Fingerprinting: Bounding
Soft-Error Detection Latency and Bandwidth

Jared C. Smolens, Brian T. Gold, Jangwoo Kim,
Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk
Computer Architecture Laboratory
Carnegie Mellon University
http://www.ece.cmu.edu/~truss

ABSTRACT

Recent studies have suggested that the soft-error rate in microprocessor logic will become a reliability concern by 2010. This paper proposes an efficient error detection technique, called *fingerprinting*, that detects differences in execution across a dual modular redundant (DMR) processor pair. Fingerprinting summarizes a processor's execution history in a hash-based signature; differences between two mirrored processors are exposed by comparing their fingerprints. Fingerprinting tightly bounds detection latency and greatly reduces the interprocessor communication bandwidth required for checking. This paper presents a study that evaluates fingerprinting against a range of current approaches to error detection. The result of this study shows that fingerprinting is the only error detection mechanism that simultaneously allows high-error coverage, low error detection bandwidth, and high I/O performance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and servicability; B.8.1 [**Performance and Reliability**]: Reliability, testing, and fault-tolerance

General Terms

Performance, Design, Reliability

Keywords

Soft errors, error detection, dual modular redundancy (DMR), backwards error recovery (BER)

1. INTRODUCTION

Technology trends will raise soft-error rates in microprocessors to levels that require changes to the design and implementation of future computer systems [12, 17, 23]. Detecting soft-errors in the processor's core logic presents a more difficult challenge than errors in storage or interconnect devices, which can be handled via error detecting and correcting codes. Today, microprocessor systems requiring reliable, available computation employ dual modular redundancy (DMR) at various levels to enable detection—ranging from replicating pipelines within the same die [25] to mirroring complete processors [15, 21]. In this paper, we offer an investigation of the general design space of error detection in DMR processor configurations.

Our study of DMR error detection is motivated by a larger effort to develop a cost-efficient reliable server architecture in the TRUSS project.[1] To exploit the economy of scale, modern servers are increasingly built from commodity modules. In particular, rack-mounted server "blades" are emerging as a cost-effective and scalable approach to building high-performance servers. The TRUSS architecture aims to provide reliability in a commodity blade cluster environment with *minimal* changes to commodity hardware and *without* modifications to the application software. The TRUSS architecture is designed to survive any single component failure (e.g., memory device, processor or system ASIC) by enforcing *distributed* redundancy at all levels of processing and storage. For example, under TRUSS, redundant program execution will be carried out by mirrored processors on different nodes interconnected by a system area network. This distributed redundancy approach means DMR error detection must be implemented under the constraints of limited communication bandwidth and non-negligible communication latency between the mirrored processors.

In this paper, we examine four design points for DMR error detection. The designs are differentiated by the monitoring points where the behavior of two mirrored processors is compared. The first three alternatives reflect previously proposed techniques. These three designs correspond to comparing the mirrored processors at 1. *chip-external* interface, 2. *L1 cache interface* and 3. *full state* of program execution. We introduce a fourth option based on the comparison of *fingerprints*. A fingerprint is a cryptographic hash value computed on the sequence of updates to a processor's architectural state during program execution. A simple fingerprint comparison between the mirrored processors effectively verifies the correspondence of all executed instructions covered by the fingerprint. Fingerprinting drastically reduces the required data exchange bandwidth for DMR error detection.

We evaluate the four DMR error detection design points

---

[1] Total Reliability Using Scalable Servers.

in conjunction with a backward-error recovery (BER) framework that restarts execution from a checkpoint in the presence of an error. The results of our evaluation offer the following key insights. First, as the soft-error rate increases, traditional chip-external error detection requires a minimal checkpoint interval of tens of millions of instructions to maintain the reference mean-time-to-failure target (MTTF) of 114 FIT [4]. Second, DMR error detection at the L1 cache interface or on the full state of two mirrored processors requires an unacceptably high interprocessor bandwidth. Third, the I/O traffic in commercial OLTP systems forces a checkpoint interval that is too short to provide adequate error coverage with chip-external and L1 cache interface error detection. This combination of results argues that fingerprinting is the only viable error detection mechanism that simultaneously allows high-error coverage, low error detection bandwidth, and high I/O performance in a DMR processor system.

Paper Outline. Section 2 continues with further details of the DMR error detection design space and evaluation metrics for this study. Section 3 presents the concepts and principles of fingerprinting. Section 4 describes our experimental setup and data analysis. Sections 5-7 present and discuss the results of our study. Section 8 describes related prior work. Finally, Section 9 provides a summary and our conclusions.

2. DESIGN SPACE OVERVIEW

Recovering from soft-errors in program execution can be accomplished in two general ways. With forward-error recovery (FER), enough redundancy exists in the system to determine the correct operation, should a processor fail. Triple modular redundancy (TMR) is the classic example of FER: three processors execute the same program and when one processor fails a majority vote determines the correct state [24]. For cost-sensitive commercial systems, the added overhead associated with a TMR system is prohibitive [18].

In this paper, we evaluate backwards-error recovery (BER) mechanisms that create checkpoints of correct system state and rollback processor execution when an error is detected. Dual modular redundancy (DMR) is a BER technique where two redundant processors are used to detect errors in execution. We use DMR as the base design for evaluating soft-error detection mechanisms and their implications for system reliability and performance.

We assume a DMR processor system where two processors execute redundantly in lockstep. We assume the chosen processor microarchitecture is fully deterministic such that in an error-free scenario, the two processors behave identically. An error in the execution of one processor manifests as a deviation in the behavior of the two processors.

In this section, we first give an overview of the checkpointing process that we assume. We then introduce the error detection design space for DMR processor systems and finally present the three key criteria we use to evaluate and compare the DMR error detection design points.

