 5 presents our evaluation setup; Section 6 contains the evaluation; Section 7 describes related work; finally, Section 8 concludes.

2 BER in Multiprocessors: A Taxonomy

To understand the design space of BER schemes, we have designed a taxonomy that classifies the schemes according to three axes: how checkpoint *consistency* is achieved, how the *separation* between the checkpoint and the working data is done, and how checkpoint *storage* is protected from errors. Figure 1 shows the resulting design space. We now consider each axis in turn.

[*]This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM, Intel, and Hewlett-Packard.

[†]Currently at Microsoft Research Asia: zzhang@microsoft.com.

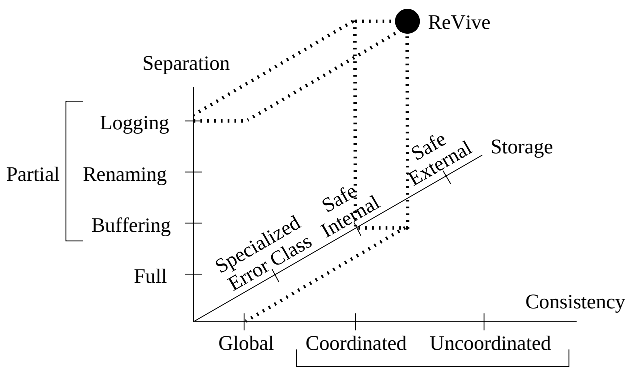

Figure 1. Design space of multiprocessor BER schemes.

2.1 Checkpoint Consistency

Since threads executing on different processors interact with each other, they may create *recovery dependences*: when one processor is rolled back, it may be necessary to also roll back other processors. To maintain checkpoint consistency, three approaches are used:

Global. All processors periodically synchronize to create a single, global checkpoint [8, 13, 14, 15, 20, 21]. This is the simplest approach.

Coordinated Local. Each processor periodically creates a local checkpoint of its own state. If the processor has been interacting with some other processors, then those other processors are forced to create their own checkpoints at the same time [1, 4, 5, 25, 32]. The advantage is that independent computations do not synchronize for checkpointing, while a disadvantage is that interactions must be recorded.

Uncoordinated Local. Each processor periodically creates local checkpoints. Interactions between processors are recorded but do not affect checkpoint creation. At recovery time, however, local checkpoints and interactions are used to find a consistent recovery line. This approach allows synchronization-free checkpointing, but runs the risk of the *domino effect* [22]. Uncoordinated checkpointing is mostly used in loosely-coupled systems, where communication is infrequent and synchronization expensive [6, 7, 26]. However, it has also been used in tightly-coupled systems [27].

2.2 Checkpoint Separation

We group schemes into four classes based on how checkpoint data is separated from working data.

Full Separation. Checkpoint data is completely separate from working data [4, 6, 26]. A naive way to establish a checkpoint is to copy the entire state of the machine to another area. A better way is to realize that much of the machine state does not change between checkpoints. Thus, establishing a new checkpoint consists of merely updating the old one, by copying into it the state that has changed since the old checkpoint was established. There are other optimizations to reduce copying, such as *memory exclusion* [18].

Partial Separation with Buffering. With partial separation, checkpoint data and working data are one and the same, except for those elements that have been modified since the last checkpoint. Consequently, less storage is needed [21]. With buffering, the modified elements are accumulated in a buffer, typically a cache or a write buffer [1, 5, 32]. When a new checkpoint is created, the main state of the machine is typically updated by flushing the buffer into main memory. While checkpoint generation may be regularly scheduled, it may also be asynchronously triggered by buffer overflow.

Partial Separation with Renaming. When a checkpoint is established, all state is marked as read-only. An update to a page causes the page to be copied to a new location, which is marked as writable and mapped into the working state in place of the original page. The original page is no longer part of the working state but remains in the checkpoint state. When a new checkpoint is established, all such pages are garbage-collected [8, 15, 20]. In COMA machines, this approach can be used at the granularity of memory lines [14, 15].

Partial Separation with Logging. Logging does the opposite of renaming: the old, checkpoint value, is copied to a *log*, while the original location is modified and remains part of the working state [13, 25, 27, 32]. As a result, logging does not require support to map and unmap pages or memory lines into and out of the working state. This makes logging more suitable to fine grain copying, which minimizes fragmentation. Typically, the log is a contiguous structure which contains data that is needed only for rollback recovery to a previous checkpoint. Once a new checkpoint is established, the log space can be easily reclaimed without requiring garbage collection mechanisms.

Different separation mechanisms may be used for different parts of a machine's state. For example, both buffering and logging are used in [32].

2.3 Checkpoint Storage Protection

Finally, we group schemes into three classes based on how the checkpoint storage is protected from errors:

Safe External Storage. The checkpoint is stored in external storage that is assumed to be safe [6, 26]. Typically, such storage is a disk array. Since RAID can be used to protect disks against most common errors [17], the assumption of safety is reasonable.

Safe Internal Storage. The checkpoint is stored in main memory or other internal storage and made safe through redundancy across the nodes. Checkpoint state can be restored even if storage on a limited number of nodes (typically one) is damaged. In some systems, safe internal storage is provided by duplication of checkpoint data in main memory [5, 8, 14, 15]. Alternatively, it can be provided using $N + 1$ parity. In this case, lost checkpoint data in a node can be recovered by examining the memories of the other N nodes [20, 21]. Checkpointing to main memory is much faster than checkpointing to external storage [21].

Specialized Fault Class. The checkpoint storage is not protected with redundancy across nodes. However, the system is not expected to recover from faults that can damage that storage [1, 4, 13, 25, 27, 32]. For example, a design for recovery from processor errors can keep the checkpoint in caches or memory without redundancy, while a system designed for recovery from cache errors can keep the checkpoint in main memory.

Overall, designs using safe external storage can recover from even the most general fault, namely loss of the entire machine. The designs using safe internal storage cannot recover if more than a certain number (typically, one) of internal storage components are faulty. Finally, the designs with a specialized fault class cannot recover from even a single error that makes any checkpoint storage component unavailable.

3 ReVive Design

This section presents our cost-effective design for rollback recovery. We first discuss the choice of design point (Section 3.1), then describe the mechanisms supported (Section 3.2), and finally explain our choice of parameters for the design (Section 3.3).

3.1 Choice of Design Point

Our goal is a cost-effective general-purpose rollback recovery mechanism for high-availability and high-performance shared-memory multiprocessors. Cost-effectiveness implies that only a modest amount of extra hardware can be added. General-purpose implies recovery from a wide class of errors, including permanent loss of an entire node. High availability requires that system downtime due to an error be short. Finally, high performance mandates low overhead during error-free operation.

ReVive is compatible with off-the-shelf processors, caches, and memory modules used in modern multiprocessors. For example, the SGI Origin 2000 [11] uses off-the-shelf processor chips, which include caches and cache controllers, and can use off-the-shelf DRAM modules. The major custom-designed components are the directory controller, the network interface and the memory controller. We keep our hardware modifications limited to the directory controller (Figure 2).

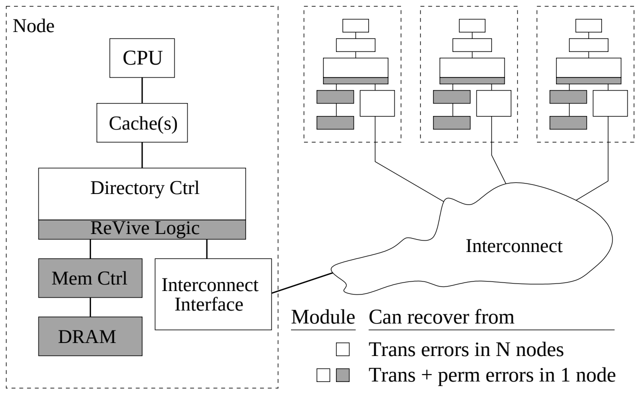

Figure 2. Scalable shared-memory multiprocessor with ReVive.

