SafetyNet: Improving the Available of Shared Memory Multiprocessors with Global Checkpoint/Recovery

Daniel Sorin
Milo Martin
Mark Hill
David Wood

Technical Report #1433

December 2001

SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery

Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood

University of Wisconsin—Madison

{sorin, milo, markhill, david}@cs.wisc.edu

Abstract

Availability is increasingly important for shared memory multiprocessors, but the market for commercial servers prefers that availability not come at the cost of appreciably more hardware or a significant degradation in performance. Implementation trends toward less-reliable deep submicron transistors necessitate architectural techniques that increase availability (with the modest impact to performance and cost permitted by the server market). Instead of only relying upon a patchwork of localized fault tolerance schemes (e.g., ECC or DIVA), we seek a unified mechanism for providing system-wide availability.

We develop an availability solution, called *SafetyNet*, that uses a unified, lightweight checkpoint/recovery mechanism to support multiple long-latency fault detection schemes. At an abstract level, *SafetyNet* logically maintains multiple, globally consistent checkpoints of the state of the system (i.e., processors, memory, and coherence permissions), and it can recover to a pre-fault checkpoint of the system and re-execute if a fault is detected. *SafetyNet* uses logical time to efficiently coordinate checkpoints across the system and "logically atomic" coherence transactions to free checkpoints of transient coherence state. Runtime overhead is minimized by pipelining checkpoint validation with subsequent parallel execution (and subsequent checkpoint validation).

We illustrate *SafetyNet* avoiding system crashes due to either misrouted coherence messages or the loss of an interconnection network switch (and its buffered messages). Using full-system simulation of a 16-way multiprocessor running commercial workloads, we find that *SafetyNet* (a) adds statistically insignificant runtime overhead in the common-case of fault-free execution, and (b) avoids a crash when tolerated faults occur.

1 Introduction

Availability becomes increasingly important as internet services are integrated more tightly into society's infrastructure. This is particularly true for the shared-memory multiprocessor servers that run the application services and database management systems (DBMSs) that must robustly manage business data. However, unless architectural steps are taken, availability will decrease over time as implementations use a larger number of increasingly unreliable components in search of higher performance [12, 22, 50]. The high frequencies and small circuit dimensions of future systems will increase their susceptibility to both transient and permanent faults. For example, higher frequencies exacerbate crosstalk [3, 8] and supply voltage noise [46], and smaller devices and wires suffer more from electromigration [53] and alpha particle disruptions [43, 58].

Decades of research in fault-tolerant systems suggest a path toward addressing this problem. Mission-critical systems routinely employ redundant processors, memories, and interconnects (e.g., triple-modular redundancy [26] or pair-and-spare [54]) to tolerate a broad class of faults. However, for many applications, the highly competitive commercial market will seek lighter-weight solutions. For example, RAID level 5 [34]

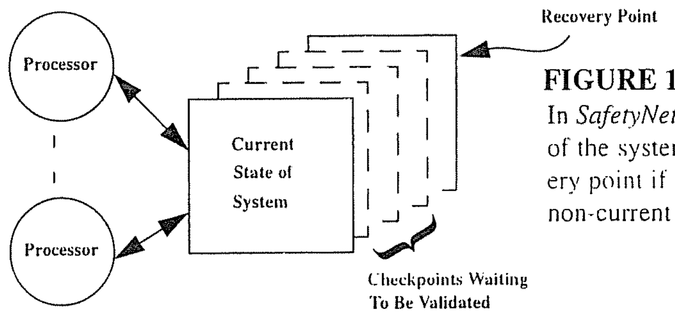

FIGURE 1. ***SafetyNet*** **Abstraction**
In *SafetyNet*, (1) processors operate on the current state of the system, (2) the system can recover to the recovery point if a fault is detected, and (3) some number of non-current checkpoints can be pending validation.

has been deployed widely because its overhead is 1/Nth (for N data disks) rather than the 100% overhead for mirroring. Commercial servers aim for high availability but will accept occasional crashes to improve cost/performance. Software-visible techniques—including database logging and clustering—help preserve data integrity and service availability in these cases.

Current servers employ a range of hardware mechanisms to improve availability. Error correcting codes (ECC), interconnection network link-level retry [19], and duplicate ALUs with processor retry [47] target specific, localized faults such as transient bit flips on memory, links, or ALUs. Computer architects seeking system-wide coverage must integrate a patchwork of localized detection and recovery schemes.

In this paper, we seek a unified, lightweight mechanism that provides end-to-end recovery from a broad class of transient and permanent faults. This recovery mechanism can be combined with a wide range of fault detection mechanisms, including strong error detection codes (e.g., CRCs), redundant processors and ALUs [19, 47], redundant threads [44], and system-level state checkers [9]. By largely decoupling recovery from detection, our approach allows a range of implementations with varying cost-performance.

This paper describes a lightweight global checkpoint/recovery scheme called *SafetyNet*, and we illustrate its abstraction in Figure 1, *SafetyNet* periodically creates a system-wide (logical) checkpoint. *SafetyNet* checkpoints can span thousands or even millions of execution cycles, permitting powerful detection mechanisms with long latencies. After detecting a fault, all processors, caches, and memories revert to and resume execution from a consistent system-wide state, the *recovery point*. *SafetyNet* is a hardware scheme that requires no changes to any software or the instruction set. Moreover, *SafetyNet* has limited impact on the processor, coherence protocol, and I/O subsystem design.

SafetyNet's basic approach is to log all changes to the architected state. This presents three main challenges for a lightweight recovery scheme. First, naively saving previous values before every register update, cache write, and coherence response would require a prohibitive amount of storage. Second, all processors, caches, and memories in a shared-memory multiprocessor must recover to a consistent point. For example, recovery must ensure that all nodes agree on the ownership and data values of each memory block. Third, *SafetyNet* must determine when it is safe to advance the recovery point (i.e., validate a new checkpoint), without degrading performance to wait for slow fault detection mechanisms.

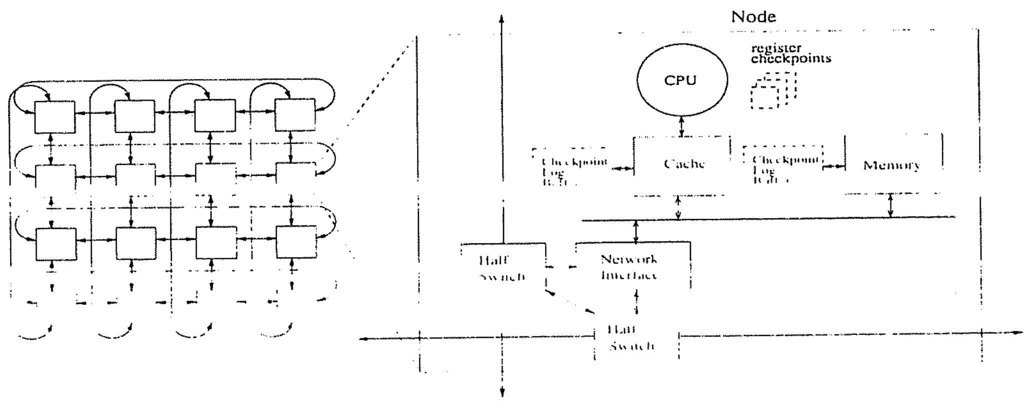

FIGURE 2. Example *SafetyNet* System

SafetyNet efficiently meets these three challenges. First, logging is reduced by checkpointing at a coarse granularity (e.g., 100,000 cycles). Only the first change to a piece of architectural state—register, memory block, or coherence permission—within a checkpoint interval requires a log entry, reducing the log overhead by one or two orders of magnitude. Second, *SafetyNet* efficiently coordinates checkpoint creation using *global logical time* and *logically atomic coherence transactions*, ensuring a consistent recovery point (Section 2). Third, checkpoint validation is pipelined and overlapped with normal execution. Pipelining validation allows *SafetyNet* to tolerate long latency detection mechanisms in the background.

