Appears in the 10th Workshop on Silicon Errors in Logic - System Effects (SELSE), April 2014

Experiences in Developing and Evaluating a Low-Cost Soft-Error-Tolerant Multicore Processor

John S. Ingalls, Adam N. Jacobvitz, Patrick J. Eibl, Michael R. Ansel, and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University, Durham, NC
{john.ingalls, eibl, michael.ansel}@alumni.duke.edu, adam.jacobvitz@duke.edu, sorin@ee.duke.edu

Abstract—The purpose of this work is to experimentally demonstrate that a synthesis and implementation of existing ideas can achieve the goal of a low-cost, soft-error-tolerant multicore processor. We show that a multicore processor can be designed that tolerates the vast majority of soft errors, with area, power, and performance costs that are within 20% of a baseline processor without fault tolerance.

Keywords—error detection, error correction, multicore processor, computer architecture, hardware prototype

I. INTRODUCTION

To tolerate soft errors—which requires both detecting the errors and recovering from their effects—industry and academia have developed a wide variety of approaches. Holistic approaches like N-modulo redundancy (NMR) are too expensive, in terms of power and area, for all but the most critical applications. All other schemes are either incomplete (e.g., redundant multithreading [1][2] protects the cores but not the memory hierarchy), unimplemented in hardware (e.g., simulated in SimpleScalar [3] or Simics [4] which may affect results and conclusions [5]), or have other significant limitations (e.g., SWAT [6] detects errors but with no bound on the detection latency). Moreover, multiple schemes can impose configuration constraints on each other and thus should be designed and evaluated in concert.

The goal of this work is to demonstrate that a synthesis and implementation of existing ideas can achieve the goal of a low-cost, soft-error-tolerant multicore processor. We start with a baseline multicore processor (Section II) implemented at the RTL level in synthesizable Verilog. We implement previously developed mechanisms for error detection (Section III) and error recovery (Section IV), including those designs that had not been built in hardware before. We then discuss how we integrate the error tolerance mechanisms and what new challenges arise during integration (Section V). We experimentally evaluate the complete multicore processor, using a benchmark developed for this purpose (Section VI), in terms of its ability to tolerate soft errors (Section VII) and its area, power, and performance costs (Section VIII). We show that a multicore processor can be designed that tolerates the vast majority of injected soft errors (only 5.3% of *unmasked* injected errors cause silent failures), with a 16% area cost, 18% power cost, and 20% performance cost. We discuss the applicability of our results to other multicore system models (Section IX), and we discuss related work (Section X).

II. BASELINE MULTICORE SYSTEM MODEL

Our baseline multicore processor consists of multiple OpenRISC 1200 (OR1200) cores [7]. Each OR1200 core is a simple, in-order, scalar pipeline that has its own instruction cache and data cache. The cores do not have floating point units. The cores are representative of the low-power cores found in embedded applications and throughput-oriented multicore chips (e.g., Niagara [8][9]).

The cores communicate across a shared bus, using a MOSI snooping cache coherence protocol that we designed (because the OR1200 does not natively support cacheable hardware coherent shared memory). The shared bus also connects to main memory, and the memory controller implements the logic required to participate in the coherence protocol. The memory infers its own coherence state by maintaining duplicate copies of the coherence states of the data caches. The system supports sequential consistency [10]. We illustrate the system in Figure 1, and we provide the full specifications in Table 1.