2.1 Checkpointing

A checkpoint of program state consists of a snapshot of architectural registers and memory values. Although a checkpoint logically represents a single point in time, our model consists of copying the register file at checkpoint creation and using a copy-on-write mechanism to maintain a change
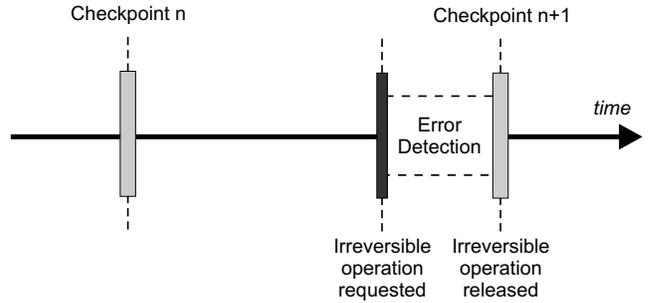

Figure 1: The error detection and checkpoint comparison timeline for a single processor. The redundant processor (not shown) precisely replicates the above timeline.

log for caches and memory (similar to that proposed in SafetyNet [27]). Rollback consists of restoring register and memory values from the log.

The time between checkpoints is referred to as the checkpoint interval. For short intervals (hundreds to tens of thousands of instructions), checkpoints are small enough to fit in on-chip structures [7, 27]. With periodic checkpoints, the storage can be guaranteed to fit on-chip. If longer intervals are required, checkpoints must be stored off-chip.

To recover from errors, the checkpoint and error detection mechanisms must be synchronized. Figure 1 illustrates the sequence of actions for the checkpoint and error detection mechanisms assumed in this paper. Immediately before an operation with irreversible effects (an operation that cannot be re-executed, such as uncached loads and stores), any latent errors should be detected and corrected by recovering to the last checkpoint. If no error is detected, the irreversible operation can then be released and a new checkpoint must be taken. If the detection mechanism does not find all latent errors before discarding the previous checkpoint, the new checkpoint may contain erroneous state and successful recovery will no longer be possible.

2.2 Error Detection

We consider four DMR error detection options differentiated by the monitoring points where the behavior of the two processors are compared (illustrated in Figure 2).

Chip-External Detection. The least intrusive approach is to monitor and compare the two processors' external behavior at the chip pins. For this paper, we conceptualize observable chip-external behavior as all address and data traffic exiting the lowest level of on-chip cache. There is an inherent error detection latency associated with this approach: the effect of an error originating in the execution core may not appear at the output pins for some time due to buffering in registers and the cache hierarchy. The exact error detection latency is a program- and architecture-dependent property. Nevertheless, detecting and containing an error at the pins is still effective in preventing the error from propagating to the rest of the system (e.g., memory, disks, network controller) with irreversible effects.

L1 Cache Interface Comparison. An alternative to chip-external detection is to monitor and compare the address and data traffic entering the interface of the L1 cache. This
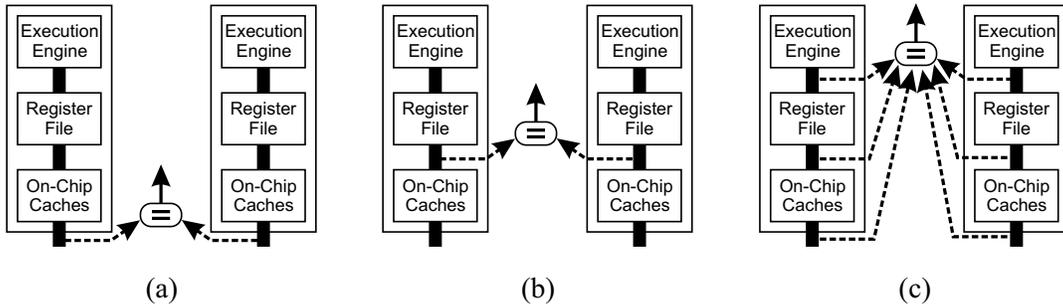
Figure 2: Error detection mechanisms: (a) chip-external, (b) L1 cache interface, (c) full state. Fingerprinting (not shown) provides detection capabilities equivalent to full state comparison, (c).

is similar to the approach taken in SRT [19] and CRT [16]. To support this comparison, we assume our DMR processor system has a dedicated channel connecting the two processors' internal cache interfaces. One of the issues we address in Section 6 is the data exchange bandwidth required by this approach. At the cost of higher DMR data bandwidth, this internal monitoring point significantly reduces the error detection latency because no errors can be buffered in the cache hierarchy. The detection latency remains nonzero, however, because an erroneous value can still be buffered in the register file until a store instruction eventually propagates the value into the cache.

Full-State Comparison at Checkpoint Creation. At the time a checkpoint is created, the complete set of changes to architectural state can be compared between mirrored processors. Naturally, this mechanism uses more pin bandwidth than the passive chip-external, but the checkpoint can be guaranteed error free. The number of unique cache lines or memory pages changed since the previous checkpoint determines the amount of work necessary for full-state comparison. As the checkpoint interval increases, we expect spatial locality to amortize the comparison cost over the interval. However, if available bandwidth is insufficient, full-state comparison may require the processors to stall until the error detection completes.

Fingerprint Comparison. A *fingerprint* is a hash value (computed using a linear block code such as CRC-16) that summarizes several instructions' state updates into a single value. The mirrored processor pairs can corroborate the effects of multiple instructions through a single fingerprint comparison. When used with checkpointing, all instructions between two checkpoints will be summarized by a single 16-bit fingerprint (assuming CRC-16). Starting at the earlier checkpoint, a fingerprint is accumulated for all instruction updates until the next checkpoint is created. If the fingerprints computed by the mirrored processors agree at this point, all of the fingerprinted instructions since the last checkpoint are known to be correct. If the fingerprints of the mirrored processors disagree, the computation must be restarted from the earlier checkpoint. Fingerprinting offers two important advantages. First, fingerprinting addresses the prohibitive bandwidth requirement of comparing all architectural state updates before they are committed. At the same time, fingerprinting provides a summary of all state changes, including any possible soft-errors.

2.3 Evaluation Metrics

In this paper, we evaluate and compare the different DMR error detection mechanisms under three key criteria, namely error coverage, DMR error detection bandwidth, and I/O performance.

Error Coverage. The error coverage of a particular fault-tolerant system is the fraction of errors that can be successfully detected and corrected. In a rollback-recovery system, a checkpoint taken after a soft-error has occurred, but before detection, leads to an unrecoverable failure. Hence, error coverage is determined by the probability that a checkpoint contains an undetected error. As the checkpoint interval increases, an error has a greater chance of being detected before the next checkpoint, since the average instruction is now further from the checkpoint time. Conversely, with smaller checkpoint intervals there is less time to detect an error before the next checkpoint is taken.