We develop a *SafetyNet* implementation (Section 3) that minimizes runtime overheads for actions in the common case of fault-free execution, including memory operations and coherence transactions. Figure 2 depicts the register checkpoint buffers and Checkpoint Log Buffers (CLBs) added to processor-memory nodes. Register checkpoints, CLBs, caches, and memories are deemed "stable storage" and protected by ECC. As currently defined, *SafetyNet* cannot recover from uncorrectable errors to these structures, which may encourage stronger ECC codes [14]. Future work will address this class of faults, including processor-cache chip kills, but solutions will necessarily trade some performance to provide availability in this case.

SafetyNet is a recovery mechanism that is largely decoupled from any specific fault detection mechanisms. In this paper, we focus on two system-level faults, described in Table 1, that we use as running examples.

(1) **Dropped Message**: A transient fault causes the loss of a coherence message in the interconnect.

(2) **Lost Switch**: A hard fault kills a switch element, irretrievably losing all buffered messages.

Section 5 expands upon the wide variety of faults and detection mechanisms compatible with *SafetyNet*. Like most prior work, we focus on tolerating all single faults, plus coverage for many double faults.

In Section 4, full system simulations with commercial workloads show that, in the common case of fault-free execution, *SafetyNet* does not increase execution time (relative to an unprotected system) by a statistically significant amount. Moreover, *SafetyNet* continues to run after the injection of the two example faults. Recovery time is reduced from a system crash/reboot to a performance "speed bump" of less than one milli-

TABLE 1. Two Example Faults

Dropped Message: This example fault assumes a lost or misrouted coherence message due to a transient environmental condition (e.g., alpha particle [28, 43, 58]). The fault may corrupt the message while it is stored in a switch buffer or by disrupting a switch's internal logic. The fault might be detected using an error detection code (e.g., CRC), by an end-point receiving an illegal message, or by a request timing out. The detection latency may be large in the case of request time out or if strong error detection codes are used (long codes are inherently stronger).
Lost Switch: This example fault assumes the permanent loss of an interconnect switch element (e.g., due to electromigration [53]), causing the loss of all buffered messages. We consider a 2D torus topology that prevents a single point-of-failure by splitting each switch into two half-switches. As illustrated in Figure 2, nodes have separate paths to the north-south and east-west half-switches, providing redundancy in case one half-switch fails. The fault might be detected by the same mechanisms discussed above and diagnosed as permanent by the service processor. Execution may resume after reconfiguring the interconnect to route around the lost switch [15], but with some loss of bandwidth.

second. We also show that 512 kbyte CLBs are large enough, for our commercial workloads, to tolerate fault detection mechanisms with over 100,000 cycles of latency.

In summary, *SafetyNet* seeks a lightweight alternative to traditional fault-tolerant systems (Section 6), by providing efficient support for system-level recovery and long fault detection latencies. By providing a unified mechanism that can tolerate an increasingly important class of transient and permanent errors, we hope to encourage pervasive use of *SafetyNet* in commercial servers.

2 *SafetyNet* Overview

This section presents a high-level overview of *SafetyNet*, discussing how it creates globally consistent checkpoints of the system. Section 3 describes one specific hardware implementation.

2.1 High-Level View

The purpose of *SafetyNet* is to allow the system to recover its state to a consistent previous checkpoint, where a checkpoint includes the state of the processor registers, memory values, and coherence permissions. *SafetyNet* has only a small impact on the underlying cache coherence protocol. We assume a sequentially consistent memory model, and *SafetyNet* does not affect its implementation.

SafetyNet addresses the three challenges for logging schemes that were raised in Section 1. First, *SafetyNet* exploits a coarse checkpoint granularity to reduce the amount of logging (Section 2.2). Second, *SafetyNet* creates consistent global checkpoints (Section 2.3) such that all processors and memories recover to a consistent recovery point upon fault detection (Section 2.5). Third, *SafetyNet* enables pipelined checkpoint validation that is off the critical path and hides the latencies of fault detection mechanisms (Section 2.4).

2.2 Checkpointing Via Logging

Logically, *SafetyNet* checkpoints contain a complete copy of the system's architectural state. For efficiency, *SafetyNet* explicitly checkpoints registers and incrementally checkpoints memory state by logging previous values and coherence permissions. Conceptually, processors and memory controllers log every change to the memory/coherence state (i.e., save the *old* copy of the block) whenever an action (i.e., a store or a transfer of ownership) might have to be undone. To reduce storage and bandwidth requirements, *SafetyNet* only logs the block on the first such action per checkpoint interval. By using coarse checkpoint intervals (e.g., 100,000 cycles), *SafetyNet* significantly reduces logging overhead (evaluated in Section 4.3). Checkpointing of processor register state can be done in many ways, including shadow register copies or writing the registers into the cache.

2.3 Creating Consistent Checkpoints in Logical Time

All of the components (caches and memory controllers) coordinate their checkpoints, so that the collection of local checkpoints represents a consistent global recovery point. Coordinated system-wide checkpointing avoids both cascading rollbacks [16] and an output commit problem [17] for inter-node communication. Checkpoints are coordinated across the system in *logical time* to avoid a potentially costly exchange of synchronization messages.

To ensure that checkpoints reflect consistent system states, the logical time base must ensure that all components can independently determine the checkpoint interval in which any coherence transaction occurs (not just its request). To do so, we exploit the key insight that, in retrospect, a transaction appears logically atomic once it has completed. A transaction's *point of atomicity* occurs when the owner of the requested block processes the request. Figure 3 illustrates how *SafetyNet* determines this point. Note that the requestor does not learn the location of the atomicity point until it receives the response that completes the transaction. To ensure that the system never recovers to the "middle" of a transaction, the requestor does not agree to advance the recovery point until all of its outstanding transactions complete successfully. After completion, the transaction appears atomic, so there is no "middle." Furthermore, by waiting for all outstanding transactions to complete, *SafetyNet* avoids checkpointing transient coherence states and in-flight messages.