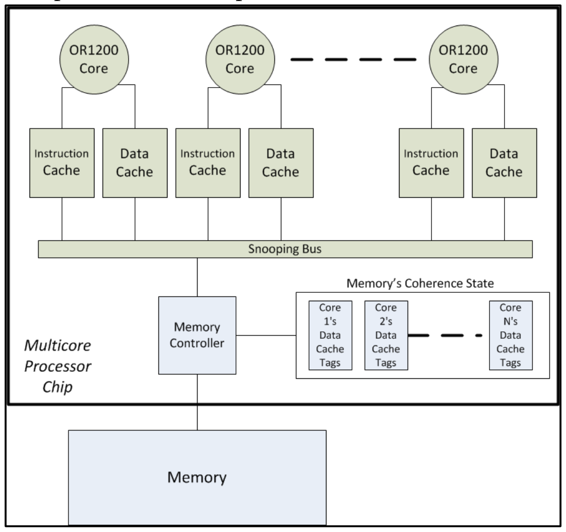

Figure 1. Baseline System Model

III. ERROR DETECTION

We detect errors using a composition of two previously developed error detection schemes. For the cores (including the instruction caches), we use an implementation of the Argus dynamic verification scheme [11]. For the memory system—including data caches, coherence controllers, and the interconnection network—we use dynamic verification of cache coherence (DVCC) [12].

Table 1. System Specifications

System Parameter	Value
Processor cores	3 in-order, single-issue, 4-stage OpenRISC 1200 cores
Instruction cache	4KB, direct-mapped, 16B lines
Data cache	8KB, direct-mapped, 16B lines
Coherence protocol	MOSI snooping
Snooping bus	1 coherence transaction per cycle
Data network	Fully connected, each link can transmit 1 word every 2 cycles
Core Error Detection (Argus)	
Signature lengths	All signatures are 5-bit
Residue checkers	31-bit modulus
Memory System Error Detection (TCSC)	
Sig_{req}	7-bits each (x4 Sig_{req} registers per cache and memory)
Sig_{resp}	5-bits each (x4 Sig_{resp} registers per cache and memory)
Checkpointing Error Recovery	
Data cache Address/State log	32 entries, 16 bits/entry
Data cache Data log	64 entries, 32 bits/entry
Memory Address/Data log	32 entries, 143 bits/entry

A. *Core Error Detection*

Our Argus implementation consists of three invariant checkers: control flow [13][14][15], dataflow [16][17], and computation [18]. Our Argus implementation uses signatures (lossy checksums) to detect errors in control flow and dataflow. It uses residue codes (modulo arithmetic) to detect errors in functional units. Prior work showed that hardware implementations of Argus can detect the vast majority of injected errors while incurring relatively low performance and area costs [19][11]. Argus's power consumption is quantitatively evaluated for the first time in this paper.

B. *Memory System Error Detection*

Prior work called Token Coherence Signature Checking (TCSC) showed how to dynamically verify coherence in a system that supports sequential consistency [12]. First, each core *locally* checks that each of its loads and stores is performed to a block for which the core has appropriate coherence permissions, using coherence state tokens kept with each cache block [20]. Second, TCSC *globally* dynamically verifies that every increase in permission at a cache or memory is offset by a corresponding decrease in permission at another cache or memory. Each cache and memory keeps a signature of its recent coherence history. For example, if a cache obtains read-only access to block B at time T, it updates its signature based on B and T and the fact that the request obtained read-only permission. The signatures are aggregated periodically and easily checked to determine whether an error has occurred.

Prior work has implemented and evaluated TCSC in a high-level simulator (Simics [4] plus GEMS [21]), but never implemented it in hardware.

TCSC's local and global checkers detect errors in coherence transitions, but they do not detect errors in static state. Thus we add parity bits to data cache blocks and memory blocks.

C. *Watchdog Timers*

Our implementations of Argus and TCSC detect violations of *safety*. That is, they detect if something incorrect occurs. However, they do not detect violations of *liveness*. We added simple watchdog timers that report an error if no instruction commits in a long time.

IV. Error Recovery

Detecting an error is sufficient to avoid silent data corruptions (SDCs) but not sufficient for seamlessly tolerating the error. Ideally, the processor detects an error, recovers from the error by putting the processor back into a pre-error state, and then resumes execution.

A. *Our Implementation*

We implement error recovery with a straightforward, checkpoint/recovery mechanism that is similar to CARER [22] and ReVive [23]. Each processor core's state, which consists mostly of architectural registers—including both general purpose and special purpose registers (e.g., program counter, processor status word, etc.)—is checkpointed to a backup register file. Changes to the memory state are logged rather than checkpointed. If a core performs a store or if a coherence event changes the state of a memory block, then the *previous* value/state of the block is logged. Each data cache and memory has its own log.

