The Granularity of Soft-Error Containment in Shared Memory Multiprocessors

Brian T. Gold, Jared C. Smolens, Babak Falsafi, and James C. Hoe

Abstract—**Concerns over rising soft-error rates in processor logic has led to numerous proposals for error tolerance mechanisms. In this paper, we examine the role of soft-error containment in a shared memory multiprocessor. We study a range of design alternatives based on how far outside the processor core errors are allowed to propagate. We discuss tradeoffs in recovery complexity and error-free performance that arise from the choice of containment granularity.**

I. Introduction

Increasing levels of integration, particularly in the era of billion-transistor chip multiprocessors (CMPs), have led to concern over rising soft-error rates in otherwise-unprotected processor logic [4,5,20]. While parity and error-correcting codes (ECC) can detect and correct many soft errors in caches, memory, and storage, the complex layout and timing-critical nature of the processor pipeline leaves it vulnerable to soft errors [11].

Sparked by the concern over rising soft-error rates, recent work has proposed numerous soft-error tolerance mechanisms, which cover a range of implementation and error tolerance issues. Often, comparing different proposals is a difficult task because the system goals vary widely across designs (e.g., high availability vs. minimizing silent data corruption).

One approach to studying the range of existing mechanisms is to classify them under a common fault model and general system architecture. In this paper, we study soft errors originating in the processor core, but make no assumptions as to their eventual effect on applications or the system (i.e., we assume silent data corruption can result). We frame our analysis in the context of a shared memory multiprocessor that uses checkpoint and rollback as a recovery mechanism.

We propose a taxonomy for reliable shared memory multiprocessors based on how far outside the processor core the design permits soft errors to propagate. We study a spectrum of containment granularities and draw the following conclusions:

- **Core containment.** Confining errors to the processor core permits the use of unmodified memory systems (caches and main memory). However, in-core detection and recovery requires extensive changes to the complex and timing-sensitive pipeline.
- **Cache containment.** Allowing errors that originate in the core to propagate into local, private caches simplifies recovery and eliminates large-scale changes to the pipeline. However, frequent checkpoints and changes to the cache-coherence protocol incur performance overheads and complexity.
- **Memory containment.** Permitting errors to propagate into main memory further reduces core and coherence protocol changes. However, checkpoints must be coordinated across the system, which interferes with conventional I/O device interaction.

Paper Outline. This paper is structured as follows. In Section II, we describe necessary background for our analysis. In Section III, we discuss three granularities of soft-error containment and their tradeoffs. Section IV concludes the paper.

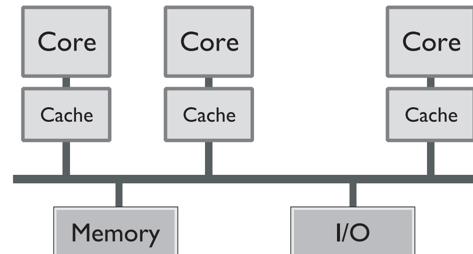

Fig. 1. The generic shared memory multiprocessor assumed in this paper. Each processor core has private caches, which are backed by an interconnect that logically connects devices, main memory, and peripherals.

II. Background

The majority of multiprocessor systems in use today are based on the shared-memory programming model, whether in the form of chip multiprocessors (CMPs), symmetric multiprocessors (SMPs), or distributed shared-memory multiprocessors (DSMs). Figure 1 depicts a generic shared-memory multiprocessor that consists of several processor cores and caches, a main memory (DRAM) subsystem, and I/O devices. Although specific designs differ in the way components connect (e.g., shared caches in a CMP or a point-to-point interconnect in a DSM), the basic connection of processors to a logically-shared memory and I/O devices is common to all designs.

In this paper, we focus on detecting and recovering from soft errors within the processor core. Our fault model assumes that errors originating in components outside the processor core in Figure 1 are protected by existing techniques. We do not restrict errors to those detectable or correctable by existing hardware or software checks. Rather, we assume errors in execution could result in silent data corruption.

The authors are with the Computer Architecture Lab at Carnegie Mellon University, Pittsburgh, PA. http://www.ece.cmu.edu/~truss; babak@cmu.edu.

This work was funded in part by NSF awards ACI-0325802 and CCF-0347560, Intel Corp., the Center for Circuit and System Solutions (C2S2), the Carnegie Mellon CyLab, and fellowships from the Department of Defense and the Alfred P. Sloan Foundation.