Many bases of logical time exist. A simple example in a broadcast snooping system is for each component to count the number of coherence requests it has processed and use that as its logical time. If components create checkpoints every K logical cycles, it is trivial for all components to agree on the interval in which a transaction's request occurred.[1] In this paper, we focus on systems with directory protocols, and thus we

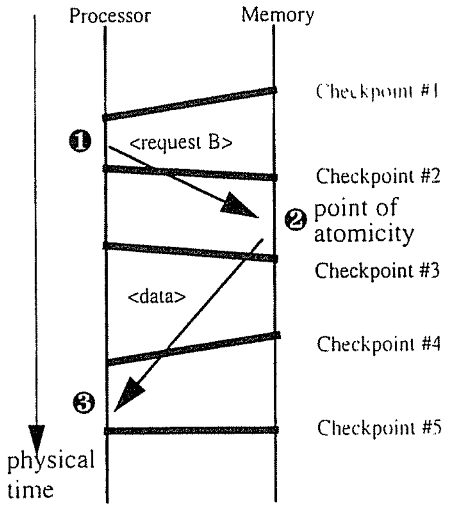

FIGURE 3. Example of Checkpoint Coordination

In this example, physical time flows downwards, and checkpoint lines in logical time are not necessarily horizontal, since logical time is not equal to physical time. Logical time respects causality, so a message cannot be sent in one checkpoint interval and arrive in an earlier interval. At ❶, the processor issues a request for ownership of block B to the memory, which is currently the owner of the block. The memory processes the request at ❷, between checkpoints 2 and 3, and defines the transaction's point of atomicity. In retrospect, the transaction appears to have occurred atomically at this point. A recovery to checkpoint number (CN) 2 or before would restore ownership to the memory. A recovery to CN 3 or later would maintain ownership at the processor. A recovery to CN 2-5 (the duration of the transaction) is not possible until after the transaction, since the processor would not validate any of these checkpoints until the transaction completed successfully at ❸.

need a different logical time base. If we could distribute a perfectly synchronous physical clock, we would have a viable logical time base in which logical and physical time are the same. In Section 3, we relax this requirement by deriving a logical time base from a loosely synchronized (in physical time) *checkpoint clock*.

2.4 Validating Checkpoints and Deallocating Checkpoint State

Checkpoint validation is the process of determining which checkpoint is the recovery point. Processors and memories coordinate checkpoint validation so that all components recover to the same checkpoint number on a recovery. Coordination can be pipelined and performed in the background, off the critical path. For example, checkpoint number 3 (CN3) can be validated only if every component agrees that it could be the recovery point, i.e., all execution prior to CN3 was fault-free. For a checkpoint interval to be fault-free, every transfer of ownership in that interval must complete successfully, by which we mean that the data was transferred fault-free to the receiver. Once every component has independently declared that it has received fault-free data in response to all of its requests in the interval before the checkpoint, the recovery point can be advanced. At this point, all transactions prior to this checkpoint have had their points of atomicity determined. After validation, state for the previous recovery point can be deallocated lazily.

Validation latency depends on fault detection latency, since a checkpoint cannot be validated until it has been verified fault free. For our transient and hard fault examples, the detection latency can be as long as the requestor's timeout latency. Timeout latency can be many traversals of the interconnect, plus some slack built in for contention delays. Adding to validation latency, validation cannot occur until all nodes have coordinated their validations, and this involves an exchange of messages. Since validation latency is long, it is important for *SafetyNet* efficiency that it be performed in the background and off the critical path.

Checkpoint validation also determines when the system can interact with the outside world of I/O devices. The output commit problem [17] requires that only validated, fault-free data can be communicated outside of the sphere of recovery. For example, the system cannot communicate unvalidated data with the disks if the effects of this communication cannot be undone through recovery. The standard solution is to delay all output events until a validated checkpoint. Implementing I/O with InfiniBand (www.infinibandta.org) is a good match for *SafetyNet*, because I/O is setup in memory and then committed with a "doorbell ring." *SafetyNet* would need to delay only the doorbell ring, which should be acceptable to many types of I/O (e.g., to disks and the Internet).

2.5 Recovering the System to a Consistent Global State

If a fault is detected, *SafetyNet* restores the globally consistent recovery point. The recovery point represents the consistent state of the system at the *logical time* that this checkpoint was taken. Recovery itself requires that the processors restore their register checkpoints and that the caches and memories unroll their logs to

1. SMPs do not need to be synchronous, i.e., a request does not need to arrive at every node at the same time. Thus, an SMP with this logical time base could have skew in logical time between nodes.

recover the system to the consistent state at the pre-fault recovery point. All state associated with transactions in progress at the time of recovery can be discarded, since this state is, by definition, unvalidated state that occurs logically after the recovery point. After recovery, the system reconfigures, if necessary, and resumes execution from the recovery point. For the lost switch example, reconfiguration involves routing around the faulty switch. For the dropped message example, no reconfiguration is necessary.

3 A *SafetyNet* Implementation

In this section, we will discuss one particular implementation of the *SafetyNet* abstraction. This implementation reflects the goal of incurring low overhead in the common case of fault-free execution, while not optimizing the rare case of recovery.

3.1 System Model

Figure 2 illustrates a single node, containing a CPU, a cache[2], and a portion of the system's shared memory. A *Checkpoint Log Buffer (CLB)*, associated with each cache and memory controller, stores logged state. The system's multiple nodes communicate through a 2D torus interconnection network to implement a cache coherence protocol. The coherence protocol in this paper is based on a typical MOSI directory protocol[3], and *SafetyNet* has only a slight impact on it. The system also includes redundant system service processors (which exist in many servers, such as the UltraEnterprise E10000 [10]), which help coordinate advancement of the recovery point as well as system restart after recovery.

3.2 Logical Time Base

As discussed in Section 2, checkpoints are coordinated across the system in logical time. For our system with directory-based coherence, we use a loosely synchronous (in physical time) *checkpoint clock* that is distributed redundantly to ensure no single point of failure. On each edge of this clock, each component creates a checkpoint and increments its *current checkpoint number (CCN)*. While it might be difficult to distribute a synchronous clock across a system with near-zero skew, it is not nearly so difficult to distribute one with the same frequency and some amount of skew between nodes. As long as the skew between any two nodes is less than the minimum communication time between these nodes, the checkpoint clock is a valid base of logical time, since no message can travel backwards in logical time.[4]

We use logical time to address the primary challenge in coordinating checkpoints across a system, which is keeping checkpoints consistent with respect to memory and coherence state. All components must agree, for every coherence transaction, in which checkpoint interval that transaction occurred. Assigning a transaction

2. For ease of exposition, we assume a single level cache, but we have implemented *SafetyNet* with multiple levels.
3. In this paper, we assume a directory protocol and a 2D torus, but we have also implemented *SafetyNet* on top of a system with a broadcast snooping protocol and a totally ordered interconnect.
4. Otherwise, the following inconsistency could arise. Consider the case in which processor P1 has a CCN of 3 and sends a request to the owner, P2, while P2's CCN is still 2. Thus, checkpoint 3 would appear to include the reception of the request but not the sending of the request!