Our design choice is influenced by the availability requirements. The error frequency we expect is from once a day to once a month [19]. To achieve reliability of 99.999%, our target system's unavailable time due to an error should be no more than 864 milliseconds for the high error frequency range and no more than 24 seconds for the low error frequency range.

In the rest of this section, we discuss which design point in our taxonomy of Section 2 is most conducive to our goal, give an overview of our solution, and then explain the types of errors from which our scheme can recover.

Checkpoint Consistency: Global. Global schemes are the simplest because they do not need to record interactions between processors. Furthermore, they are suited to shared-memory machines, where processor communication and synchronization are efficiently supported. For example, in the Origin 2000, 16 processors can synchronize at a barrier in 10 μs [10].

Checkpoint Separation: Partial Separation with Logging. Partial Separation schemes have low storage overhead and, because they restore only a fraction of the working state, recover quickly. Among these schemes, Logging is the most flexible. With Logging, we can choose the checkpoint frequency; with Buffering, buffer overflows trigger checkpoint generation. With Logging, we perform fine-grain copying, which has low overhead and minimizes memory fragmentation; with Renaming, the copying can only be easily done in software at the page granularity. Finally, the simplicity of logging allows an efficient hardware-assisted implementation through simple extensions to the directory controller.

Checkpoint Storage: Safe Internal Storage with Distributed Parity. Given the low speed of disks, using external storage for checkpoints typically induces a high recovery time. Furthermore, it dictates a low checkpoint frequency to maintain tolerable overhead under error-free conditions [21]. For this reason, we store the checkpoint data in memory. However, since we target a broad range of errors, we must assume that the contents of main memory can be damaged and even a node can be lost. Consequently, we protect memory with parity distributed across memory modules. This scheme uses much less memory than mirroring. Additionally, instead of having dedicated parity node(s) as in [20], we distribute the parity pages evenly across the system. This approach allows all nodes to be used for computation and avoids possible bottlenecks in the parity node(s). Finally, instead of updating the parity in software at checkpoint creation time, we extend the directory controller hardware to automatically update the distributed parity whenever a memory write occurs. This approach reduces the overhead of creating a checkpoint.

3.1.1 Overview of Solution

During error-free execution, all processors are periodically interrupted to establish a global checkpoint. Establishing a checkpoint involves flushing the caches to memory and performing a two-phase commit operation [23]. After that, main memory contains the checkpoint state. Between checkpoints, the memory content is being modified by program execution. When a line of checkpoint data in main memory is about to be overwritten, the home directory controller logs its content to save its checkpoint state. After the next checkpoint is established, the logs can be freed and their space reused. In practice, sufficient logs are kept to enable recovery across as many checkpoints as the worst-case error detection latency requires.

When an error is detected, the logs are used to restore the memory state at the time of the last checkpoint that precedes the error. The caches are invalidated to eliminate any data modified since the checkpoint and the execution can proceed.

To enable recovery from errors that result in lost memory content, pages from different nodes are organized into parity groups. Each main memory write is intercepted by the home directory controller, which triggers an update of the corresponding parity located in a page on another node. The parity information will be used when the system detects an error that caused the loss of memory content in one node (e.g., if a memory module fails or the node is disconnected). Then, the parity and data from the remaining nodes are used to reconstruct the lost memory content, which includes both the logs and the program state. Logs are recovered first. Then the regular rollback described above can proceed. After that, normal execution can continue, while the remaining lost memory is reconstructed in the background.

3.1.2 Types of Errors Supported

Error Detection Assumptions. In our system, we assume error detection support that provides fail-stop behavior [23] for the ReVive hardware in the directory controller (Figure 2). This can be done with careful design and judicious use of replication in that module. In addition, parity update messages and their acknowledgments have to be protected by error detection codes. Finally, the data paths in the memory controllers and memory modules also have to use error detection codes. All of this is needed to detect garbled parity or log updates before they damage the checkpoint state. We do not make any additional fail-stop assumptions. Of course, error detection latency must have an upper bound of no more than a few checkpoint intervals, to keep the space requirements in the logs reasonably modest. Further discussion of error detection mechanisms is beyond the scope of this paper.

Recovery from Multi-Node Errors. ReVive can recover from multiple transient errors that occur in the white areas of Figure 2 in multiple nodes simultaneously. For example, consider a glitch that causes a reset of all the processors in the system and the loss of all cached data. This leaves the checkpoint and the logs in memory intact, and so ReVive can recover. Another example when ReVive recovery is possible is an interconnect glitch that damages several in-transit messages in different parts of the network or network interfaces. However, ReVive cannot recover from multiple errors that occur in the gray areas of Figure 2 in multiple nodes simultaneously. For example, two malfunctioning memory modules on different nodes may damage a parity group beyond ReVive's ability to repair.

Recovery from One-Node Errors. ReVive can recover from multiple permanent or transient errors that occur in a single node. This includes complete loss of an entire node. For example, this occurs when a node's network interface permanently fails. In this case, ReVive performs recovery of the lost memory and a rollback. Another example of a one-node error is a single faulty processor that erroneously modifies memories in several nodes. After this error is detected, a rollback to a past checkpoint restores the system.

3.2 Mechanisms in ReVive

The new mechanisms are hardware-based distributed parity protection in memory and hardware-based logging. This section describes these mechanisms plus how to perform a global checkpoint and a rollback.

3.2.1 Distributed Parity Protection in Memory

In Section 3.1, we explained our decision to protect the checkpoint data in main memory by using distributed parity. Technically, parity protection is needed only for checkpoint data, as recovery would overwrite non-checkpoint data with checkpoint contents. However, in error-free execution non-checkpoint data later *becomes* the new checkpoint data. We speed up the creation of a

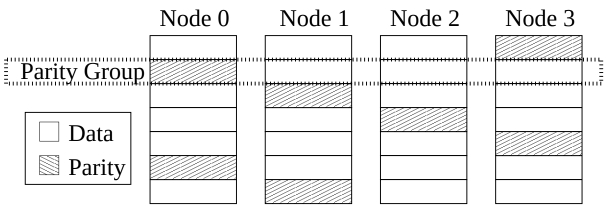

Figure 3. Distributed parity organization (3+1 parity).

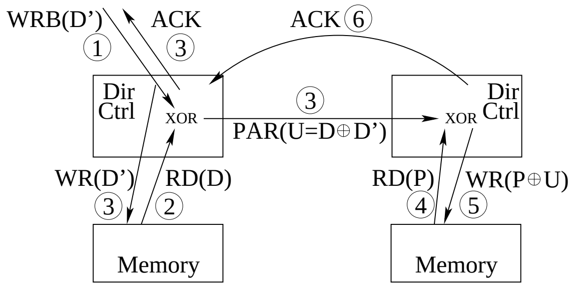

Figure 4. Distributed parity update on a write-back. Messages are numbered in the chronological order.

new checkpoint by protecting the *entire* main memory with distributed parity that is updated whenever the memory is updated. In this way, the distributed parity is already up-to-date when a new checkpoint is to be created.