We coordinate the taking of checkpoints at caches and memory to create a consistent recovery point to avoid the possibility of cascading rollbacks (the so-called "domino effect") [24]. Checkpoint coordination follows a standard centrally coordinated handshaking protocol that we illustrate in Figure 2. The decision to take a checkpoint is initiated by a cache or memory with a nearly full log.

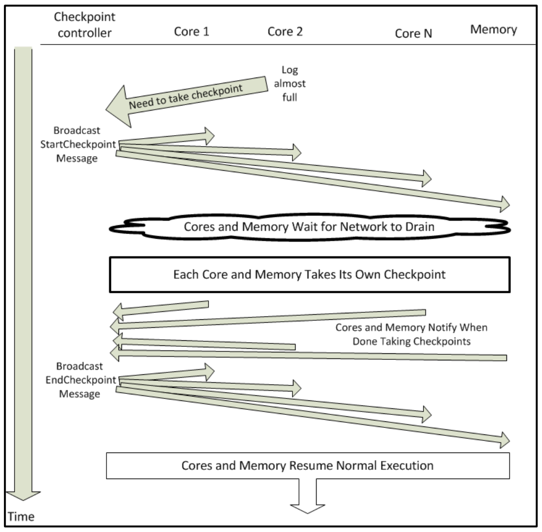
Checkpoint controller
Core 1
Core 2
Core N
Memory
Log almost full
Need to take checkpoint
Broadcast StartCheckpoint Message
Cores and Memory Wait for Network to Drain
Each Core and Memory Takes Its Own Checkpoint
Cores and Memory Notify When Done Taking Checkpoints
Broadcast EndCheckpoint Message
Cores and Memory Resume Normal Execution
Time

Figure 2. Process of Taking a Checkpoint

When a recovery is initiated, the cores revert back to their checkpointed register state, and the caches and memory rewind their logs to recover their prior memory state. When the caches recover their prior state, they also convey the recovery information to the memory controller to recover its coherence state (i.e., its duplicate tags).

Our checkpoint/recovery mechanism handles the output commit problem [24] in the standard way. We buffer potential outputs until the error detection mechanisms have sufficient time to determine whether the data is error-free and can safely leave the sphere of recoverability.

B. *Error Recovery Capability*

Some errors that are detected will be unrecoverable due to limitations of our checkpoint/recovery mechanism. In general, an error that somehow corrupts both checkpoint state and "normal" (non-checkpoint) state may be unrecoverable.

C. *Costs*

The logs are the primary cost of the checkpoint/recovery mechanism, and choosing their size is critical. A (nearly) full log forces a checkpoint to be taken, and thus too-small logs would force too-frequent checkpoints. However, too-large logs would consume more chip area and power. We empirically determined a log size that balances this trade-off.

V. Putting It All Together

At a high level, the design of an error tolerant multicore processor appears to be the composition of several independent mechanisms. However, integrating and optimizing these mechanisms together into a cohesive whole involved a significant amount of effort and understanding the subtle interactions between them, primarily between Argus with checkpoint/recovery and TCSC with checkpoint/recovery. The interactions between Argus and TCSC are minimal. There is also a straightforward interaction between the watchdog timeout and checkpointing: the error detection latency (timeout threshold) must be shorter than the checkpoint interval.

A. *Argus and Checkpoint/Recovery*

Argus's control flow and dataflow checking operate at the basic block granularity. Thus, in a single-core system, it would be simplest to checkpoint Argus only at the ends of basic blocks. However, in a multicore system with coordinated checkpoints, the decision of when to take a checkpoint is not left to the preference of each core. When one cache or memory has a log that is nearly full, it initiates a checkpoint at all cores and memory. Thus we support checkpointing at any time during a basic block, which requires us to also checkpoint the state of the first two core pipeline stages, including their micro-architectural Argus state, due to the OpenRISC ISA's branch delay slots.