If all architectural state updates are compared before a checkpoint is taken, the error coverage is independent of the checkpoint interval because errors in the current interval will be exposed by the detection mechanism. The error detection latencies of chip-external and L1 cache interface comparisons, however, place a restrictive lower bound on acceptable checkpoint interval. In this paper, we show that the lower bound for checkpoint interval with chip-external detection is 10 to 25 million instructions; for L1 cache interface detection the minimum interval is 10 to 100 thousand instructions.

Error Detection Bandwidth. The detection bandwidth required to compare the behavior between a mirrored processor pair is critical to the overall system's feasibility and implementation cost. While the chip-external comparison approach places no additional data on the chip's pins, the other three techniques we consider create additional traffic exiting the chip. Our bandwidth requirement study in Section 6 will show L1 cache interface detection and full-state comparison both require bandwidth exceeding the capability of current packaging technologies.

I/O Performance. Conventional rollback-recovery mechanisms log all input from external devices and delay all output until it is guaranteed that the system will not be rolled back across a committed I/O operation [6, 29]. In software-assisted checkpointing of I/O-intensive applications, buffering can hide the performance impact of delaying I/O to the

Table 1: Evaluating rollback-recovery mechanisms.

Checkpoint Interval	Detection Mechanism	Error Coverage	B/W	I/O
Large	Chip-External	✓	✓	
	L1 Cache Interface	✓		
	Full Comparison at Checkpoint	✓	✓	
Small	Chip-External		✓	✓
	L1 Cache Interface			✓
	Full Comparison at Checkpoint	✓		✓
	Full Comparison w/ *Fingerprinting*	✓	✓	✓

end of a coarse-grained checkpoint (millions of instructions). However, a hardware solution must create checkpoints immediately after an I/O operation to prevent rollback that might reissue past I/O operations. When saving architectural state immediately after accessing a device, we require an error detection mechanism with high detection coverage to ensure that rollback is not necessary. In Section 7, we show that I/O intervals in commercial OLTP systems are often in the range of 100s to 1000s of instructions and fine-grained checkpointing is necessary.

From the above discussion, there are contradicting demands on the checkpoint interval by I/O performance and error coverage requirements. Table 1 summarizes the opposing factors when trying to balance I/O performance, error coverage, and comparison bandwidth in choosing a DMR detection mechanisms. Only fingerprinting, presented in detail in the next section, can simultaneously satisfy all requirements of high error coverage, low detection bandwidth, and high I/O throughput.

3. FINGERPRINTING

In this section, we first discuss the properties of a fingerprint in detail. We then explore the options available in an implementation of fingerprinting.

3.1 Fingerprinting Overview

Fingerprints provide a concise view of past and present program state in order to detect differences in execution across two redundant processor dies, without the overhead normally associated with full state comparison. A fingerprint must provide high coverage of errors for all instructions in the checkpoint interval, require little comparison bandwidth, and work with a variety of checkpoint intervals.

The fingerprint contains a summary of the outputs of any new register values created by each executing instruction, new memory values (for stores), and effective addresses (for both loads and stores). This set of updates is both necessary and sufficient to capture errors in architectural state; errors

in other architectural and microarchitectural structures will quickly propagate to those being fingerprinted. Examples of structures outside the scope of fingerprinting include decoded instruction bits, condition code values, the current program counter, and internal microarchitectural state.

Fingerprint comparisons are forced at the end of every checkpoint interval. A matching fingerprint indicates that the computation in both processors, from the beginning of the checkpoint to the current instruction, is identical. At this point, the old checkpoint will be replaced with a new one and the process starts again.

The hash used to compute fingerprint values should be a well-constructed, linear-block code. There are two key requirements for the code. First, the code must have a low probability of undetected errors. Second, the code should be small for both easy computation and low bandwidth comparison. Hamming codes are a class of well-known codes with an understood lower bound on the probability of an undetected error and a low storage overhead. For a p-bit hamming code, the probability of an undetected error is at most 2^{-p} [35]. This lower bound is independent of the number of updates to the code and represents the probability of detecting an error given an infinitely long sequence of uniformly random bits.

Given this statistical bound, we choose a 16-bit cyclic redundancy check (CRC) for our evaluation that has a 0.999985 probability of detecting an error. In Section 5, we show that this CRC is sufficient to achieve acceptable system reliability when using fingerprinting as an error detection mechanism over the useful range of checkpoint intervals.

3.2 Fingerprint Implementation

Hardware implementations of hash mechanisms are well known and readily found in the literature [24]. In this section, we study the implications of fingerprinting on the processor pipeline.

Microarchitecture alternatives. Two major alternatives exist when implementing fingerprinting in a speculative superscalar out-of-order processor pipeline, as illustrated in Figure 3. First, we can capture all committed state by taking the values from instructions retiring from the reorder buffer (ROB) and load-store queue (LSQ). Conventionally, the ROB does not contain instruction results. Rather than trying to read the instruction results from the register file, we believe it would be more cost-effective to add the instruction results to the ROB.

Alternatively, if the cost of adding instruction results to the ROB is too high, another approach would be to fingerprint microarchitectural state by hashing instruction results as they complete. Figure 3(b) shows this design. In this case, the fingerprint will contain a hash of committed state plus additional, potentially speculative state.

Precise Replication. Implicit in our description of fingerprinting is that redundant processors must now have precisely replicated behavior. For example, suppose a redundant pair of nodes are in a spin-lock, waiting for another processor pair to release ownership of a shared variable. The two nodes must receive ownership at exactly the same (logical) time, such that they terminate waiting after the same number of loop iterations.
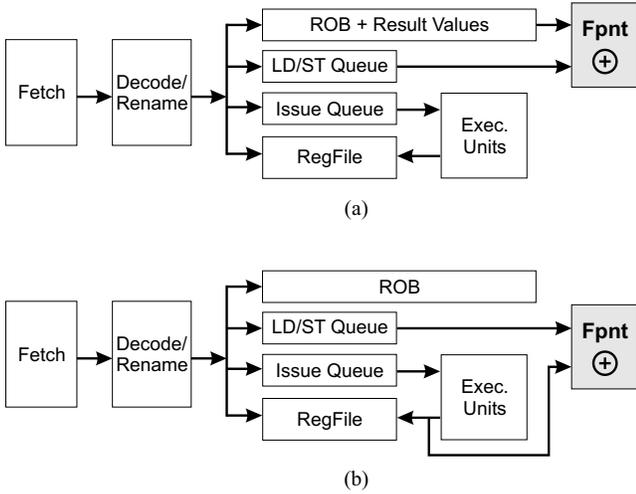
Figure 3: Microarchitectural implementation options: (a) committed state (b) committed + speculative state.