Figure 3 shows how memory pages are organized into parity groups ($3 + 1$ parity is shown). Figure 4 shows the actions performed when a memory line is written-back to main memory. The home directory controller intercepts the write-back request D'. It first reads the current contents D of the line from memory. Then the new contents D' are written. At this time, the write-back can be acknowledged to the requester, if such an acknowledgment is required, but the directory entry for the line stays busy. The *parity update* $U = D\ XOR\ D'$ is computed and sent to the home of the parity. When it arrives there, the directory controller of the parity's home node reads the previous parity P, computes the updated parity $P' = P\ XOR\ U = P\ XOR\ (D\ XOR\ D')$, and stores it back. Then, the parity update is acknowledged to the home of the data. At such time, the directory entry for the memory line is marked as no longer busy and other transactions for that memory line can be processed.

Note that the same hardware can be used to support *distributed memory mirroring* (maintaining an exact copy of each page on another node). Mirroring is just a degenerate case of our parity protection mechanism, when one parity page is used to protect only one data page. In that case, the two memory reads and and the XOR operations in Figure 4 can be omitted.

Finally, we note that updating parity (or even mirroring) whenever data is written to memory would be prohibitively expensive if performed in software. However, with our hardware implementation, these updates are performed in the background while the processors continue program execution uninterrupted.

3.2.2 Logging

After a checkpoint is established, the checkpoint state consists of all the data in main memory. Subsequent program execution modifies this data. To prevent the loss of part of the checkpoint

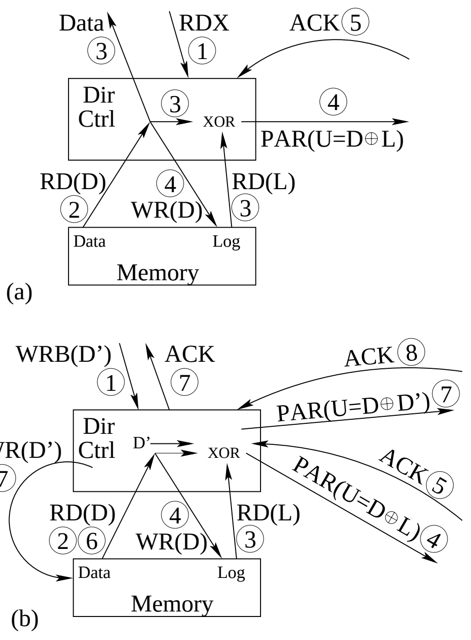

Figure 5. Logging and parity updates for (a) read-exclusive and (b) write-back access to a line that has not already been logged since the last checkpoint. Only the home node of both the data and the log is shown.

state, we use logging. Before a line in memory is written for the first time after a checkpoint, the previous content of the line is logged. In this way, all checkpoint data that has been overwritten can still be found in the log. Like the parity updates in Section 3.2.1, the logging is performed by our enhanced directory controller.

Main memory is modified by write-backs of dirty lines. When a write-back arrives at the home node of the data, we check whether this is the first modification of the line since the last checkpoint. If it is, the previous content of the line is read from memory and saved in the log before the new content of the line is written to memory. Note that the log and the data are in the main memory of the same node, and that both are protected by distributed parity. The log *and its parity* must be fully updated before the data line can be written.

Fortunately, most often we know that the block will be modified before the write-back is received by the home. Requests like *read-exclusive* or *upgrade*, which result from write misses in the cache or write hits on shared lines, signal an intent to modify the block. Figure 5(a) shows the operations performed by the hardware when a read-exclusive (RDX) message is received by the memory for a line that has not yet been logged since the last checkpoint. From Figure 5(a) we see that the data can be supplied to the requester as soon as it is read from memory. Alternatively, if an upgrade permission is all that is needed, it can be granted immediately. The logging is performed in the background by the directory controller. The directory entry for the block stays busy until the acknowledgment is received for the parity update. This ensures that no new operation is started for this block until its log entry has been fully created. When the write-back arrives, the line has already been logged and the write-back proceeds as shown in Figure 4.

In some cases, the directory controller may not receive a read-exclusive or upgrade message before it receives the write-back for a line. For example, this occurs in uncached writes and when the processor writes to lines in shared-exclusive state. In this case, the operations on the log and the data are performed as part of the same transaction. This case is shown in Figure 5(b). Note that the second read to the line D in memory could be eliminated if the contents read by the first read are cached by the directory controller. In our evaluation we do not assume such support, as it would require a small data cache in the directory controller.

A modified line only needs to be logged *once* between a pair of checkpoints. To this end, the directory controller is extended with one additional state bit for each memory line, which we call the *Logged (L)* bit. This bit is used to detect whether a particular line has already been logged. The *L* bits of all lines are gang-cleared after each new checkpoint is established. The *L* bit of a line is set when the line is logged, to prevent future logging of that line.

Table 1 summarizes the events that trigger parity updates and logging, the actions performed, whether the actions are on the critical path of the processor's execution, the number of additional memory accesses performed, the number of additional memory lines accessed and the number of additional inter-node messages required. As we can see, none of the actions directly affect the processor's execution, although the most complicated and, fortunately, least frequent case does result in delaying the acknowledgment of a writeback. We also see that, although the new operations require 3 to 8 additional memory accesses, they access only 1 to 3 additional memory lines. The remaining additional accesses are re-accessing already accessed memory locations. Furthermore, the log is accessed in a sequential manner, and so is its parity. Repeated accesses to the same memory line and accesses to consecutive lines can be performed very efficiently in modern DRAMs.

3.2.3 Establishing a Global Checkpoint

Parity updates and logging allow the machine to recover to a previous checkpoint state. Establishing a new checkpoint essentially commits the work done since the previous checkpoint. Because the main memory contains the checkpoint state, to create a new checkpoint we must first ensure that the entire current state of the machine is in the main memory. This is done by storing the execution context of each processor to memory and writing-back all dirty cached data to memory. Each processor waits until all its outstanding operations are complete. Then, we atomically commit the global checkpoint on all processors, which we do using a two-phase commit protocol [23]: all processors synchronize, mark the state as tentatively committed, synchronize again and fully commit. After the new checkpoint is established, we can free the space used by logs needed to recover to an old checkpoint that is no longer needed. If the maximum detection latency is small, we keep only two most recent checkpoints. This is needed because an error can occur just before establishing the newest checkpoint, but be detected after it is already established. In that case we recover to the second most recent checkpoint. For larger error detection latencies we can keep sufficient logs to recover to as many past checkpoints as needed. Support for that can be easily provided without additional hardware modifications.

Figure 6 shows the time-line of establishing a global checkpoint. The timing parameters are discussed in Section 3.3.

Event	Actions	Critical Path?	# of Extra Memory Accesses	# of Extra Lines Accessed	# of Extra Network Messages
Write-back to memory, already logged (L=1). Figure 4.	Update data parity	No, done after ack to CPU	3	1	2
Read-exclusive or upgrade, not yet logged (L=0). Figure 5(a).	Copy data to log	No, done after reply to CPU	1	1	0
	Update log parity	No, done after reply to CPU	3	1	2
Write-back to memory, not yet logged (L=0). Figure 5(b).	Copy data to log	No, but ack to CPU delayed	2	1	0
	Update log parity	No, but ack to CPU delayed	3	1	2
	Update data parity	No, done after ack to CPU	3	1	2

Table 1. Events that trigger parity updates and logging.