Argus's control flow and dataflow checkers detect corrupted instructions and thus obviate the need for TCSC to protect the instruction caches. However, after recovery, the erroneous instruction is still in the cache and will be soon fetched again, leading to an endless cycle of detection/recovery events. To solve this problem, we conservatively invalidate the instruction cache if Argus detects a control flow or dataflow error.

B. *TCSC and Checkpoint/Recovery*

When taking/restoring a checkpoint, we must capture/restore consistent cache coherence and TCSC state.

What to Checkpoint. TCSC state includes the signature registers at the caches and memory, as well as the token state at the caches. Recovering the signature registers involves simply clearing them, thus they do not need to be checkpointed. The token state, however, cannot be "cleared" like the signatures (i.e., we cannot zero tokens for all blocks), because that would be incompatible with the actual recovered coherence state in the caches. Thus, as a core rewinds its cache logs, it also restores the token state for TCSC. Recall from Section IV that the memory recovers its coherence state (i.e., its duplicated cache tags) using the cache logs. We also optimize logging by only logging changes in cache block *ownership*; *shared* blocks are downgraded to invalid as part of a restore. Thus the system is restored to a slightly different but still correct coherence state.

When to Checkpoint. Checkpointing coherence state while coherence messages are in flight is complicated and can lead to cascading rollbacks, because some, but not all, TCSC signatures reflect the effects of in-flight messages. Notably, a cache that sends a message that gives away permissions has updated its signature but the cache receiving that message has not yet updated its signature to reflect receiving permissions. We chose the simplest solution to this problem: the Checkpoint Controller waits to send a TakeCheckpoint message until all messages have drained from the network.

VI. Benchmark

To evaluate error tolerance, performance, and power, we need a software benchmark to run on the hardware. We specially designed a benchmark to exercise as much of the hardware as possible and to have a clearly defined output that can be checked to determine whether execution was correct.

We developed a custom pthreads library for the OR1200, and we used it to create a multithreaded benchmark that is a synthesis of two benchmarks from the ParMiBench suite [25]. Our benchmark computes a series of square roots and then a series of SHA-1 hashes. The benchmark is computationally intensive, exercises all registers and all instructions, and performs enough communication to extensively stress the coherence protocol. Each thread performs approximately 800,000 instructions.

VII. Evaluation of Error Tolerance

To determine how our system tolerates errors, we injected 8,567,253 errors [26][27] into the synthesized Verilog netlist and observed its behavior on an FPGA [28]. In particular, we monitored whether the error was detected and, if so, whether it was recovered from. Furthermore, we experimented whether each error was *masked* (would have had no impact on the running software output) by running all experiments again with recovery disabled in the post-synthesis netlist.

Where to inject an error? We injected errors on 30,707 wires in the entire processor, including those wires in the circuitry we added for error tolerance. The only exception is that we did not inject errors inside storage structures (caches, register file), but did inject errors on outputs of storage structures.

How many errors to inject at once? We injected only one error at a time. This error may fan-out through the circuitry and affect multiple downstream gates and flip-flops.