Most systems operate the processors in lockstep with a (small) fixed delay between them [15, 21]. Additionally, we require that all microarchitectural structures be deterministic so that, given identical input streams, replicated processors execute the same instruction sequence. This requirement exists for chip-external detection as well; otherwise, correlating replicated processor results would be impossible.

4. ANALYSIS TECHNIQUES

In this section, we present the dynamic dependency analysis algorithms used to measure the error detection latency in a processor with caches for detection mechanisms at the chip-external and L1 cache interfaces. Then, we present a method for estimating the bandwidth required to compare full architectural state across redundant processors. Finally, we summarize our simulation environment.

4.1 Dynamic Dependency Analysis

Soft-errors propagate from an initial, affected instruction to a detection point through program dataflow. We measure error coverage by finding the minimum error detection latency (counted in instructions) from execution to detection for each instruction. The minimum detection latency of a particular instruction is obtained by generating dynamic dependency graphs (DDGs) and finding the distance to the first detection opportunity for the data value produced by that instruction. A DDG is a directed, acyclic graph defining a partial ordering of instructions as related through register and cache dependencies [3].

For chip-external detection, we consider three classes of detection opportunities: load/store addresses, register-indirect jumps, and cache writebacks. If an address used for a load or store is in error, we assume that it will cause a cache miss and be detected outside the chip (i.e., a direct comparison with a fault-free processor would not show this address request). An erroneous register value used for a register-indirect jump will likely cause an instruction cache miss, a TLB miss, or an exception. Any of these will result

in near-immediate detection outside the chip. Finally, cache writeback values and addresses are explicitly compared in the external detection schemes.

For L1 cache interface detection, we consider three classes of detection opportunities: load/store addresses, register-indirect jumps, and store values. Each of these classes requires communication between the processor and L1 cache, which will be directly compared using this detection method.

DDG Construction. The DDGs are constructed by annotating in-order program execution traces with register-register dataflow, load/store addresses, cache miss, and writeback information to track the lifetime of values in the register file and cache. A post-simulation analysis tool then traverses the traces and builds graphs from data dependencies. The dependencies flow through the registers and cache to capture the paths that an erroneous instruction result may take, including being stored, read from the cache, and reused in another instruction.

The analysis tool traverses backwards from the end of the trace. When a detection opportunity occurs (e.g. cache writeback or load/store), a new DDG is constructed to find all instructions leading to the present detection. Each new instruction is added to the most recently-constructed DDG containing one or more dependent instructions. Each instruction belongs to one and only one DDG, indicating the earliest point of detection.

Figure 4 shows a simple example of the trace analysis graph construction. The trace contains instructions, a timestamp (instruction count), and cache access information. In this simple example, two graphs are shown for the two detection points (instructions 2 and 6). As an instruction i is added to its graph, the instruction distance between the detection opportunity and i is written to a log.

Analysis Limitations. This error analysis makes several simplifying assumptions. First, we treat all transient faults as equally likely to result in an architecturally-visible error. As discussed by Mukherjee, et al. [17], this assumption ignores the fact that certain structures are more likely to result in architectural errors. An instruction queue, for example, may cause many more architecturally-visible errors than the functional units performing computation.

Second, by not injecting actual faults in the system, we ignore the effects of logical masking on error detection latency. However, logical masking only extends the error detection latency if an erroneous instruction result is masked in one chain of instructions, but used in another, longer chain.

Third, we assume all errors in address computation miss in the on-chip cache and that the effects of control flow errors are detected immediately. Because the size of the on-chip cache is miniscule when compared to the entire address space, we assume errors in address computation cause a miss in the cache that is immediately detected. Similarly, errors in control flow will most likely cause an immediate instruction cache miss that is visible outside the chip.

Finally, the instruction traces used in the analysis are of a finite length, which artificially shortens cache block dead times (time between the last cache line modification and writeback). All of these limitations makes our analysis conservatively favor existing detection mechanisms; we estimate a shorter detection latency than may actually exist.
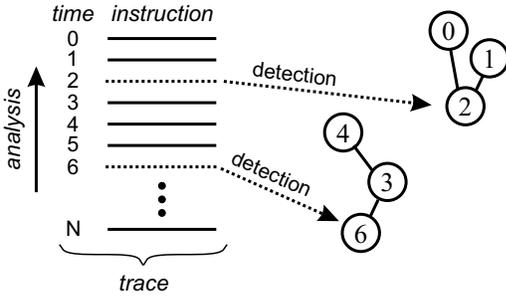
Figure 4: An instruction trace and the associated Dynamic Dependency Graphs (DDGs), with detection points at instructions 2 and 6.

4.2 Comparison Bandwidth

To measure the comparison bandwidth needed by an error detection mechanism, we use the same traces collected for the dynamic dependency analysis. For chip-external detection, the bandwidth required includes all L2 load/store miss addresses and addresses/data for writebacks. When detecting errors at the L1 cache interface, the comparison bandwidth includes all loads/stores addresses and stores. Both chip-external and L1 cache interface detection bandwidth are independent of the checkpoint interval.

Comparing the full processor state at the checkpoint requires bandwidth proportional to the amount of memory changed during the checkpoint interval and inversely proportional to the checkpoint interval itself. For example, if 1MB of memory is changed during a checkpoint interval of 10 milliseconds, then we must be able to compare at a rate of 100MB/s to avoid any performance loss, assuming comparisons can be overlapped with execution. We evaluate the bandwidth requirements for this detection class by counting the unique cache lines written during a checkpoint interval.

The bandwidth required for fingerprint comparison is dependent only on the checkpoint interval and size of the fingerprint register, which we assume is 16 bits. Because fingerprints are compared before every checkpoint, the bandwidth required is simply 2 bytes per checkpoint interval.