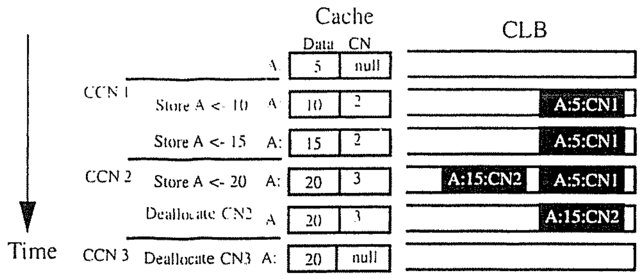

FIGURE 4. Logging at the Cache

to a checkpoint interval is protocol-dependent, and it is the only significant difference in implementing *SafetyNet* on top of different classes of protocols (i.e., directory vs. snooping). In a directory protocol, the point of atomicity occurs when the owner of the block (either the directory or a processor) processes the request.[5]

3.3 Logging

SafetyNet uses *Checkpoint Log Buffers (CLBs)* to incrementally checkpoint memory state. Logically, *SafetyNet* writes a memory block to a CLB whenever an *update-action* (i.e., store or transfer of ownership) might have to be undone in the case of a recovery. Since caches perform stores and both caches and memories can transfer ownership of blocks, each of these components has a CLB. Except during recovery, CLBs are write-only and off the critical path.

SafetyNet only logs a block on the first update-action per checkpoint interval. To detect this case, *SafetyNet* adds a *checkpoint number (CN)* to each block in the cache, denoting to which checkpoint it belongs. On each update-action, *SafetyNet* (1) compares the component's current checkpoint number (CCN) with the block's CN, (2) logs the block, if necessary, (3) updates the block's CN to CCN+1, and (4) performs the update-action. Blocks must be logged to the CLB if CCN >= CN. For example, a store by a processor with CCN=3 to a block with CN=4 need not be logged. Blocks with null CNs have not been written or transferred lately, and they implicitly belong to the recovery point as well as all subsequent checkpoints. Having CNs on blocks enables logic to determine whether logging of a store or ownership transfer would be redundant.[6] Figure 4 illustrates an example of logging at a cache. In Appendix A, we discuss efficient ways to store and manipulate CNs at the caches.

When giving up ownership of a block, a component performs logging (as described above) and then sends the block *with the updated CN* to the requestor. This policy follows from a key insight from Wu et al. [57]: a transfer of ownership is just like a write, in that we cannot be sure that it will not be undone by a recovery. Consider the case where block ownership is transferred with its CN set to 3 (i.e., the sender's CCN is 2) and the receiver wishes to then perform a store to it while its CCN is still 2. Logging is unnecessary, since the receiver was not the owner at checkpoint 2. This is the same as if the owner of a block with CN=3 performed a store to it while its CCN is still 2.

5. Note that the *ordering point* in a directory protocol is different, and it occurs when the directory processes the request.

6. There are other optimizations for reducing logging due to attaining ownership, but they are less important

The CLBs can be sized for performance and not correctness, since *SafetyNet* can avoid situations in which the CLB fills up. Even when it appears that an entry must be logged in the CLB, logging can be avoided at the cost of some performance. In the case of store overwrites, we can throttle requests from the CPU. In the case of coherence ownership transfers, we can negatively acknowledge (nack) coherence requests.

3.4 Checkpoint Creation

Checkpoint creation is kept simple, since it is a common-case event that occurs on each edge of the checkpoint clock. A processor checkpoints its non-memory architectural state (i.e., registers) and increments its CCN.[7] A memory controller simply increments its CCN. Checkpointing of memory and coherence state is achieved through logging, so no checkpointing of that state is necessary at checkpoint creation.

Checkpoint creation policy is simply choosing a suitable checkpoint clock frequency, f_c. As f_c decreases (given a constant number of outstanding checkpoints), *SafetyNet* can tolerate longer fault detection latencies. For example, we allow four outstanding checkpoints and choose f_c equal to 10 MHz (i.e., the checkpoint interval is 100,000 processor cycles at a processor clock of 1 GHz) to enable 400,000 cycles (0.4 msec) of detection latency tolerance. The cost of increasing tolerable detection latency is more storage at the CLBs. While decreasing f_c allows for more compression of logged data, since only the first of multiple writes or ownership transfers in a checkpoint interval requires logging, total CLB storage is a function both of logging frequency and interval length. The value of f_c has little effect on common-case performance, since *SafetyNet* adds little overhead, as will be shown in Section 4. The choice of f_c has a greater impact on recovery latency, as will be discussed in Section 4.2.

3.5 Checkpoint Validation and Deallocation of Checkpoint State

Checkpoint validation requires that all components agree that execution up until that checkpoint was fault-free. A cache controller only agrees to validate a checkpoint once every transaction it initiated in the interval before that checkpoint completed successfully. A directory controller only agrees once every transaction for which it forwarded a request to a processor owner (i.e., 3-hop transaction) completed successfully. Thus, the requestor must send an acknowledgment to the directory after its request has been satisfied, so that the directory can deallocate its buffer entry for the transaction. Any lost message will prevent recovery point advancement. If the recovery point cannot be advanced after a given amount of time, the system assumes an error has occurred (such as a lost message) and triggers a system recovery. We coordinate validation with a 2-phase scheme. Once every component has informed the service processor that it is ready to advance the recovery point, the service processor broadcasts the new *recovery point checkpoint number (RPCN)*.[8] Exe-

7. Since checkpoint numbers are encoded in a finite number of bits, say k, we can only have 2^k active checkpoints. CN wraparound can only occur if validation ceases (i.e., because a coherence transaction does not complete) while checkpoint creation continues. We avoid wraparound by choosing a request timeout latency that is shorter than the latency to wraparound. Thus, a request would timeout before it could stall validation to the point at which wraparound could occur.

8. Communication of coordination messages (which are infrequent) can be made reliable through redundancy, if this double fault model is to be tolerated. We will discuss this issue in Section 5.

cution does not slow down while checkpoints are validated in the background, similar to a fuzzy barrier [23].

Processor and memory controllers deallocate a checkpoint by discarding their now unneeded architectural checkpoint. A processor discards its register checkpoint. In the caches, a checkpoint is deallocated by clearing the CN of all blocks that had CN set to the newly deallocated checkpoint. Logged data at the cache and memory CLBs from this checkpoint is discarded.

3.6 System Recovery and Restart

If a component detects a fault, it triggers a recovery. The recovery message, which includes the RPCN, is broadcast (redundantly) by the service processor, and all nodes proceed to recover to the recovery point. The process of recovery involves several steps, and it leverages the insight that any transactions in progress, by definition, belong to unvalidated checkpoints that can be discarded. First, the interconnection network is drained, and all state related to coherence transactions that were in progress at the time of the recovery are discarded. Second, processors, caches, and memories recover the RPCN checkpoints. Memories just sequentially undo the actions in their CLBs. Processors restore their register checkpoints. Caches invalidate all blocks written or sent in an unvalidated checkpoint interval (i.e., blocks with non-null CNs), and they undo the logged actions in their CLBs.