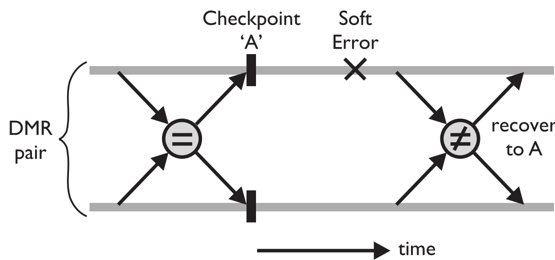

Fig. 2. Error detection and checkpointing in a DMR processor pair. After the pair successfully compares previous execution, a new checkpoint is created as the point of recovery, should a soft error occur later.

A. Backwards Error Recovery

In this paper, we evaluate systems with backwards error recovery (BER) mechanisms that create checkpoints of correct system state and roll back execution to the checkpoint when an error is detected. Our BER model assumes that only one checkpoint is kept at a time. When recovering, the system is restored to the previous checkpoint, and when a new checkpoint is created, it immediately replaces the previous one. Thus, it is critical that checkpoints be free of errors; otherwise, the system cannot recover to an error-free state.

B. Flexible Error Detection

Fingerprinting, a recent proposal for error detection [22], uses a compressed signature of execution results to compare dual-modular redundant (DMR) processor pairs. In a basic implementation, retired instruction results are added to a continuously-updated hash value that, when compared across DMR processors, provides sufficient certainty that execution is error-free.

One benefit of this approach is the flexibility given to *when* error detection occurs. Although the fingerprint mechanism captures execution state when instructions retire, there is no explicit requirement as to when the signatures are compared across DMR pairs.

Figure 2 illustrates the relationship between checkpointing and the fingerprint-based error detection assumed in this paper. The two processors in a DMR pair generate and compare fingerprints that summarize the execution state. If the fingerprints match, the recovery point is advanced by creating a new checkpoint. Conversely, if the fingerprints differ, recovery initiates using the previous checkpoint.

Because errors must be detected before advancing the recovery point, the creation of a new checkpoint always follows a fingerprint comparison. Coupling the checkpointing mechanism with fingerprint comparison enables us to examine tradeoffs in complexity and performance as a function of how far we permit errors to propagate outside the processor.

C. Soft-Error Containment

The idea of containing faults originated in large-scale distributed systems [13], where designs isolate the effects of a fault to a specific boundary. In the distributed-systems context, fault containment permits a portion of the system to remain operational after a fault occurs.

In a shared-memory multiprocessor, fault containment is particularly critical to system reliability because a fault in one processor can lead to the entire system failing [26]. Traditionally, containment has been used to minimize the impact of a permanent failure (e.g., the loss of a processor or memory component). Recent work has focused on minimizing the impact of such failures by providing operating system 'containers' which define the processor and memory resources available to a given process or virtualized operating system image [14]. This technique is complementary to the mechanisms discussed in this paper, as it targets the impact of hard errors on the system.

In the context of soft errors, several tradeoffs depend on the granularity of error containment, such as the portion of the system state that must be rolled back and the overhead associated with error-free checkpoint creation. The error detection and recovery scheme illustrated in Figure 2 provides a definition for the error containment boundary at the point of fingerprint comparison. Before committing side effects outside the containment boundary, the system initiates fingerprint comparison and checkpoint creation. By coupling specific events, such as device accesses, coherence activity, stores to memory, etc., to checkpoint creation and error detection, the system has a well-defined containment boundary.

III. Containment Granularities

We examine a range of containment boundaries, illustrated in Figure 3 using a dashed border. In each case, we discuss the necessary checkpoint state, when error detection and checkpoint creation are required, and the impact on recovery complexity and error-free performance.

A. Processor Core

We first consider a fine-grain boundary that prevents errors from leaving the processor core itself: *core containment*. At this granularity, no error may be committed to architectural state outside the core—that is, stores to cacheable memory and uncached accesses to devices cannot exit the core without being checked.

Some designs go further with these checking requirements, ensuring that each instruction is checked before retirement to the architectural register file. A widely-recognized example of this design point is the IBM z-series processor [21], which uses lockstepped, redundant pipelines to ensure no soft error propagates to any architectural state. The z-series processor is a custom design—the redundant pipelines are carefully laid out to provide low-overhead checking of results. If an error is detected, the processor raises an exception that forces the offending instruction to be re-executed.