4.3 Methodology

We simulate 26 SPEC2K applications using SimpleScalar [5] and two commercial workloads using Virtutech Simics [14]. In SimpleScalar, we use a functional cache simulator for the Alpha ISA and simulate CPU-intensive workloads using the host platform for system calls. Simics is a full system functional simulator that allows functional simulation of unmodified commercial applications and operating systems on the SPARC v9 architecture.

In SimpleScalar, we simulate the first input set for all 26 SPEC2K benchmarks. For each benchmark, we simulate up to eight pre-determined 100-million instruction regions from the benchmark's complete execution trace, using the prescribed procedure from SimPoint [22]. In Simics, we boot an unmodified Solaris 8 operating system to run two commercial workloads: a 40 warehouse TPC-C like workload with IBM's DB2 and SPECWeb. The TPC-C like workload [30] consists of a 40 warehouse database striped across five raw disks and one dedicated log disk with 100 clients. The SPECWeb workload [28] services 100 connections with Apache 2.0. Both commercial workloads are warmed un-

til the CPU utilization reaches 100% and, in the case of the DB workload, until the transaction rate reaches steady state. Once warmed, the commercial workloads execute for 500 million instructions. In both models, the processor is assumed to run at 1GHz, with a constant IPC of 1.0. In this study, the relevant architectural parameter is the level-two cache. We simulate an inclusive 1MB 4-way set associative cache with 64-byte lines.

5. ERROR COVERAGE

In this section, we present an evaluation of the soft-error coverage for four error detection mechanisms. We then use a simple DMR reliability model to relate the coverage to a mean time to failure (MTTF).

As discussed in Section 2.3, rollback recovery fails when the error detection latency extends past the checkpoint interval. In this section, we use the dynamic dependency analysis from Section 4.1 to compute error detection latencies for both chip-external and L1 cache interface detection points. Then, we present a model of the MTTF calculated from these detection latencies and, finally, we discuss the tradeoff between checkpoint interval length and error coverage as it applies to the four detection mechanisms.

5.1 Detection Latency

From the output of the dynamic dependency analysis, we are interested in knowing the likelihood that an error has been detected by the end of a checkpoint interval. The dynamic dependency analysis generates a listing of minimum distances to detection for each instruction in the program trace. We construct a histogram according to the instruction's minimum distance to detection. We normalize by the total instruction count to find the probability distribution of detection distances for the overall program. This distribution tells us the probability of instructions having a given distance to detection.

When working with checkpoint intervals, it is more convenient to work with the corresponding cumulative distribution function (CDF). The CDF for a given detection distance gives the probability that a random instruction has been detected *by* a given distance (instead of *at* a fixed distance). We calculate the CDF for each distance by summing the probability distribution from zero to the detection distance.

In Figure 5, we show the detection latency CDFs for both the chip-external and L1 cache interface detection methods. We compute the aggregate detection distances for three classes of applications: SPEC integer, SPEC floating-point, and the commercial workloads. The full state and fingerprinting detection methods are not shown here because they have immediate detection at the end of the checkpoint interval.

In the L1 cache interface CDF curves, the probability of detecting all latent errors in the register file is almost unity, but a small fraction of instructions are left undetected for millions of cycles. Detection latencies longer than the simulation period cannot be measured, so all CDFs reach 1.0 by the end of the period, even though some errors may actually hide within the cache hierarchy for longer periods of time. While most general purpose registers are constantly overwritten, a few values such as return address pointers may be left untouched for long periods of time before accesses that eventually lead to detection.

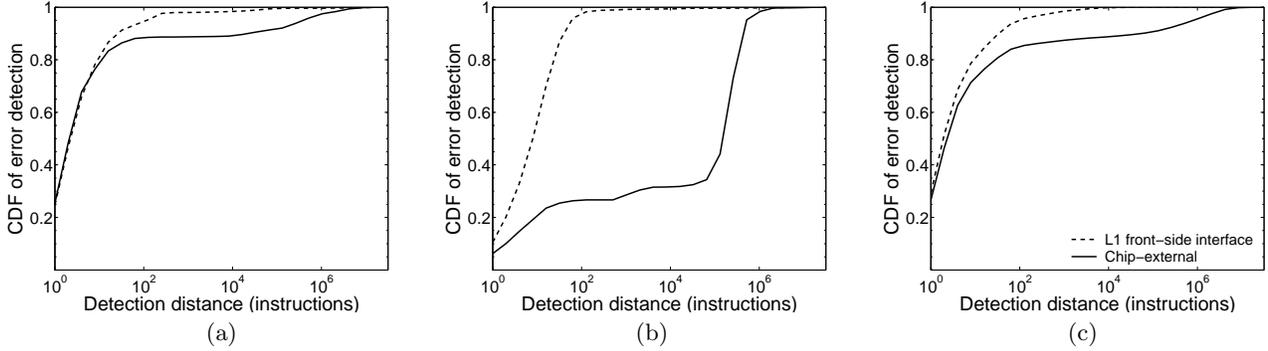For the chip-external detection case, a significant frac-

Figure 5: CDF of error detection computed over the average of (a) SPEC integer, (b) SPEC floating-point, and (c) commercial workloads.
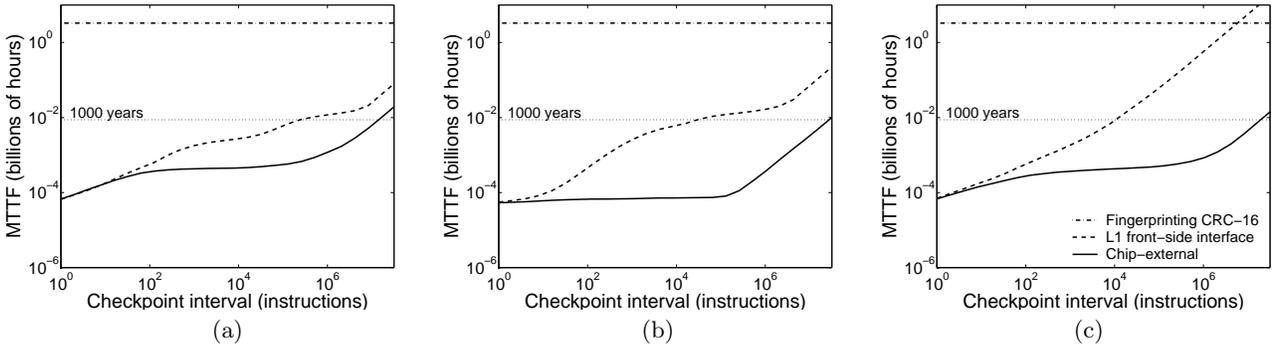