After recovery and reconfiguration (if needed), a restart message is broadcast to inform the nodes that they can resume operation. The restart cannot begin until every node has finished its recovery. As with coordination to validate checkpoints, we implement a 2-phase coordination where every node informs the system service processor once it is ready to restart and then the service processor broadcasts the restart message.

3.7 Summary of Implementation

We have developed one particular implementation of the *SafetyNet* abstraction. The implementation addresses the three challenges that were raised for logging schemes. First, we exploit checkpoint granularity to reduce the amount of logging necessary. Second, we efficiently coordinate checkpoints across the system in a logical time base that is loosely tied to physical time. Third, we enable checkpoint validation to be performed in the background, thus hiding the potentially length latency of fault detection. We now quantitatively evaluate our design with full-system timing simulation and commercial workloads.

4 Evaluation

In this section, we evaluate *SafetyNet*. We begin in Section 4.1 by describing our methodology. Then, in Section 4.2, we quantitatively determine *SafetyNet* performance by running three experiments in which we compare the performance of *SafetyNet* versus that of an unprotected system. Lastly, in Section 4.3, we perform sensitivity analyses on the amount of cache bandwidth and CLB storage that *SafetyNet* uses.

TABLE 2. Target System Parameters

Cache	4 MB, 4-way set associative
Memory	2 GB, 64 byte blocks
Miss From Memory	180 ns (minimum, uncontended, 2-hop)
Checkpoint Log Buffer	512 kbytes total, 72 byte entries
Interconnection Network	2D torus, link bandwidth = 1.6 GB/sec
Checkpoint Interval	100,000 cycles = 100 μsec

4.1 Methodology

We simulate a 16-processor target system with the Simics full-system, multiprocessor, functional simulator [31], and we extend Simics with a memory hierarchy simulator to compute execution times. We evaluate *SafetyNet* with four commercial workloads and one scientific workload.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB that can boot unmodified commercial operating systems and run arbitrary unmodified applications. We use Simics/sun4u, which can simulate Sun Microsystems's SPARC V9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot an unmodified copy of Sun Solaris 8. Simics is a functional simulator only, and it assumes that each instruction takes one simulated cycle to execute (although I/O may take longer), but it provides an interface to support detailed memory hierarchy simulation.

Processor Model. We use Simics to model a processor core and L1 cache that, given a perfect memory system, would execute four billion instructions per second and generate blocking requests to the L2 cache and beyond. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. While an out-of-order processor model might have an impact on the absolute values of the results, it would not qualitatively change them (e.g., whether a crash is avoided). For evaluating processor/cache overhead for checkpointing register state, we model a conservative latency of 100 cycles.[9] We conservatively charge eight cycles for logging store overwrites (8 bytes/cycle for 64 byte cache blocks), but these are only about 0.1% of instructions.

Memory Model. We have implemented a memory hierarchy simulator that supports a directory protocol, not unlike that of the SGI Origin, with and without *SafetyNet* support. The simulator captures all state transitions (including transient states) of our coherence protocols in the cache and memory controllers. We model a 2D torus interconnection network contention within this interconnect, including contention due to validation coordination messages. In Table 2, we present the design parameters of our target memory system. With a checkpoint interval of 100,000 cycles and four outstanding checkpoints, *SafetyNet* can tolerate fault detection latencies up to 400,000 cycles (0.4 msec at 1GHz). To exercise the protocol implementation, we drove it for billions of cycles with a random tester that injected faults and stressed corner cases by exploiting false

9. If checkpointing was a more frequent event (e.g., if we were using *SafetyNet* to support speculation), we could optimize register checkpointing latency by using shadow register copies.

TABLE 3. Workloads

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are 8 simulated users per processor. We warm up for 10,000 transactions, and we run for 50 transactions
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system and includes driver threads to generate transactions. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 10,000 transactions.
Static Web Server: We use Apache 1.3.19 (`www.apache.org`) for SPARC/Solaris 8, configured to use pthread locks and minimal logging as the web server. We use SURGE [6] to generate web requests. We use a repository of 2,000 files (totalling ~50 MB). There are 10 simulated users per processor. We warm up for ~80,000 requests, and we run for 500 requests.
Dynamic Web Server: Slashcode is based on a dynamic web message posting system used by `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com`, and it contains ~3,000 messages. A multithreaded driver simulates browsing and posting behavior for 3 users per processor. We warm up for 240 transactions, and we run for 5 transactions.
Scientific Application: We use *barnes-hut* from the SPLASH-2 suite [55], with the 16K body input set. We begin measurement at the start of the parallel phase to avoid measuring thread forking.

sharing and reordering messages [56]. We simulate each design point multiple times with small, pseudo-random perturbations of memory latencies to cause alternative OS scheduling paths, and error bars in our results represent one standard deviation in each direction.

Workloads. Commercial workloads are an important workload for high availability systems. As such, we evaluate *SafetyNet* with four commercial applications and one scientific application, described in Table 3.

4.2 Experiments

We perform three experiments to evaluate *SafetyNet* performance and show their results in Figure 5. For each of our five workloads, we plot five bars: two bars for systems unprotected by *SafetyNet* and three bars for systems with *SafetyNet*.

Experiment 1: Fault-Free Performance. In this experiment, we run two systems, *SafetyNet* and unprotected by *SafetyNet*, in a fault-free environment. In Figure 5, the first and the third bars (from the left) for each workload reflect the normalized performances of the unprotected system and *SafetyNet*, respectively. We observe that the two systems perform statistically similarly for all workloads. Intuitively, *SafetyNet* does not impact common case actions, such as loads and stores that do not require logging. Overheads due to register checkpointing (every 100,000 cycles) and stores that require logging (0.1% of all instructions) are negligible, and back pressure due to filling up the CLBs is rarely needed. We present sensitivity analysis of CLB sizing in Section 4.3.

Experiment 2: Dropped Messages. In this experiment, we periodically inject transient faults into the system by dropping a message every billion cycles (i.e., once per second). The requestor times out on its request and triggers recovery. The second "bar" reflects the unprotected system performance (crash). The fourth bar from the right represents *SafetyNet* behavior, and we see that it is statistically similar to the fault-free sce-

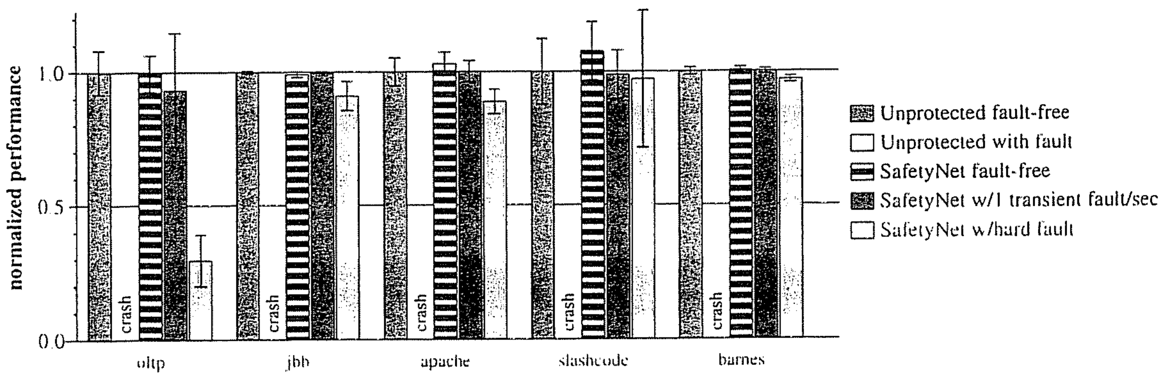

FIGURE 5. Performance Comparison of *SafetyNet* with an Unprotected System

nario. Sensitivity analysis (not shown) involved injecting faults even more frequently, and we discovered that they start to impact performance at the frequency of one per million cycles (i.e., every millisecond).