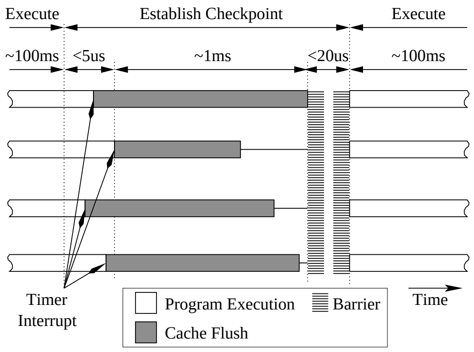

Figure 6. Time-line of establishing a global checkpoint.

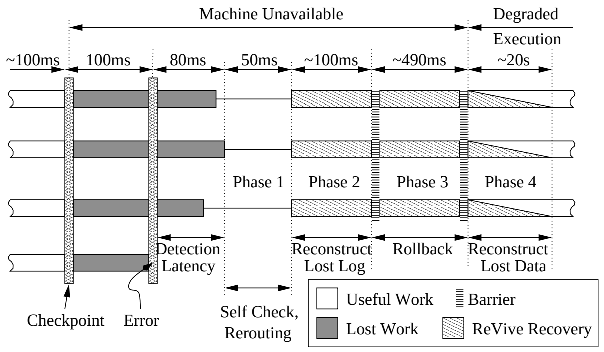

Figure 7. Time-line of recovering from node loss.

3.2.4 Rollback

Finally, we examine the operations performed when an error is detected and our rollback mechanism is activated. Figure 7 shows a time-line of recovery in the worst scenario, in which a node is permanently lost just before a new checkpoint is created[1]. When the error is detected, the Phase 1 of recovery involves testing the hardware and re-initializing it. This includes resetting the processors, invalidating the caches and directory entries and, in case of permanent errors, routing around the failed component. These steps are outside the scope of this paper. Phase 2 involves using the distributed parity to rebuild the contents of the lost node's log. This is needed only if the main memory contents of a node have been damaged or lost. Phase 3 involves using the logs to restore the main memory into a checkpoint state (rollback). Pages to which checkpoint data is restored are rebuilt on demand, using the distributed parity. At the end of Phase 3, parity groups affected by losing a node are marked as inaccessible and the program execution can continue. Figure 7 also shows barriers at the end of phases 2 and 3.

Recovery is not complete when the program execution continues. Because memory content has been lost, unavailable parity groups must be repaired. This is done by background processes, as Phase 4 of the recovery. The processes rebuild the missing pages of inaccessible parity groups. In addition, if program execution attempts to access an inaccessible page, the resulting page fault is handled by immediately rebuilding the group's missing page.

If an entire node has been lost, a large amount of memory can be inaccessible. Specifically, with $N+1$ parity and M megabytes per node, $M \times N$ megabytes of data and M megabytes of parity are inaccessible[2] due to either being lost or belonging to a parity group where another page has been lost. This means that the performance of the machine after the recovery can be degraded for two reasons: the machine has one less processor and the remaining processors are devoting some of their time to rebuilding the damaged parity groups. However, the machine is available during this time, performing useful computation and responding to external events, although with reduced computational capabilities.

The times shown in Figure 7 are worst-case unavailable times for the application that required the longest recovery time in our evaluation (Section 6.3). The unavailable time due to a node loss in the average case (error half-way into a checkpoint interval) and on average across the applications we study is only about 350 ms. We also note that there are many transient and even permanent errors that do not result in the loss of a node's memory. For example, errors in the processor core or caches of a node may leave the memory of that node fully operational and accessible. In such cases, no reconstruction of any lost pages is needed. Consequently, Phases 2 and 4 in Figure 7 are completely eliminated and Phase 3 is significantly faster. In such cases, the unavailable time in the average case and on average across the applications is only about 250 ms, using the same parameters as in Figure 7.

3.3 Overheads

3.3.1 Error-Free Execution

Logging and Parity Maintenance. These operations overlap with useful computation on the processors. They cause performance overhead only through increased contention for memory and the

[1]In reality, this particular error will be detected when the missing node fails to arrive to the barrier when establishing the checkpoint. To conservatively determine the worst-case timing, we ignore this and allow the remaining processors to establish a faulty checkpoint and continue.

[2]Minus those pages already rebuilt because they contained the logs or the data accessed during the rollback phase.

network. In general, the overhead of logging is proportional to the number of lines written between two checkpoints, while the overhead of parity maintenance is proportional to the number of dirty lines displaced from the caches. Consequently, the parity maintenance overhead depends on whether or not the working set of the application fits in the L2 cache: if it does not fit, the resulting frequent write-backs can cause a high overhead of parity maintenance. Finally, note that logging and parity maintenance are performed by the directory controllers and do not significantly affect scalability of the system: adding more nodes to the system results in more logging and parity maintenance, but also adds more directory controllers to perform these operations.

Establishing Global Checkpoints. When it is time to create a new checkpoint, a cross-processor interrupt is delivered to all the processors. This interrupt can be delivered in under 5 μs [24]. Saving the processor's execution context takes little time. Most of the overhead in establishing a new checkpoint comes from writing the dirty cached data back to memory[3]. The time this takes depends on the cache size. Our simulation experiments show this to be on the order of 100 μs for small (128 Kbyte) caches, and 1 ms for larger (2 Mbyte) caches. In Figure 6 we assume 2 Mbyte caches. In the two-phase commit, most of the overhead comes from two global barrier synchronizations, which take up to 10 μs each [10]. Reclaiming the log space only involves moving the log head pointer and a few bookkeeping operations locally performed by each processor, which have negligible overheads. To keep checkpointing overheads small (about 1% of the execution time), the checkpoints in Figure 6 are created once every 100 ms.

Table 2 summarizes the overheads in the error-free execution.

Characteristics of the application's working set	High Checkpoint Frequency	Low Checkpoint Frequency
Does not fit in L2	High Overhead	High Overhead
Fits in L2, mostly dirty	High Overhead	Low Overhead
Fits in L2, mostly clean	Medium Overhead	Low Overhead

Table 2. Effect of application behavior and checkpoint frequency on error-free performance.

3.3.2 Recovery

Recovery. The first phase of recovery is to check the system components to determine what happened and, in case of permanent faults, route around the faulty component. While the implementation of this phase is outside the scope of this paper, its duration has to be taken into account. It has been reported in [29] that the hardware recovery time for Hive/FLASH are about 50 ms for a 16-processor system. This time includes diagnosis, reconfiguration and a reset of the coherence protocol. We assume that a similar hardware recovery can be performed in our system in 50 ms. Phase 2, rebuilding the log pages of the failed node, takes the time proportional to the size of the log, but can be done in parallel by the remaining processors. Our experimental results indicate that in the scenario of Figure 7 this phase takes up to 100 ms, assuming checkpoint frequency of once every 100 ms. In Phase 3 each processor uses the local log to roll back the memory content of its own node. If the log entry is to be restored into a page that is unavailable, that page's parity group is rebuilt first. The time to perform the rollback is proportional to the size of the logs, but also depends on how many lost data pages have to be rebuilt while rolling back. Rebuilding the parity groups of these pages, if it is needed, takes more time than the actual copying of data from the log into these pages. Our experiments indicate that this phase takes up to 490 ms in the scenario presented in Figure 7.

[3]Note that these operations trigger the parity updates and possibly even logging.

Redoing the Work. When an error occurs, rollback recovery restores the system to the checkpoint state that precedes the error. All work performed between that checkpoint and the activation of the rollback has to be re-done. On the average, the lost work performed before the error occurs is half of the checkpoint interval's. Also lost is the work performed until the error is actually detected. If we assume a checkpointing frequency of once every 100 ms and error detection latency of 80 ms, the resulting lost work is 130 ms. The worst case is if the error occurs just before the system establishes a new checkpoint, in which case 180 ms of work is lost.