Figure 6: MTTF averaged over (a) SPEC integer, (b) SPEC floating-point, and (c) commercial workloads.

tion of values hide within the cache hierarchy for millions of cycles before being written back to memory. Both SPEC integer benchmarks and the commercial applications show a clear tendency towards keeping some values buffered in the cache extended periods of time. Floating-point benchmarks, however, regularly displace the L2 cache contents due to streaming data access patterns. The cache hierarchy has roughly 131,000 double words and the probability of detecting a fault increases drastically within a few multiples of that many instructions (not every instruction is a load or store).

5.2 Mean Time to Failure Model

Next, we calculate the MTTF of a system where undetected errors may exist at the time a checkpoint is taken. We define a system's error coverage, C, as the probability of detecting all possible errors within a checkpoint interval. For a specific instruction, the CDF gives the probability of detecting an error before the next checkpoint. In our model, each instruction is equally likely to experience an error. Therefore, the coverage is defined as the mean probability that an error could be detected in each instruction before the next checkpoint. For a checkpoint interval of length t, we sum over all i instructions from the checkpoint to obtain:

$$Coverage = \frac{\sum_{i=1}^{t} CDF(i)}{t}.$$

Given the coverage and fault rate, the system MTTF with dual modular redundancy is [31]:

$$MTTF = \frac{1}{2\lambda(1-C)},$$

where λ is the raw transient fault rate and C is the error coverage. The transient fault rate, determined by process and circuit characteristics, is considered constant and independent from this work.

A raw fault rate of 10^4 FIT is a predicted fault rate in logic circuits for high-performance processors early in the next decade [17, 23]. Using the above formula for MTTF, we obtain the failure data shown in Figure 6. For fingerprinting, which has an instantaneous detection latency and detection independent of the checkpoint interval, we define coverage as the probability of detecting an error from Section 3.1, $C = 1 - 2^{-p}$, where we use $p = 16$ for the fingerprint code (i.e., CRC-16).

In all of the benchmarks, we see that chip-external detection requires checkpoint intervals of at least 10 to 25 million instructions to achieve the required reliability. This indicates that in order to achieve acceptable error coverage, checkpoint intervals for chip-external detection must be long: on the order of milliseconds. In Section 7, we will show that the high frequency of I/O operations in OLTP workloads makes this size checkpoint interval infeasible without sophisticated I/O delay mechanisms.

By comparing at the L1 cache interface, the checkpoint interval necessary to achieve an acceptable reliability is reduced to 10 to 100 thousand instructions. This interval is sufficient to support OLTP I/O requirements; however, we

Table 2: Bandwidth requirements for the passive detection mechanisms, in bytes/instruction.

	SPEC Int	SPEC FP	Commercial
Chip-external	0.0038	0.0456	0.1210
L1 cache interface	6.13	5.39	5.92

show in Section 6 that inordinate bandwidth is required to support detection across chips.

In our model, full state comparison covers all errors, leading to an infinite MTTF. Finally, fingerprinting with a 16-bit CRC, whose coverage is independent of the checkpoint interval, has an MTTF that is superior to the chip-external and L1 cache interface detection methods. The checkpoint interval can be reduced to support a high I/O rate, while imposing a minimal bandwidth requirement.

6. COMPARISON BANDWIDTH

In this section, we evaluate error detection mechanisms based on their requirements for state comparison bandwidth between mirrored processors.

Chip-external detection is used in existing systems by placing the mirrored processors in close proximity and comparing data values as they exit the chips [15, 21]. This approach to detection requires no extra pin bandwidth over that already required to run applications, since it passively monitors traffic going to memory or the rest of the system. In Table 2, we report the average bandwidth generated by the three application classes (calculated as the sum of address and data traffic required to complete the memory requests). The passive nature of this approach makes it an attractive option for detecting errors in redundant processors across physical chips. However, the error coverage with this technique only makes it viable at large checkpoint intervals.

Alternatively, error detection can be implemented at the L1 cache interface. As with chip-external detection, this method involves passive comparison of addresses and data, but now the comparison includes all loads and stores. As expected, the average comparison bandwidth reported in Table 2 is orders of magnitude higher than with chip-external. The bandwidth reported here is well above the external pin bandwidth sustainable by a modern processor; therefore, this technique is only applicable when the redundant cores are located on the same die

Full-state comparison at checkpoint creation compares the data changed since the last checkpoint and is therefore dependent on the checkpoint interval. We show the average comparison bandwidth required for a range of checkpoint intervals in Figure 7(a). Notice that the bandwidth for this form of comparison is greater than or comparable to the chip-external bandwidth already demanded by the applications. The average bandwidth requirement decreases as the checkpoint interval increases. Intuitively, this trend is expected because programs have spatial locality within a limited working set. As the checkpoint interval grows, the working set size generally grows at a slower rate. Because of the bandwidth overhead, full-state comparison is only viable for very large checkpoint intervals.

Another view of the full-state comparison is to consider the amount of data that must be transfered in each checkpoint interval. We show this graphically in Figure 7(b). If

this bandwidth cannot be sustained during the checkpoint interval, execution must stall until the comparison has completed. The following equation estimates the checkpoint bandwidth for a checkpoint interval, given the checkpoint size, the estimated instructions per cycle, and processor clock frequency:

$$Bandwidth = \frac{Checkpoint\ Size \cdot IPC \cdot Frequency}{Checkpoint\ Interval}.$$

The bandwidth required for fingerprinting is the size of the fingerprint (two bytes) over the checkpoint interval. We also plot the fingerprinting bandwidth requirement as a function of the checkpoint in Figure 7(a). For checkpoint intervals larger than 1000 instructions, the bandwidth required for fingerprinting is at least an order of magnitude less than the chip-external comparison bandwidth.