Recent work has focused on reducing the overhead of completely replicating the pipeline as in [21]. Instead, a number of designs propose microarchitecture-based checkers, which execute a redundant copy of an instruction later in time—enabling the second execution to leverage much of the work of the first, speculative execution. Some designs (e.g., [1]) use separate, simplified checker hardware that compare only the final, in-order instruction stream. Other proposals (e.g., [16,17,23]) integrate checking within a speculative, out-of-order core and leverage otherwise-unused resources to execute the redundant

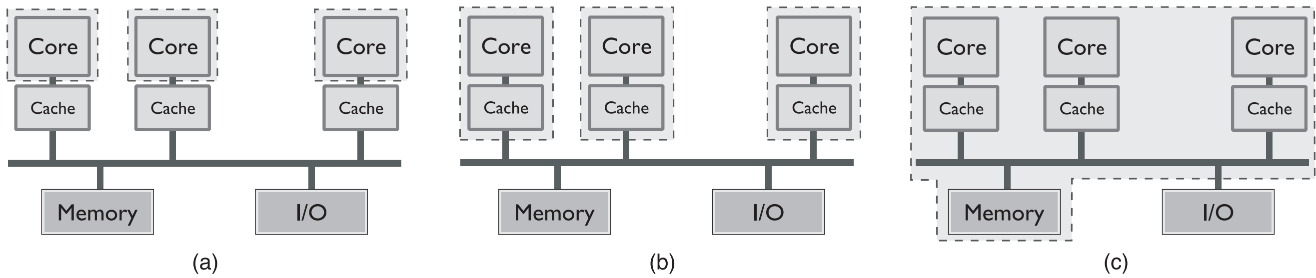

Fig. 3. Soft-error containment granularities studied: (a) processor core, (b) cache, and (c) memory. In each case, the shaded region represents the components in which errors are permitted to propagate. Allowing errors to propagate further away from the core increases the state kept in each checkpoint, but also reduces the changes necessary in the complex and timing-sensitive processor core.

copy. As in the z-series processor, recovery is initiated by raising an exception on any instruction whose redundant executions fail to match, thereby preventing errors from entering the architectural register file or beyond.

We also include proposals for redundant multithreading as providing core containment, whether within one core [18,19,27] or across a pair of cores [8,10,25]. In these designs, one thread of execution runs ahead of the second, redundant copy, which typically requires fewer pipeline resources for its execution—memory values and branch outcomes are forwarded from leading thread to trailing thread. Detection can be enforced before all register updates [8,10,27] or before committing side effects outside the core [18]. As proposed in [8,27], rollback recovery can utilize the trailing thread state as the checkpoint after detecting an error.

Containment within the core will observe more errors than at coarser levels, where errors may be masked by later computation. However, with hardware support for recovery within the pipeline, more frequent recovery will have little or no performance overhead.

In all variations of the core-containment granularity, the key drawback to these designs is the magnitude of changes required to the complex and timing-critical pipeline. However, confining errors to such a small boundary avoids system-wide recovery and changes to the memory-system design.

B. Cache

The next granularity of error containment surrounds the core and private caches, which we refer to as *cache containment*. Here, erroneous results are allowed to propagate into the local caches, but are stopped from being transferred to main memory or other logical processors.

Unlike in core containment, where individual instructions define when error detection and checkpoint creation occur, cache containment is defined at the level of the cache coherence protocol. Writebacks to main memory and the sharing of modified data among processors induce detection and checkpoint creation [7], because these transactions can expose errors outside of the containment boundary.

A downside to the cache containment granularity is complexity stemming from the asynchronous nature of cache coherence protocols. In general, the processor does not know when a coherence request is about to arrive, or even when a writeback will be necessary. By involving the coherence protocol, this approach requires changes to the cache coherence controller—a complex part of a multiprocessor system that is already difficult to design correctly.

Several industry examples of cache containment exist, although none provide hardware-only rollback recovery. Traditionally, HP (formerly Tandem) NonStop servers paired commodity processors in lockstep, checking the pair's results that appear on the front-side bus [2]. Other, similar examples of cache containment include SMP servers from Stratus, Marathon, and Sun (ftSPARC). All of these examples require software checkpointing for rollback recovery—the hardware provides fail-stop processing and passes the failure to software.

Detection outside the pipeline avoids complex modifications to the processor core; however, when hardware rollback recovery is desired, a checkpoint of architectural registers is necessary. The frequency of coherence traffic enables on-chip checkpoints with little storage overhead, but requires checkpointing of cache state. Fortunately, both register and cache checkpointing has been widely studied in the context of speculative microarchitectures [9,28].

TRUSS [7] is our recent proposal for a reliable shared-memory multiprocessor. TRUSS utilizes cache containment to provide software-transparent support for both error detection and recovery in the processor. In the TRUSS architecture, processors create checkpoints on every cache coherence event, forcing error detection before committing side-effects to shared machine state.

Forcing error detection and checkpoint creation at the granularity of coherence protocol transactions avoids the problem of coordinating checkpoints across the system. Each logical processor creates a local checkpoint, with the next checkpoint occurring at or before the next coherence message, which removes any form of communication among logical processors between checkpoints. Thus, cache containment avoids the problem of coordinating recovery or cascading rollbacks associated with conventional, uncoordinated checkpointing schemes [6,15].