Overall, assuming the parameters explained above, the machine is unavailable for about 820ms in the worst case. The availability of the machine is $A = (T_E - T_U)/T_E$, where T_E is mean time between errors and T_U is the mean time the machine is not available due to an error. Even assuming $T_E = 1$ day, the resulting availability with ReVive is $A = 99.999\%$. If most errors do not result in losing memory contents, the average unavailable time is only 250 ms, which results in $A = 99.9997\%$ availability.

Rebuilding Lost Memory Pages. Our experiments indicate that, if the lost node had 2GB of memory and $7 + 1$ parity was used, a 16-processor machine requires about 20 seconds to fully rebuild all affected parity groups, if it devotes half of its computation to rebuilding the damaged parity groups and the other half to useful computation. Note that this step is not needed if the error does not result in losing memory contents.

4 ReVive Implementation Issues

ReVive does not require processor or cache modifications. All hardware modifications are confined to the directory controller. Now we discuss these modifications, as well as the possible races.

4.1 Extensions to the Directory Controller

The additional supports required by ReVive are protocol extensions and, optionally, the *L* bit for each directory entry as described in Section 3.2.2.

4.1.1 Protocol Extensions

ReVive requires protocol extensions to perform the parity and log operations described in Sections 3.2.1 and 3.2.2, respectively. These extensions need new transient states in the directory controller entries and new types of messages. The new transient states implement the protocols in Figures 4 and Figure 5. The new messages are the *parity update* message and its acknowledgment. Only the directory controller is affected by these changes – the new messages are communicated between directory controllers and need not be observed by the caches.

4.1.2 Hardware Modifications

Recall that the L bit indicates if the line has already been logged in this checkpoint interval. Using this bit improves performance, but is not needed for the correctness of ReVive. Indeed, without it we simply have to log the previous content of a memory line every time it the line is written back. However, recovery is still possible by restoring the log entries back into memory lines in the reverse order of their insertion into the log.

Since the L bits are optional, we can design the controller so that they are supported inexpensively. For example, if the system has a directory cache, then only the entries in that cache need to have the L bit. When the entry of a line is displaced from the directory cache, its L bit is lost. As a new entry is allocated, the L bit is reset to zero. With this approach, a memory line is occasionally logged multiple times between two checkpoints, but the correctness of ReVive is unaffected.

During the operations on the parity and the log for a line, the line remains in a new transient state in the directory. Once the operations are complete, the line reverts to one of the normal coherence states. Overall, this support requires only some additional storage (at most a few additional bits per directory entry) and does not interfere with the overall design of the directory controller.

4.2 Race Conditions

Most race conditions in our extended protocol can be handled in the same way other similar race conditions are handled in the baseline protocol without ReVive – by serializing accesses to the same memory line and sending negative acknowledgments to avoid deadlocks and livelocks. However, some race conditions are related to error recovery and need to be carefully considered. We identify five classes of race conditions. Four are specific to our protocol, while the fifth one is common to all checkpointing protocols. In the following we assume that D is the checkpoint content of a data line and that D' is the modified content of that data line. Before D is overwritten with D', it is logged into a log entry L'. Creation of L' overwrites some previous memory content L. The parities of D, D', L, and L' are D_p, D'_p, L_p, and L'_p, respectively. Note that if D is lost but D_p is still in memory, memory rebuilding using parity groups will restore D, and vice versa. A similar property holds for D' and D'_p, L and L_p, and L' and L'_p.

Log-Data Update Race. We do not allow any update to data (or its parity) before the log (and its parity) are fully updated. In this way, if an error occurs before L' and L'_p are safely stored, D and D_p are still in memory. If, on the other hand, an error occurs while D' or D'_p are written to memory, L' and L'_p are safely stored and can be used to roll back to D and D_p.

Atomic Log Update Race. Consider an error that results in a partial update of a log entry. It would be a mistake to use such an entry to "restore" the data content. Thus, the log entry is created in the following manner: the log entry is written, followed by a Marker that validates it. Incomplete entries have no valid Markers and are not used for recovery. Similarly, the parity update for the log entry is written before the parity update for the Marker. This prevents an incomplete parity update from being used in a recovery.

Log-Parity Update Race. Consider an error that occurs after log entry L' has been written, but before its parity L'_p is updated. If the L' becomes inaccessible, then the memory where it was written will be rebuilt using L_p into the L state. Because L does not contain a valid Marker for the current checkpoint, it will not be used for recovery. The original checkpoint data D is still unmodified in memory, so no recovery is needed to restore it. Similarly, if the node where L_p is stored becomes inaccessible, L' will be used to restore its memory to the L'_p state. Then the log entry L' will be used to restore D. This operation is unnecessary because the data memory still contains the checkpoint data D. However, it is correct.

Data-Parity Update Race. An error that occurs after the log and its parity have been correctly updated does not compromise recovery, even if the write of D', D'_p, or both is incomplete or not performed at all. This is because the checkpoint content D is found in the log and restored into data memory.

Checkpoint Commit Race. To make sure that checkpoint data from different checkpoints is cleanly separated, we use a variant of the two-phase commit protocol [23]. It is implemented with two barrier synchronizations. Passing the first barrier indicates that all processors have flushed their caches and all resulting memory updates are complete. After the first barrier, each processor marks in the local log that the new checkpoint is established. Then, passing the second barrier means that *all* processors have marked the checkpoint as established. Without the second barrier, it would be possible for a processor X to continue executing before processor Y has marked its checkpoint as established. As a result, data stored in Y's local memory and modified by X would be logged as part of the old checkpoint instead of the new one. After creating the new checkpoint, the log space that is no longer needed can be reclaimed. For example, assume that the error detection latency is such that two checkpoints must be kept. After creating checkpoint N, checkpoint $N-2$ is no longer needed. Therefore, log entries created between checkpoints $N-2$ and $N-1$ can be reclaimed.

5 Evaluation Environment

Architecture. To evaluate ReVive, we use execution-driven simulation. Our simulator is based on an extension to MINT that can model dynamic superscalar processors in detail [9]. The architecture modeled is a CC-NUMA multiprocessor with 16 nodes. Each node contains a processor, two levels of cache, a directory controller, a network interface, and a portion of the main memory of the system (Figure 2). The processor is a 6-issue dynamic superscalar. The caches are non-blocking and write-back. The system uses a full-map directory and a cache coherence protocol similar to that used in DASH [12]. The directory controller is extended to support logging and distributed parity needed for ReVive, as described in Section 3.2. Contention is accurately modeled in the entire system, including the busses, the network and the main memory. Table 3 lists the main characteristics of the architecture.