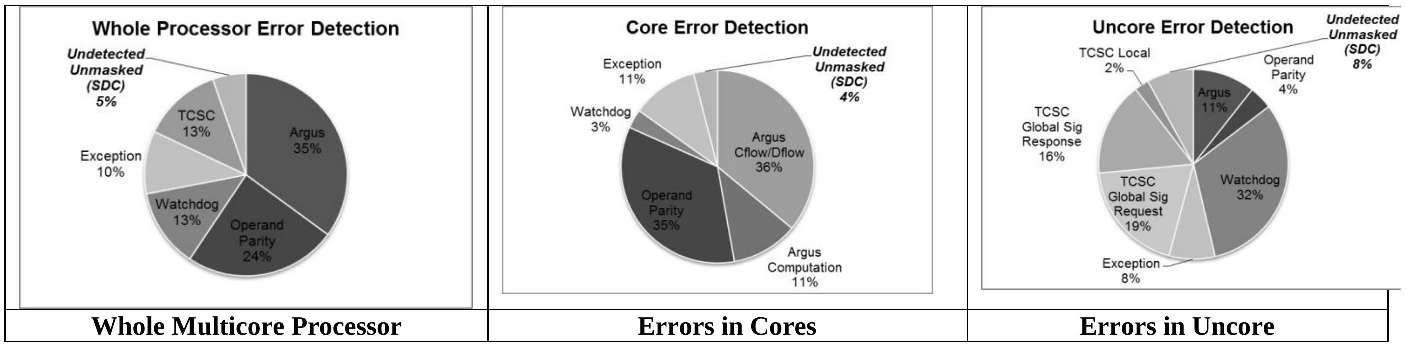

Figure 3. Error Detection

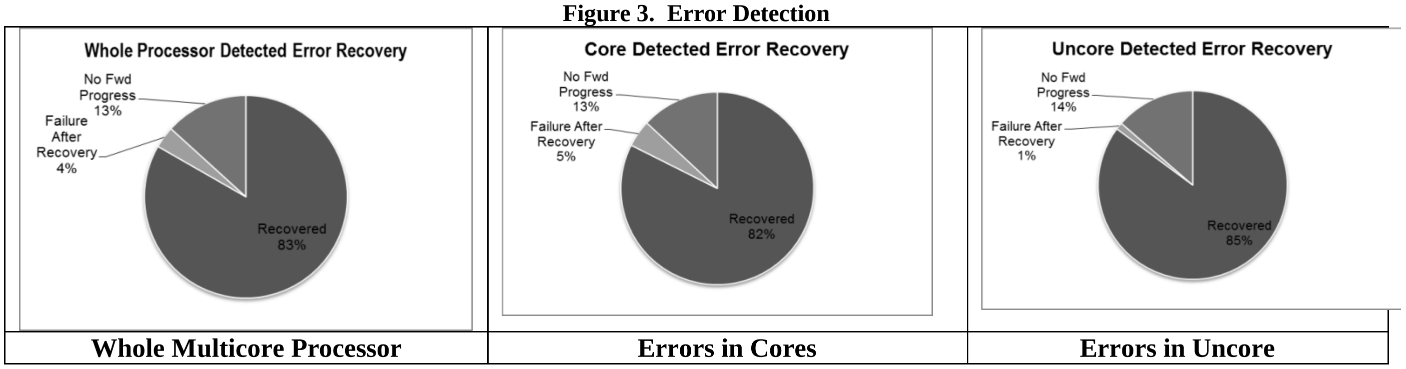

Figure 4. Recovery of Detected Errors

When to inject an error? For each wire, we performed 279 experiments in which we injected only a single error per experiment, but at different times during the benchmark run uniformly distributed after program initialization.

How do we model a soft error? We model a single event upset (SEU) transient error by forcing the selected wire to flip for one clock cycle. On the next cycle, the wire is released to be driven normally.

A. *Error Detection*

The results in Figure 3 show the error detection capability of our processor. Because the vast majority (95.2%) of injected transient errors are masked, we must run a vast number of experiments to obtain statistically significant results, and we must be careful not to skew the results. (A system can "tolerate" 100% of masked errors.) In the figure, we consider all errors except those that are masked+undetected, which we refer to as *unobserved errors* and filter out of the results in this section. All other errors are *observed errors* (including errors that are masked+detected) and we include them here.

From left to right, we plot the results for errors across the entire processor, errors just in cores, and errors just in the "uncore" (everything but the cores). The key wedge of each graph is the undetected+unmasked errors, because these are silent data corruptions (SDCs). Overall, only 5.3% of the errors lead to SDCs. If we had considered all errors, including unobserved errors, then only 0.5% of injected errors lead to SDCs. Argus and TCSC detect all but a small fraction of unmasked errors. Furthermore, these results are quite pessimistic, because we are not injecting errors in storage structures. All single-bit errors in storage structures would be detected by parity, and the storage structures comprise a large fraction of the processor area.