Finally, as a reference point on a current state-of-the-art system with a 2.8GHz Pentium 4 Xeon, the theoretical bandwidth from the 533MHz bus is 4.2GB/s. A memory streaming test intended to maximize utilized external bus bandwidth achieves a maximum throughput of 2.5GB/s. Of this usable bandwidth, we measure a TPC-C like database on the system to generate a load of 980MB/s on the bus, which corresponds to the chip-external bandwidth. Assuming the system executes one instruction per cycle and the checkpoint interval is 32K instructions, the estimated bandwidth required for full-state detection is 440MB/s—50% of the existing bandwidth from the TPC-C like database application. In comparison, fingerprinting requires 64KB/s under the same assumptions.

7. I/O PERFORMANCE

The preceding results for error coverage and bandwidth requirements would suggest that, without fingerprinting, a larger checkpoint interval is necessary. In I/O-intensive workloads, however, performance suffers with the conventional approach of logging input and delaying output to avoid rollback across the I/O [6, 29].

Sophisticated software solutions can be developed, which delay writes to disk or a network interface and log results of reads from external devices. If enough concurrency is present in the application, and buffering space is not a concern, a software approach may be possible. However, operating systems and commercial applications are too complex to support major changes to the I/O subsystem.

A hardware solution to avoid rollback across I/O would create a checkpoint immediately following every read or write at the device level (uncached loads and stores). Using our full-system TPC-C like simulation (see Section 4.3), we collect traces of reads and writes to the SCSI controller during 100 billion instructions (50,000 DB transactions). Figure 8 shows the cumulative interarrival distribution of device reads and writes.

We observe a clustering of accesses in several places. There is a fine-grained clustering corresponding to the reads and writes required to initiate every disk access (up to 5000 instructions). At 50,000 instructions, we observe a clustering due to the time between physical I/Os sent to the SCSI controller. This point is corroborated with Amdahl's I/O rule of thumb [11] and Gray's updated version [9]: for random I/O access, approximately 50,000 instructions separate physical I/O operations.
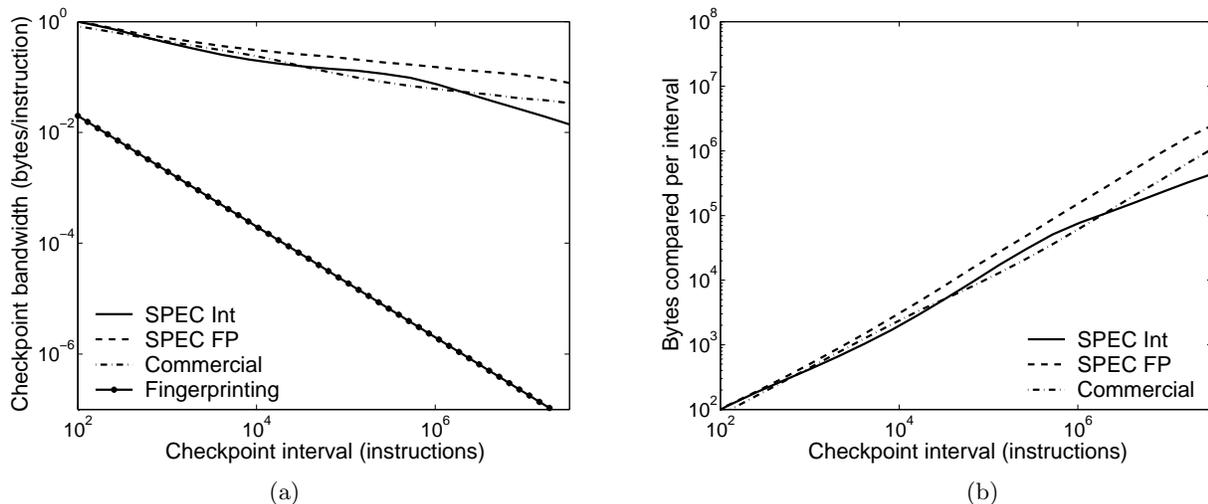
(a)   (b)

Figure 7: In (a) we show the bandwidth requirements for full-state error detection over a range of checkpoint intervals, and (b) bytes compared per checkpoint interval.
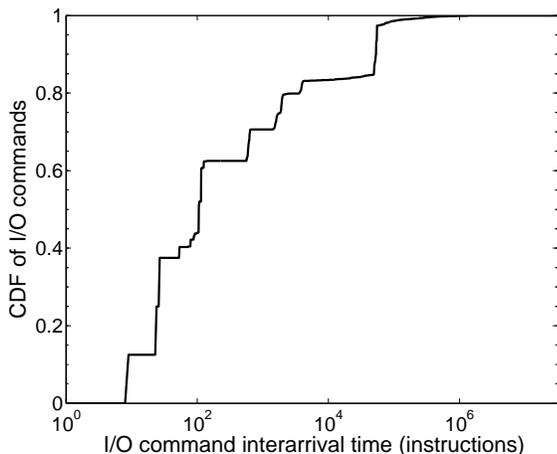

Figure 8: CDF of interarrival of SCSI controller accesses running a TPC-C like workload.

Based on these measurements, we conclude that checkpoints must be created at least as frequently as physical I/O operations (50,000 instructions). Fine-grained checkpoints require a high-coverage, low-bandwidth detection method to guarantee fault-free operation with checkpoint intervals of thousands of cycles. Fingerprinting is precisely this mechanism which when combined with low-cost checkpoint creation, can be used to construct rollback-recovery systems with excellent performance in I/O-intensive workloads.

8. RELATED WORK

Fault Detection. Sogomonyan, et al. [26] propose a mechanism similar in spirit to fingerprinting. Their technique relies on a modified flip-flop design that permits a scan chain to operate without interfering with normal computation. A sequence of single-bit outputs of the scan chain constitute a signature of the internal state. The signature can be

compared to detect differences in the operation of two mirrored systems on a chip (SoC) designs. This proposal differs from the fingerprinting approach in that all flip-flops on the chip must be changed to scannable Multi-Mode Elements (MMEs), and the state is continuously monitored through a 1-bit signature stream. Unlike fingerprinting, the signature stream cannot be tied to a specific instruction; rather, errors may take some time to propagate out the scan chain. This error detection latency may lead to low error coverage and unacceptable poor system MTTF. Additionally, fingerprinting consumes less bandwidth by sending a single 16-bit word for error detection rather than a continuous bit stream.