Applications. We evaluate our scheme using all 12 applications from the Splash-2 suite [31]. These applications are representative of parallel scientific workloads and exhibit a wide variety of sharing and memory access patterns. Table 4 shows the application names and the input sets we used. The data are allocated on the nodes of the machine according to the first-touch policy. This results in local allocation of private data, while shared data are allocated in the memory of the first node that accesses them. Cache sizes of 16kB for L1 and 128kB for L2 are chosen following [31],

Processor	
6-issue dynamic 1GHz	Int,fp,ld/st FU: 5,3,2
Inst. window: 96	Pending ld,st: 16,16

Memory System
L1: 16KB, 2ns hit, 4-way assoc, 64-B line, write back
L2: 128KB, 12ns hit, 4-way assoc, 64-B line, write back
Bus: 100MHz 64-bit quad-data-rate (Like Pentium 4 system bus)
Memory: 100MHz 16-bank DDR, 128 bits wide, 60ns row miss (Essentially, two PC1600 DDR SDRAM modules in parallel)
Dir controller latency: 21ns (pipelined at 333MHz)
Network: 2-D torus, virtual cut-through routing
Message transfer time 30ns + 8ns * # hops
No-contention latency (ns): 2 (L1 hit), 14 (L2 hit), 105 (Local Mem), 191 (Neighbor Mem)

Table 3. Architectural characteristics of the system we model.

to produce representative behavior given the relatively small input sets of Splash-2. The working sets of most Splash-2 applications fit even in relatively small caches [31]. The only exception is Radix, where about 256kB are needed to accommodate the first working set. Only FFT, Ocean, and Radix have important second working sets large enough to overflow our L2 caches. In Radix, we use 4 million keys instead of the default 256 thousand. In FFT, we use 1 million complex numbers instead of the default 64 thousand. These inputs are needed to get a long enough running time, but result in larger working sets for these applications. Because both the first and the second working sets of Radix are larger than our L2 cache, we expect ReVive to exhibit close to worst-case performance on this application.

Application	Problem Size	Total # of Instructions	Global L2 Miss Rate
Barnes	16K particles	1230M	0.05%
Cholesky	tk29.0	1224M	0.26%
FFT	1M points	468M	1.78%
FMM	16K particles	1002M	0.24%
LU	512x512 matrix, 16x16 block	336M	0.07%
Ocean	258x258 grid	270M	2.02%
Radiosity	-test	744M	0.15%
Radix	4M keys, radix 1024	186M	2.51%
Raytrace	car	612M	0.26%
Volrend	head	984M	0.29%
Water-N2	1000 molecules	1074M	0.02%
Water-Sp	1728 molecules	870M	0.02%

Table 4. Characteristics of the applications.

Overheads in Error-Free Execution. The applications simulated have smaller problem sizes and run for shorter periods than real-life workloads. We need to consider how these issues affect the way we model ReVive error-free overheads, namely maintaining logs and parities, and establishing checkpoints.

The overhead of keeping logs and parities is dominated by parity updates, which are both more expensive and more frequent. Parity overhead depends on the rate of write-backs which, to a large extent, is proportional to the cache miss rate. In our simulations with the small problem sizes of Splash-2, we reduced the cache sizes to preserve the cache miss rates. Therefore, the logging and parity overheads that we measure in the simulations should match those that would be observed in a real system.

As for the overhead of establishing checkpoints, most of it is due to writing back all the dirty lines in the caches. This overhead is largely proportional to the size of the L2 cache. Since we use small caches, we can model the overhead in a real system by checkpointing proportionally more often. In Section 3.3.2, we estimated that a real system needs to checkpoint once every 100ms to achieve 99.999% availability when error frequency is once a day. This estimate assumes 2MB L2 caches. According to Section 3.3.1, the time to establish a checkpoint in a system with 128KB L2 caches is an order of magnitude smaller than with 2MB L2 caches. Consequently, our simulated system checkpoints one order of magnitude more frequently – once every 10ms.

To help isolate the overheads of parity updates and log maintenance, we also perform simulations with an infinite checkpoint interval.

Comparison to Commercial Workloads. While ReVive targets commercial, technical, and scientific workloads, the evaluation in this paper does not include commercial loads. We have focused on recovering the computational part of the state of an application. Further work is required to fully flesh out the details when ReVive has to recover in the presence of external network communication and disk activity. These issues have to be addressed to present a fair evaluation of ReVive on commercial workloads. We leave these issues for future work.

Another characteristic of commercial workloads is that they tend to have high miss rates. As a result, ReVive could induce high overheads in error-free execution. In practice, the set of applications used in our evaluation covers a range of miss rates that includes those typically found in commercial workloads. Specifically, the number of L2 misses per 1,000 instructions in our applications ranges from 0.06 in Water-Sp to 6.4 in Ocean and 9.3 in Radix. This range covers typical miss rates in OLTP and other commercial applications. As one example, several web server and OLTP applications have been reported to have around 3 misses per 1,000 instructions [2]. Consequently, ReVive overheads with commercial workloads should not be higher than those we report here.

6 Evaluation

To evaluate ReVive, we examine three issues: overhead in error-free execution, storage requirements, and recovery overhead.

6.1 Overhead in Error-Free Execution

To evaluate the impact of ReVive on error-free execution, we compare ReVive to a baseline system that includes no recovery support. As explained in Section 3.3.1, the sources of performance overhead in error-free execution with ReVive are parity and log updates, and checkpoint generation. For given cache sizes and other machine parameters, the overhead of parity and log updates mainly depends on the characteristics of the application being executed. The overhead of checkpoint generation depends on the frequency of checkpointing. To better understand these overheads, Figure 8 shows the performance overhead of our mechanism using $7 + 1$ parity and with checkpoints performed every 10ms (`Cp10ms`) and with an infinite checkpoint interval (`CpInf`). For comparison, we also show the results of our scheme when mirroring is used instead of parity (as described in Section 3.2.1), for the same checkpoint

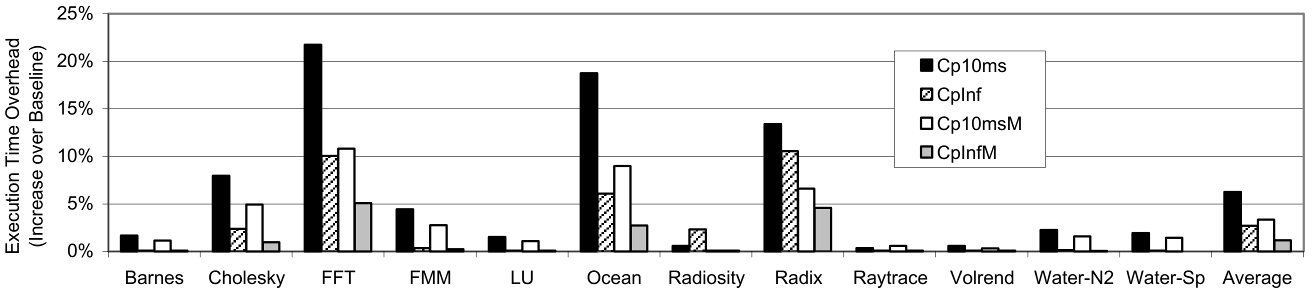

Figure 8. Performance overhead of ReVive in error-free execution.

frequencies: once every 10ms (`Cp10msM`) and with an infinite checkpoint interval (`CpInfM`). The CpInf and CpInfM bars reveal the overheads of logging and parity maintenance with 7+1 parity and mirroring, respectively. The difference between Cp10ms and CpInf, and between Cp10msM and CpInfM, represents the overhead of establishing checkpoints every 10ms, using $7 + 1$ parity and mirroring, respectively.

The average overhead of logging and parity maintenance is low, 2.7% for 7+1 parity (CpInf) and 1% for mirroring (CpInfM). In applications with important working sets that do not fit in the L2 cache (FFT, Ocean, and Radix), this overhead can be high. It reaches 11% in Radix.

The overhead of establishing checkpoints every 10ms is usually small, but it can be relatively high, as in FFT and Ocean. When the checkpoint is established in these applications, almost all lines in their caches are dirty, so the checkpoint takes close to worst case time. In FFT, this effect combines with the high logging and parity maintenance overheads for an overall overhead of 22%, the highest overhead we observe in any of the twelve applications. It is important to note that a checkpoint interval of 10ms is the least favorable end of the spectrum for our scheme. Increasing the checkpoint interval or simply using mirroring instead of parity can reduce the overhead to 10% in FFT. When mirroring is used and the checkpoints are infrequent, the overhead is reduced to 5% on FFT and 1% on the average.