The exact recovery latency is not critical, since *SafetyNet*'s recovery latency is orders of magnitude shorter than the latency of crashing and rebooting (while preserving data integrity). Recovery latency consists of discarding unvalidated checkpoint state, restoring the state from the recovery point, re-configuring (e.g., changing the routing to avoid a dead switch) if necessary, and re-executing the work that was lost between the recovery point and the fault. Re-executing lost work is the dominant factor, since the recovery point can be hundreds of thousands of cycles in the past. *SafetyNet* can tolerate longer fault detection latencies with less frequent (i.e., larger) checkpoints, but it does so at the cost of more potential lost work. Nevertheless, even a one million cycle recovery latency is still only one millisecond (i.e., much shorter than a crash).

Experiment 3: Lost Switch. In this experiment, we inject a hard fault into an interconnection network switch after 5 million cycles, killing the half-switch and dropping its buffered messages. The second "bar" reflects the crash of the unprotected system. The fifth bar reflects *SafetyNet* performance, and we observe that, most importantly, *SafetyNet* avoids a crash. Its performance is degraded, with respect to the fault-free scenario, due to the restricted post-fault bandwidth.

4.3 Sensitivity Analyses: Cache Bandwidth and Storage Cost

Cache Bandwidth. *SafetyNet*'s additional consumption of cache bandwidth depends on the frequencies of stores that require logging. These stores consume additional cache bandwidth for reading out the old copy of the block. Logging due to transferring cache ownership, however, does not incur additional bandwidth, since the cache line must be read anyway. In Figure 6, for the OLTP workload, we plot this frequency as a function of the checkpoint interval. Both the x and y axes are log scale. Distinguishing between all stores and only those stores that require logging, we notice the striking dropoff in the latter as the checkpoint interval increases. These trends are the same for the other workloads, and the intuition explaining this is that spatial and temporal locality reduce the number of distinct blocks touched per checkpoint interval. For a checkpoint interval of 100,000 cycles, only 2-3% of stores (less than 0.1% of all instructions) require logging. In

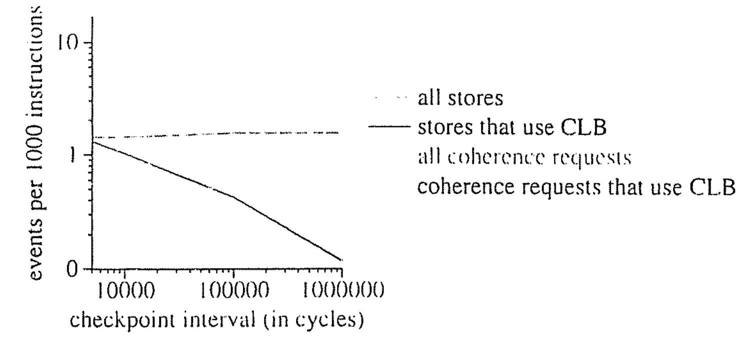

FIGURE 6. Frequencies of Stores and Coherence Requests (OLTP Workload)

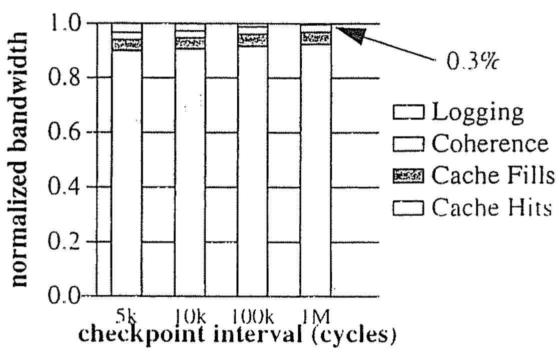

FIGURE 7. Bandwidth vs. Checkpoint Interval (OLTP Workload)

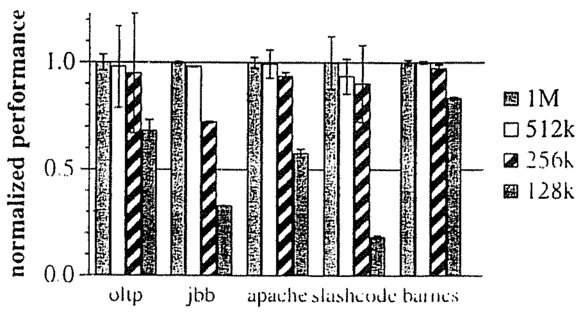

FIGURE 8. Performance vs. CLB Size

Figure 7, we plot the percentage of cache bandwidth used by cache hits, cache fills, responding to coherence requests, and logging due to store overwrites. The additional cache bandwidth used by *SafetyNet* ranges from 0.3% for million cycle intervals up to 4% for short 5,000 cycle intervals.

Storage Cost. An implementation of *SafetyNet* seeks to size the CLBs to avoid performance degradation due to full CLBs. Total CLB storage is proportional to the number of allowable checkpoints and the number of entries per checkpoint. We allow for four checkpoints and a CLB entry is 72 bytes (8-byte address and 64-byte data block). The number of entries per checkpoint corresponds to logging frequency which is, in turn, a function of the frequencies of stores and coherence requests. Figure 6 shows that, on average, only about 100-150 CLB entries are created per 100,000 instructions (although the variance in this rate requires more storage). In Figure 8, we plot the performance of *SafetyNet* as a function of CLB size. While 512 kbyte and 1 Mbyte CLBs produce statistically equivalent performances across the workloads, 256 kbyte CLBs degrade the performances of *jbb* and *apache*, and 128 kbyte CLBs degrade the performances of all of our workloads.

5 Tolerating Other Faults with *SafetyNet*

To this point, this paper has explained how SafetyNet can enable a recovery after the detection of a lost message or lost switch fault. Most generally, SafetyNet can enable recovery for any fault that does not corrupt ECC-protected architectural state, provided that:

- a system can be augmented with a mechanism to detect the fault (or sign off on its absence),

- and faults are detected while *SafetyNet* still maintains a fault-free recovery point.