Other proposed techniques instrument binaries with self-checking mechanisms or use compiler-inserted instructions to check for errors in control flow [34]. Unlike fingerprinting, these techniques cannot detect a large class of errors in dataflow and require difficult analysis to determine error coverage.

Existing commercial fault-tolerant systems use replication in conjunction with lockstepped execution. Two identical copies of a program run on the redundant hardware. The IBM G5 uses replicated, lockstepped pipelines in a single core [25]. The Tandem Himalaya [15] and Stratus [21] use replicated, lockstepped processors and compare execution using the chip-external detection mechanism.

There are several proposals for simultaneous multithreaded processors (SMT) and chip multiprocessors (CMP) with redundant execution on the same die. The DIVA architecture [2] uses a simple in-order checker to detect soft and permanent errors in a closely-coupled out-of-order core using full state comparison. Rotenberg proposed using two staggered, redundant threads in an SMT processor [20] to detect soft errors with full state comparison at commit. Vijaykumar, et al. suggested full-state comparison for detection and recovery in SMT [32] and CMP [8] architectures. Alternatively, Reinhardt and Mukherjee proposed the SRT processor [19] and later the CMP-based CRT processor [16], which compares only stores across threads (equivalent to the L1 cache interface detection mechanism).

Architectural Checkpointing. A prerequisite of all backward error recovery schemes is the checkpoint mechanism. Microarchitectural techniques work for short checkpoint intervals (thousands of instructions). The Checkpoint Processing and Recovery proposal [1] scales the out-of-order execution window by building a large, hierarchical store buffer and aggressively reclaiming physical registers. The SC++lite [7] mechanism speculatively allows values into the processor's local memory hierarchy, while maintaining a history of the previous values.

On a larger scale, global checkpoints can take a consistent checkpoint of the architectural state across a multiprocessor, but at intervals of hundreds of thousands to millions of instructions. SafetyNet [27] provides a way to build a global checkpoint in a multiprocessor system. Processor caches are augmented with checkpoint buffers that contain old cache line values on the first write to a cache block in each interval. ReVive [18] takes global checkpoints in main memory by flushing all caches and enforcing a copy-on-write policy for changed cache lines after the checkpoint time.

Fault Analysis. Our technique for measuring error detection latency differs substantially from the traditional fault-injection approach, which inserts single errors at the gate, register, or pin level and observes the corresponding detection latency [10, 13, 33]. The injection approach suffers from a number of limitations. Modern microprocessors are sufficiently complex that fault injection cannot cover a reasonable subset of potential transient faults in an acceptable time. Other fault injection techniques include bombarding an actual chip in heavy ion testing [36].

Austin [3] used dynamic dependency analysis very similar to our dependency analysis to analyze the dataflow parallelism in SPEC benchmarks. The goal of this work was to find available parallelism in real programs using register renaming. Austin did not incorporate cache dependencies and there is no sense of detection or appearance of values at the chip pins.

9. CONCLUSIONS

Increasing soft-error rates in microprocessor logic will reach unacceptable levels in the near future. We identify three metrics for evaluating DMR error detection mechanisms: error coverage, comparison bandwidth, and I/O performance. No existing error detection mechanism satisfies all three metrics. We propose *fingerprinting*, a hash of updates to architectural state, as an approach that offers excellent error coverage, low bandwidth requirements, and inexpensive on-demand comparison. We evaluate existing mechanisms for error detection in DMR systems, and quantify the benefits fingerprinting provides.

10. ACKNOWLEDGEMENTS

11. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec 2003.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, Nov. 1999.

[3] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[4] D. Bossen, J. Tendler, and K. Reick. Power4 system design for high reliability. In *Hot Chips - 13*, August 2001.

[5] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[6] E. N. Elnozahy, D. B. Johnson, and Y. M. 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.

[7] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the Tenth International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[8] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30h Annual International Symposium on Computer Architecture*, June 2003.

[9] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proceedings of the IEEE International Conference on Data Engineering*, Feb 2000.

[10] M. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: a framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Annual Workshop on Compilers for Parallel Processing*, Aug 1993.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[12] 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.

[13] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the valiadation of system dependability properties. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, 1992.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[15] D. McEvoy. The architecture of tandem's nonstop system. In *ACM/CSC-ER*, 1981.

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[17] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec 2003.

[18] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, June 2002.

[19] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 2000.

[20] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, June 1999.

[21] L. Sherman. Stratus continuous processing technology – the smarter approach to uptime. Technical report, Stratus Technologies, 2003.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[23] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *International Conference on Depdendable Systems and Networks*, June 2002.

[24] D. P. Sieworek and R. S. S. (Eds.). *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[25] T. J. Slegal and et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12 – 23, March - April 1999.

[26] E. Sogomonyan, A. Morosov, M. Gossel, A. Singh, and J. Rzeha. Early error detection in systems-on-chip for fault-tolerance and at-speed debugging. In *Proceedings of the 19th VLSI Test Symposium*, May 2001.

[27] 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 *Proceedings of the 29th Annual International Symposium on Computer Architecture*, June 2002.

[28] Standard Performance Evaluation Corporation. SPECweb99 benchmark. http://www.specbench.org/osg/web99/.

[29] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[30] The Transaction Processing Performance Council. TPC Benchmark C: Standard specification. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, Dec 2003.

[31] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, 2nd edition, 2001.

[32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[33] N. Wang and S. Patel. Modeling the effect of transient errors on high performance microprocessors. In *Center for Circuits, Systems, and Software (C2S2), 2nd Annual Review*, March 2003.

[34] K. Wilken and J. P. Shen. Continuous signature monitoring: Low-cost concurrent dectection of processor control errors. *IEEE Transactions on Computer-Aided Design*, 9(6):629–641, June 1990.

[35] J. K. Wolf, A. M. Michelson, and A. H. Levesque. On the probability of undetected error for linear block codes. *IEEE Transactions on Communications*, 30(2), Feb 1982.

[36] J. F. Zeigler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross. Accelerated testing for cosmic soft-error rate. *IBM Journal of Research and Development*, 40(1), 1996.