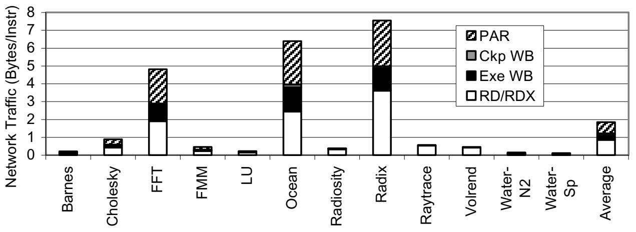

Figure 9. Breakdown of network traffic in Cp10ms.

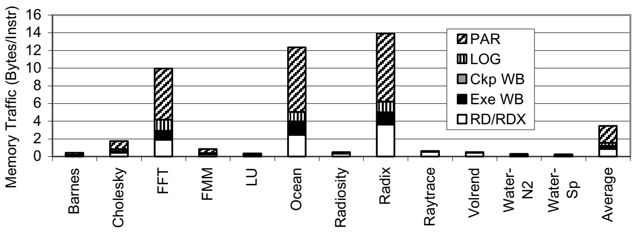

Figure 10. Breakdown of memory traffic in Cp10ms.

ReVive can be designed to be configured at boot time to support parity or mirroring. If the machine is mostly going to run applications that exhibit good caching behavior, the performance overheads are small and parity should be used to reduce the memory space overhead (Section 6.2). For applications with poorer caching behavior, a tradeoff exists between memory space overheads and performance: mirroring is faster but uses more memory. In reality, parity and mirroring need not be used in a mutually exclusive fashion. For example, a small part of the memory can be protected by mirroring, while the rest is protected by parity. Careful allocation of frequently used pages into the mirrored region should result in low overheads, as most of the memory modifications result in mirroring updates, while reducing the memory space overheads, as most of the memory space is uses the efficient parity approach.

To help understand the overheads observed, Figures 9 and 10 show the network and memory traffic in the machine with the `Cp10ms` configuration. The breakdown of the traffic is as follows: `RD/RDX` represents the traffic due to supplying the data on cache misses; `Exe WB` is the traffic due to writing back dirty lines to memory in regular execution; `Ckp WB` is the traffic due to writing back dirty lines when checkpoints are established; `LOG` is the traffic of writing data to the logs; `PAR` is the traffic due to parity updates (for both data and logs). Traffic shown as `RD/RDX` and `Exe WB` is the same as in the baseline system. Traffic shown as `Ckp WB`, `LOG`, and `PAR` is caused by ReVive. If mirroring was used instead of parity, the network traffic would stay the same as in Figure 9; the memory traffic would change only in that `PAR` would shrink to one-third of its size.

Figures 9 and 10 show that, for most of the applications, both the network and the memory traffic are low, without or with ReVive. The exceptions are FFT, Ocean, and Radix, where traffic is already high in the baseline system. For these three applications, the additional traffic, mostly resulting from parity maintenance, further degrades the already poor performance.

6.2 Storage Requirements

ReVive requires additional memory space to store distributed parity and logs.

Parity Storage Requirements. To keep the hardware simple, the number of nodes should be a multiple of the parity group size. In addition, the latter should be a power of two, so that to determine which node has the parity page for a given group, we can use a trivial implementation of the *mod* operation. With $7 + 1$ parity, 88% of the main memory is used for data, while 12% is used for parity. We can reduce this requirement by employing larger parity groups. However, doing so slows down recovery and increases the risk of contention in the home of a parity page belonging to a particularly popular parity group. If mirroring is used instead of parity, the overhead is 50% of the memory.

Log storage requirements. Figure 11 shows the maximum log size for different applications for the Cp10ms configuration, assuming that logs for two most recent checkpoints are kept. As we can see, the largest log is about 2.5MB. With the conservative assumption of a log growing proportionally to the checkpoint interval, that yields 25MB for a checkpoint interval of 100ms. In reality, we expect the actual size to be significantly less, as longer intervals allow more filtering out of redundant log entries (Section 3.2.2).

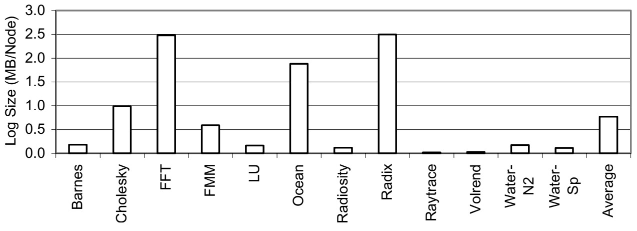

Figure 11. Maximum log size in the Cp10ms configuration.

Overall, if we assume 2GB of DRAM memory per node and a checkpoint interval of 100ms, each node needs 256MB for parity and 25MB for logs, bringing the total memory overhead of ReVive to 14%. Increasing the checkpoint interval to one second would result in up to 25% memory overhead. In comparison, using mirroring instead of parity could result in as much as 62% of memory overhead.

6.3 Recovery Overhead

To estimate the unavailability due to an error, we trigger the error recovery mechanism in each benchmark 8 ms after the second checkpoint in *Cp10ms* is committed. With a checkpoint interval of 100 ms this corresponds to an error that occurs just before the second checkpoint is established, and is detected 80 ms later. As discussed in Section 3.3.2, this results in maximum lost work and maximum ReVive recovery time. Figure 12 shows the resulting ReVive recovery time during which the machine is unavailable (Phases 2 and 3 in Figure 7). The longest such time is 59ms (in Radix), while the average is 17ms. This corresponds to 590ms and 170ms with a 100ms checkpoint interval. After adding 180 ms for lost work and 50 ms for hardware recovery, the resulting unavailable time is 820 ms for Radix and 400 ms on average. If errors occur one per day and all are worst-case node losses, this results in 99.999% availability for Radix and 99.9995% on average.

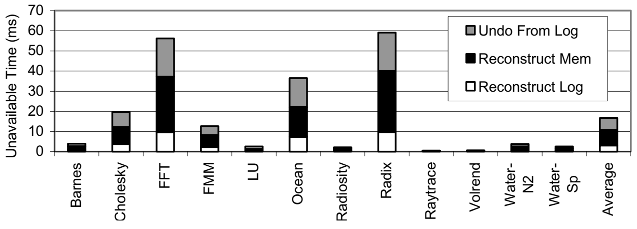

Figure 12. Breakdown of the unavailable time due to an error in the Cp10ms configuration.

7 Related Work

The work most related to our distributed parity mechanism is [20], which implements a software-only checkpointing mechanism where special nodes store parity information. Our work differs from [20] in several important aspects. First, we use hardware to maintain the parity, which significantly reduces the performance overhead. Second, we overlap parity maintenance with useful execution, while [20] performs all parity maintenance while establishing a checkpoint. As a result, the time needed to establish a checkpoint in [20] is a few seconds, instead of a few milliseconds with ReVive. Third, we distribute our parity across the system, rather than keeping it on a few dedicated nodes that can become potential bottlenecks in [20]. Fourth, our parity is updated at a memory line granularity, as opposed to the page granularity used in [20]. Finally, we protect the entire main memory with our parity, rather than just the checkpoint data as in [20]. Protecting the entire memory could make it easier to prevent loss of information about recent external I/O operations when an error occurs.