C. Memory

Where creating frequent checkpoints or changing the cache controller is too costly, the *memory containment* granularity offers an alternative. Here, all logical processors and memory are grouped together in the containment boundary, so that only operations that cause device accesses (disk, network, etc.) require detection and recovery.

The memory-containment granularity is used in HP's NonStop Advanced Architecture (NSAA) [3]. Unlike previous NonStop designs, NSAA permits processors to write into a private region of main memory and compares results before performing peripheral device accesses. Although not strictly a shared-memory multiprocessor—NonStop systems use message passing for inter-processor communication—the NSAA approach still fits within the definition of memory containment.

In the context of a shared-memory multiprocessor, checkpoints at the memory-containment granularity consist of both architectural register state (as in the previous cases) and memory state. A common approach to creating a memory checkpoint is to log old values when new, speculative values are written [15,24].

The primary challenge at the memory-containment granularity is that of checkpoint coordination. Without coordinating checkpoint creation, the system can become incoherent upon recovery if a processor's modified data is not kept synchronized with the state of the memory checkpoint. Therefore, most solutions at this granularity target a global system checkpoint [15,24].

Unlike checkpoint creation in the cache-containment granularity, the system cannot support arbitrarily fine-grained intervals between checkpoints. Coordinating the logical processors and memory to create a global checkpoint necessarily implies time-intensive operations such as the flushing of caches or waiting for coherence activity to complete.

With a limit on how frequently checkpoints can be created, there is a potential problem for device accesses, which must wait until the next interval to be sent outside the containment boundary. One solution [12] is to modify the operating system to provide a non-blocking pseudo-device driver (PDD) that permits applications to continue operation while the request is buffered in memory. Once the next checkpoint interval arrives and error detection is successful, the PDD releases the device access outside the containment boundary.

A limitation to the approach in [12] is the reliance on idempotent input and output operations, which can be reissued manually or automatically by a higher-level protocol (e.g., TCP). Although the classes of protocols and devices suitable for such a system are widely applicable, not all protocols work with this approach (e.g., UDP).

IV. Conclusions

Within the context of a shared memory multiprocessor, we examined the impact of soft-error containment granularity on recovery complexity and error-free performance. We studied a range of design alternatives based on how far errors are allowed to propagate outside the processor core.

References

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, November 1999.

[2] W. Bartlett and B. Ball. Tandem's approach to fault tolerance. *Tandem Systems Rev.*, 8:84–95, February 1988.

[3] D. Bernick et al. Nonstop advanced architecture. In *Proc. Intl. Conf. on Dependable Systems and Networks*, 2005, 12–21.

[4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–17, November-December 2005.

[5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[6] E. N. Elnozahy et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[7] B. T. Gold et al. TRUSS: a reliable, scalable server architecture. *IEEE Micro*, Nov-Dec 2005.

[8] M. Gomaa et al. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[9] J. F. Martinez et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. of 35th IEEE/ACM Intl. Symp. on Microarch. (MICRO 35)*, Nov 2002, 3–14.

[10] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002, 99–110.

[11] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. of 36th IEEE/ACM Intl. Symp. on Microarch. (MICRO 36)*, Dec 2003.

[12] J. Nakano et al. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *HPCA*, 2006.

[13] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Micro*, 23(7):19–25, 1990.

[14] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proc. 18th Large Installation Systems Administration Conf. (USENIX LISA)*, 2004.

[15] M. Prvulovic et al. ReVive: cost-effective architectural support for rollback recovery in shared memory multiprocessors. In *Proc. of 29th Intl. Symp. on Comp. Arch. (ISCA-29)*, June 2002.

[16] M. K. Qureshi et al. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *Proc. of 32nd Intl. Symp. on Comp. Arch. (ISCA-32)*, June 2005.

[17] J. Ray et al. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[18] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

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

[20] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002, 389–398.

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

[22] J. C. Smolens et al. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proc. of Eleventh Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS XI)*, Boston, Massachusetts, Oct. 2004. 224–234.

[23] J. C. Smolens et al. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. of 37th IEEE/ACM Intl. Symp. on Microarch. (MICRO 37)*, December 2004.

[24] D. J. Sorin et al. SafetyNet: improving the availability of shareed memory multiprocessors with global checkpoint/recovery. In *Proc. of 29th Intl. Symp. on Comp. Arch. (ISCA-29)*, June 2002.

[25] K. Sundaramoorthy et al. Slipstream processors: Improving both performance and fault tolerance. In *ASPLOS*, October 2000.

[26] D. Teodosiu et al. Hardware fault containment in scalable shared-memory multiprocessors. In *Proc. of 24th Intl. Symp. on Comp. Arch. (ISCA-24)*, June 1997.

[27] T. N. Vijaykumar et al. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[28] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996