SafetyNet can maintain a recovery point as long as necessary, in the worst case, by stalling execution. However, fault-free performance is best if, in the average case, fault detection mechanisms validate checkpoints fault-free in one or a few checkpoint intervals (e.g, in 100,000 cycles or 0.1 milliseconds).

The rest of this section considers some additional faults that *SafetyNet* could tolerate.

Interconnection Network Faults. A typical interconnection network fault model focuses on link errors, trying to detect single, double, or burst errors. Link errors are normally detected with error detecting codes (EDC), such as parity, SECDED, or cyclic redundancy check (CRC) [15, 35, 39]. Current systems [19] use short codes (e.g., on 8 or 16 bytes), since the code must be checked before the data is forwarded or used. *SafetyNet* permits the use of longer, and inherently stronger codes [36] because of its ability to tolerate long fault detection latencies. *SafetyNet* is also compatible with other fault models such as lost and misrouted messages (e.g., detected with time-outs), corrupted internal switch state (e.g., detected with internal EDC), and switch controller malfunction (e.g., detected with internal consistency checks).

Coherence Protocol Faults. There are numerous soft faults in the protocol engine that can be tolerated with global checkpoint/recovery. This class of faults includes sending the wrong message or sending duplicate messages, as well as faults in the reception of messages. *SafetyNet* also could be used to recover from design faults in the protocol, if they could be detected reliably [9, 18] and would not keep recurring after recovery (leading to livelock).

Cache Hierarchy and Memory Faults. Fault tolerance schemes for memory, both SRAM and DRAM, are already well-established, and we present the fault model and prior detection techniques for completeness. Detecting faults in storage cells can be accomplished with error detecting codes. A system with *SafetyNet* has to protect the cache hierarchy and memory with ECC, since they contain memory blocks that could potentially be the only valid copies in the system, so an uncorrectable fault could be unrecoverable. Memory chip kills can be tolerated by using a RAID-like scheme for DRAM [14].

Processor Core Faults. Processor faults can be detected with numerous schemes, including parity, redundant ALUs, and redundant threads [44, 42, 49]. Localized recovery schemes, including DIVA [4], can tolerate processor faults, but *SafetyNet* provides a unified mechanism that tolerates these faults, as well as others.

***SafetyNet* Hardware Faults.** The *SafetyNet* hardware itself is also susceptible to faults. Most faults in the *SafetyNet* hardware only manifest themselves during a recovery, which implies a double fault situation. While our fault model targets single faults, we discuss a few simple techniques for tolerating this specific class of double errors. The processors' checkpoints of their architectural states are protected with ECC, since an error in this state is unrecoverable if we have to restore a checkpoint. Also, the mechanism for communicating messages regarding checkpointing (e.g., a message telling each node to recover) must tolerate faults. We assume a redundant transmission of these messages over the existing interconnection network. One possibility is time redundancy, in which a message is sent multiple times, possibly over different paths. Triple modular redundancy (TMR) with voting can mask a corrupted or lost message in any of the redundant transmissions. Performance is not critical for these messages, but reliable delivery is crucial.

6 Related Work

Related research exists in fault tolerance, as well as in logging for speculation and versioning of data. Prior work in fault tolerance can be classified into two broad categories: backward error recovery (BER) through checkpointing or logging and forward error recovery (FER) through redundant hardware. Among other differences, the evaluation of *SafetyNet* also advances previous work in fault tolerance by using full-system simulation of commercial workloads.

Hardware Backward Error Recovery. In BER schemes, the state of the system is checkpointed periodically (or differences are logged), and a fault is tolerated by recovering to a previously checkpointed state. IBM mainframes [24, 47] have long used register checkpoint hardware and store-through caches to recover from processor and memory system errors, respectively. The CARER scheme [25] for uniprocessors uses a normal cache with a writeback update policy to assist rapid rollback recovery. This scheme is integrated with the cache controller, checkpointed system state is maintained in main memory, and checkpoints are established whenever a modified cache block needs to be replaced. Ahmed et al. [1] extend CARER for multiprocessors by synchronizing the processors whenever any of them need to take a checkpoint. Wu et al.'s [57] multiprocessor extension of CARER allows a processor to write into its private cache between checkpoints. Checkpointing, which flushes all modified blocks, is performed when ownership of a block modified since the last checkpoint changes. *SafetyNet* is more efficient, since it does not checkpoint before every ownership transfer. The Sequoia computer system [7] uses private caches to hold state between checkpoints. The memory holds the consistent (checkpoint) state, and all dirty cache blocks are flushed to the main memory at every checkpoint. Banâtre et al. [5] describe a scheme that is identical to a normal bus-based SMP except that the traditional memory module has been replaced by an RSM (Recoverable Shared Memory) module. RSM requires a shadow copy of the entire memory as well as a mechanism for maintaining the inter-processor dependence graph to establish consistent recovery points.

Software Backward Error Recovery. Software checkpointing has also been done, but at radically different engineering costs. Tandem machines prior to the S2 (e.g., the Tandem NonStop) use checkpointing, where every process periodically checkpoints its state on another processor [45]. If a processor fails, its processes are restarted on the other processors that hold the checkpoints. Condor [30], a batch job management tool, can checkpoint jobs in order to restart them on other machines. Work by Plank [37, 38] and Wang and Hwang [52, 51] uses software to periodically checkpoint applications for purposes of fault tolerance. These schemes differ from each other primarily in the degree of support required from the programmer, libraries, and the operating system. *SafetyNet* differs from these works in that it is a hardware solution.

(Hardware) Forward Error Recovery. FER schemes use redundant hardware to mask faults. For example, redundant processors [4, 26, 27, 54] or redundant threads within a processor [49] can be used to mask processor faults. Redundant paths through adaptive networks allow packets to be routed around faults [13, 15]. There are many industrial systems that use FER. The Intel 432 [27] uses replication of VLSI components (i.e., commodity parts) to achieve a range of fault tolerance needs. The Stratus [54] computer system uses two pairs of processors to mask faults. Within each pair, the two processors compare results—if the results do not match, a fault has been detected and the other pair is now responsible. The Tandem S2 [26] uses triply modular redundant (TMR) processors to mask faults. Slipstream [49] is a lighter-weight processor FER scheme that can use redundant threads within a processor to mask faults. DIVA [4] uses a checker processor to implement FER on the processor (but not on the system).

Speculation and Versioning of Data. Prior research for supporting speculation has logged changes in state that is local to a given node [20, 41]. *SafetyNet*'s logging is logically similar, although *SafetyNet* must also log transfers of ownership in our global checkpoint/recovery scheme. Researchers have used data versioning in the context of sequential semantics. Databases use versioning to maintain serializability [33]. Speculative multithreading schemes use versioning to implement sequential program semantics [2, 11, 21, 32, 40, 48]. Our goal differs in that we superimpose checkpoints on system execution with *parallel semantics*, to support availability. We use globally consistent checkpoints rather than local checkpoints at different places in a sequential execution.