The work most related to our log-based rollback mechanism is [13], where a snooping device is attached to the bus to intercept write-back and write-miss operations and log previous values of modified memory lines. Our mechanism differs from the one proposed in [13] in several important ways. First, ReVive allows recovery from errors that occur anywhere in the system, while the design in [13] recovers from errors in the processor and cache, as well as from some operating system errors. Second, we use main memory to store the logs, whereas the logs in [13] are stored on their dedicated bus-snooping device. Our approach results in higher flexibility in choosing how much memory is dedicated to logging, while allowing us to store the logs in the cost-effective high-capacity memory modules together with other data. Finally, using hardware-maintained distributed parity with a logging device like that in [13] would be difficult.

Concurrently to our work, a system called SafetyNet that targets some classes of system-wide transient faults has been proposed in [25]. While both ReVive and SafetyNet use log-based rollback mechanisms, Revive differs from SafetyNet in several important ways. First, ReVive enables recovery from permanent faults such as losing a node, in addition to the transient faults that can be tolerated by SafetyNet. Second, ReVive does not require any changes to the processor's caches. In SafetNet, each line in the cache is augmented with a checkpoint number, which is then checked whenever the line is modified by the processor. Furthermore, SafetyNet adds a 256-512KB checkpoint log buffer to the cache. Third, the error detection latency that SafetyNet can tolerate is largely determined by the size of the checkpoint log buffers. In contrast, ReVive uses the main memory to store its logs and, as a result, can tolerate longer detection latencies. Finally, because of ReVive's more general fault model, ReVive causes more network and memory traffic, which may result in larger performance overheads than with SafetyNet.

While we target errors whose effect modifies the system-wide state, other work has targeted errors that can be contained within a single device such as a processor [3, 16, 28, 30]. Our scheme is fully compatible with such mechanisms. The lightweight recovery of a device-specific mechanism would be used for such device-specific errors. Errors whose effect escapes the device and errors not covered by device-specific mechanisms would be recovered using ReVive.

8 Conclusions

This paper presented ReVive, a new cost-effective rollback recovery mechanism for shared-memory multiprocessors. ReVive performs memory-based checkpointing, logging, and distributed parity maintenance without requiring any hardware modification to the processors or caches. ReVive enables recovery from a wide range of system-level errors, including total loss of a node. ReVive's average execution time overhead is only 6.3%, even when establishing checkpoints as often as once every 100ms. Assuming an error detection latency of 80 ms, an error results in up to 820ms unavailable time, including lost work. The resulting availability is better than 99.999% even if errors occur as frequently as once per day. Finally, the main memory space overhead is only 14% of the main memory, and external storage is not used.

The work is being extended in three ways. First, we are examining mirroring support for the most frequently accessed pages and N+1 parity for all other pages in memory, as suggested in Section 6.1. Second, we are evaluating ReVive with commercial workloads and with longer run times to use realistic checkpointing frequencies. Third, we are further developing details of ReVive to support recovery in the presence of I/O activity such as network or disk access. In general, our distributed parity mechanism is a powerful building block that can be used to protect the I/O buffers. In the long term, we plan to combine ReVive with error detection schemes to fully evaluate error recovery.

Acknowledgments

This work is partially based on the ideas from [33]. The authors would like to thank Manohar Prabhu for proposing log updates on read-exclusive and upgrade requests.

References

[1] R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. In *Proc. 20th Intl. Symp. on Fault-Tolerant Computing Systems*, pages 82–88, June 1990.

[2] A. R. Alameldeen et al. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *5th Workshop on computer Architecture Evaluation using Commercial Workloads*, pages 30–38, Feb. 2002.

[3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. 32nd Annual Intl. Symp. on Microarchitecture*, pages 196 –207, Nov. 1999.

[4] M. Banatre et al. An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors. *IEEE Trans. Computers*, 45(10):1101–1115, Oct. 1996.

[5] M. Banatre and P. Joubert. Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. In *Proc. 20th Intl. Symp. on Fault-Tolerant Computing*, pages 89–96, June 1990.

[6] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Trans. Computers*, 41(5):526 –531, May 1992.

[7] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Tech. Rep. CMU-CS-99-148, Carnegie Mellon University, June 1999.

[8] A.-M. Kermarrec et al. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. In *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, pages 289–298, June 1995.

[9] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *Proc. 1998 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 286 –293, Oct. 1998.

[10] R. Kufrin. Barrier Synchronization on the Origin 2000. http://www.ncsa.uiuc.edu/∼rkufrin/projects/CompSci/Barriers/, July 1999.

[11] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. 24th Intl. Symp. on Computer Architecture*, pages 241–251, June 1997.

[12] D. Lenoski et al. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, Mar. 1992. It is Dash, not DASH.

[13] Y. Masubuchi et al. Fault Recovery Mechanism for Multiprocessor Servers. In *Proc. 27th Intl. Symp. on Fault-Tolerant Computing*, pages 184–193, June 1997.

[14] C. Morin, A. Gefflaut, M. Banatre, and A.-M. Kermarrec. COMA: an Opportunity for Building Fault-tolerant Scalable Shared Memory Multiprocessors. In *Proc. 23rd Intl. Symp. on Computer Architecture*, pages 56–65, May 1996.

[15] C. Morin, A.-M. Kermarrec, M. Banatre, and A. Gefflaut. An Efficient and Scalable Approach for Implementing Fault-Tolerant DSM Architectures. *IEEE Trans. Computers*, 49(5):414–430, May 2000.

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proc. 29th Intl. Symp. on Computer Architecture*, May 2002.

[17] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. ACM SIGMOD Intl. Conf. on the Management of Data*, pages 109–116, June 1988.

[18] J. S. Plank et al. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software – Practice and Experience*, 29(2):125–142, Feb. 1999.

[19] J. S. Plank and W. R. Elwasif. Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing*, pages 48–57, June 1998.

[20] J. S. Plank and K. Li. Faster Checkpointing with N + 1 Parity. In *Proc. 24th Intl. Symp. on Fault-Tolerant Computing*, pages 288–297, June 1994.

[21] J. S. Plank, K. Li, and M. A. Puening. Diskless Checkpointing. *IEEE Trans. Parallel and Distributed Systems*, 9(10):972–986, Oct. 1998.

[22] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. Soft. Eng.*, SE-1(2):220–232, June 1975.

[23] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 3rd edition*. McGraw-Hill, 1999.

[24] Silicon Graphics, Inc. REACT^{TM}in IRIX^{TM}6.4 Technical Report. http://www.sgi.com/software/react/react_tr.pdf, 1997.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proc. 29th Intl. Symp. on Computer Architecture*, May 2002.

[26] F. Sultan, T. D. Nguyen, and L. Iftode. Scalable Fault-Tolerant Distributed Shared Memory. In *Proc. Supercomputing 2000*, Nov. 2000.

[27] D. Sunada, D. Glasco, and M. Flynn. Multiprocessor Architecture Using an Audit Trail for Fault Tolerance. In *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, pages 40–47, June 1999.

[28] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. 9th Intl. Conf. on Arch. Support for Prog. Lang. and OS*, Nov. 2000.

[29] D. Teodosiu et al. Hardware Fault Containment in Scalable Shared-Memory Multiprocessors. In *Proc. 24th Intl. Symp. on Computer Architecture*, pages 73–84, June 1997.

[30] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proc. 29th Intl. Symp. on Computer Architecture*, May 2002.

[31] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–38, June 1995.

[32] K.-L. Wu, W. K. Fuchs, and J. H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Trans. Parallel and Distributed Systems*, 1(2):231–240, Apr. 1990.

[33] Z. Zhang. Single system high-availability solutions. Tech. Rep. HPL-2001-81, Hewlett-Packard Laboratories, Apr. 2001.