7 Conclusions and Future Work

In this paper, we develop a scheme, called *SafetyNet*, that improves the availability of shared memory multiprocessors. We describe an implementation of *SafetyNet*, and we demonstrate that it adds little performance overhead and has reasonable storage costs. In developing *SafetyNet*, this paper makes three contributions which allow *SafetyNet* to be efficient in the common case of fault free execution.

- *SafetyNet* adds no latency to the common case of 99.9% of all instructions.
- *SafetyNet* hides the latency of fault detection by pipelining the validation of checkpoints. The system can continue to execute while it determines if old checkpoints can be validated.
- *SafetyNet* maintains memory and coherence checkpoint state in place within the cache hierarchy instead of copying it to memory and/or disk.

We see interesting avenues for future work. First, we will use *SafetyNet* to tolerate many of the faults discussed in Section 5 by developing suitable detection mechanisms. Since *SafetyNet* provides recovery for long-latency detection mechanisms, we can focus on stronger, high-latency codes and signatures. Second, we will use *SafetyNet* to tolerate harder faults, such as the loss of architectural state in a processor-cache chip kill. However, this alternative design will achieve this higher level of fault-tolerance for increased overheads in time, space, and/or cost. Finally, we will use *SafetyNet* to tolerate selected hardware design errors by developing mechanisms for detecting such errors and addressing forward-progress issues. Notably, some of these detection mechanisms can use system-wide communication to detect violations of global invariants (e.g., in the coherence protocol), since *SafetyNet* efficiently supports long-latency detection mechanisms.

Acknowledgments

We thank the following people for their helpful comments and suggestions on this work: Alaa Alameldeen, Peter Hsu, Carl Mauer, Ravi Rajwar, Craig Zilles, and the Wisconsin Industrial Affiliates. This work is supported in part by the National Science Foundation with grants EIA-9971256, CDA-9623632, and CCR-0105721, IBM Graduate Fellowship (Martin), Intel Graduate Fellowship (Sorin), two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Compaq Computer Corporation, Intel, IBM Corporation, and Sun Microsystems.

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 *Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems*, pages 82–88, June 1990.

[2] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[3] R. Anglada and A. Rubio. An Approach to Crosstalk Effect Analyses and Avoidance Techniques in Digital CMOS VLSI Circuits. *International Journal of Electronics*, 6(5):9–17, 1988.

[4] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[5] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin. An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, Oct. 1996.

[6] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

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

[8] M. Bohr. Interconnect Scaling - The Real Limiter to High Performance. In *Proceedings of the International Electron Devices Meeting*, pages 241–244, Dec. 1995.

[9] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001. In conjunction with ISCA.

[10] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[11] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[12] B. Colwell. Maintaining a Leading Position. *IEEE Computer*, pages 45–47, Jan. 1998.

[13] W. J. Dally, L. R. Dennison, D. Harris, K. Kan, and T. Xanthopoulos. Architecture and Implementation of the Reliable Router. In *Proceedings of 2nd Hot Interconnects Symposium*, Aug. 1994.

[14] T. J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Microelectronics Division Whitepaper, Nov. 1997.

[15] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. IEEE Computer Society Press, 1997.

[16] E. Elnohazy, D. Johnson, and Y. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, Sept. 1996.

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

[18] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, Jan. 1984.

[19] M. Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34–39, Jan/Feb 1997.

[20] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[21] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, Feb. 1998.

[22] G. Grohoski. Reining in Complexity. *IEEE Computer*, pages 41–42, Jan. 1998.

[23] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Apr. 1989.

[24] R. Gustafson and F. Sparacio. IBM 3081 Processor Unit: Design Considerations and Design Process. *IBM Journal of Research and Development*, 26:12–21, Jan. 1982.

[25] D. Hunt and P. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.

[26] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.

[27] D. Johnson. The Intel 432: A VLSI Architecture for Fault-Tolerant Computing. *IEEE Computer*, pages 40–48, Aug. 1984.

[28] T. Juhnke and H. Klar. Calculation of the Soft Error Rate of Submicron CMOS Logic Circuits. *IEEE Journal of Solid-State Circuits*, 30(7):830–834, July 1995.

[29] D. D. Lee and R. H. Katz. Using Cache Mechanisms to Exploit Nonrefreshing DRAM's for On-Chip Memories. *IEEE Journal of Solid-State Circuits*, 26(4):657–66, Apr. 1991.

[30] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, Computer Sciences Department, University of Wisconsin–Madison, Apr. 1997.

[31] P. S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[32] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University, May 1997.

[33] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[34] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of 1988 ACM SIGMOD Conference*, June 1988.

[35] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[36] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, 1972.

[37] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

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

[39] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[40] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, July 2001.

[41] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[42] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[43] J. Robertson. Alpha Particles Worry IC Makers as Device Features Keep Shrinking. *Semiconductor Business News*, October 21, 1998.

[44] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[45] O. Serlin. Fault-Tolerant Systems in Commercial Applications. *IEEE Computer*, pages 19–30, Aug. 1984.

[46] K. Seshan, T. Maloney, and K. Wu. The Quality and Reliability of Intel's Quarter Micron Process. *Intel Technology Journal*, Sept. 1998.

[47] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.

[48] J. G. Steffan and T. C. Mowry. The Potential for Using Thread Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, Feb. 1998.

[49] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[50] M. Tremblay. Increasing Work, Pushing the Clock. *IEEE Computer*, pages 40–41, Jan. 1998.

[51] Y. M. Wang, E. Chung, Y. Huang, and E. Elnozahy. Integrating Checkpointing with Transaction Processing. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems*, pages 304–308, June 1997.

[52] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, pages 22–31, June 1995.

[53] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing Co., 1982.

[54] D. Wilson. The Stratus Computer System. In *Resilient Computer Systems*, pages 208–231, 1985.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[56] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, Aug. 1990.

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

[58] J. Ziegler et al. IBM Experiments in Soft Fails in Computer Electronics. *IBM Journal of Research and Development*, 40(1):3–18, Jan. 1996.

Appendix A: Cache Implementation Details

In this appendix, we describe how to store and manipulate checkpoint numbers at the caches. Cache operation is conventional, with three important exceptions: (1) a store hit may trigger logging of the block that would be overwritten, (2) a validation of checkpoint *i* must find blocks with CN=*i* and then set CN=*null*, and (3) a recovery must invalidate blocks with CN≠*null*. Since we always recover to RPCN, there are no partial rollbacks. Case (1) can be detected by comparing the processor's CCN and the stored block's CN in parallel with a standard tag comparison. A store to the cache thus reads the cache tags (but not data) before writing it, but this is also true for normal stores, since they require a tag lookup (like updating the LRU bits).

Checkpoint validations and system recoveries can be made to operate globally on the caches in constant time with two changes. First, we store checkpoint numbers encoded as 1-hot bit vectors. This encoding requires *k* bits to support *k* active checkpoints, which is not a problem for the small *k* we envision (e.g., four). Second, we keep the checkpoint numbers in the same SRAM with the cache tags and augment the cache tags with a flash clear on each CN bit column, similar to the mechanism in caches with flash invalidation [29].