B. *Error Recovery*

We now examine the fraction of the detected errors that were recoverable using checkpoint/recovery. We consider all detected errors, regardless of whether they are masked or not, because the processor attempts to recover from all detected errors. If we fail to recover from a masked+detected error, that is a problem, so we must consider masked errors in this experiment. We ran an experiment with error recovery disabled in which we discovered that 57% of detected errors would ultimately have been masked.

In Figure 4, we present the results for error recovery. On the left are the full multicore processor, before subdividing the results by where the errors were injected. Across the entire processor, 83.3% of detected errors are successfully recovered. The results for errors detected by Argus and TCSC are fairly similar. Of the errors that are not recoverable, the majority are errors that stall forward progress by causing an unending series of detection/recovery events, which is preferable to SDCs.

VIII. EVALUATION OF COSTS

A. *Area*

We used Synopsys CAD tools to floorplan and lay out multicore processors both with and without error tolerance in the Nangate 45nm CMOS technology library [29]. For storage structures, such as caches, we used a modified version of Cacti 4.1 [30] to estimate area, energy and power. The results, plotted in Figure 5, show that the combination of Argus, TCSC, and checkpoint/recovery incurs an area overhead of only 16%. The largest area cost is for checkpoint/recovery, because of the size of the checkpoint logs. An overhead of 16% in an academic design is quite small, especially considering how small the baseline itself is.

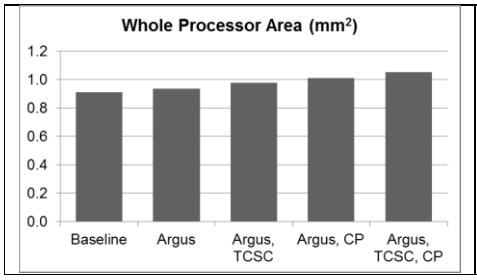

Figure 5. Area of Entire Multicore Processor

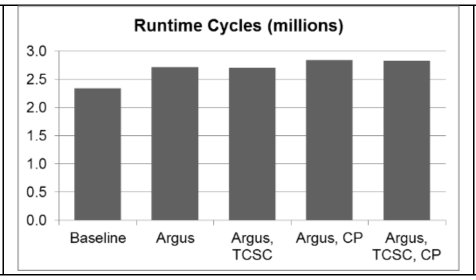

Figure 6. Impact on Runtime

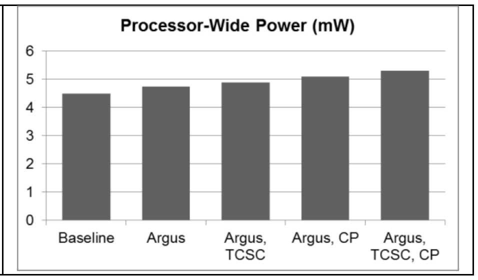

Figure 7. Processor-Wide Average Power Consumption

B. *Performance*

We did not observe a meaningful difference in clock periods (not graphed), which were within the "noise" of the CAD tools (<4% difference). We did expect and observe additional cycles due to the following factors: First, Argus adds signature instructions (special NOPs) to the binary. Second, TCSC requires each cache to issue an extra explicit coherence request to evict a read-only (Shared) block. Third, when a checkpoint is taken (which occurs every 500 instructions on average), the system must go through the process described in Section IV.

To determine the impact on cycle count—and thus runtime, because the clock periods are approximately equal—we ran our benchmark in a Verilog simulator with and without the error tolerance mechanisms. The runtime overhead in Figure 6 for the completely error tolerant processor is approximately 20%.

C. *Power*

We evaluated power consumption by running the benchmark on the floorplanned processors and, after program initialization, dynamically recording activity in the synthesized logic and black box RAMs. The CAD tools considered the processor floorplan and circuit parasitics when computing power. For SRAM storage arrays, we used Cacti's static leakage and dynamic energy per access.

In Figure 7, we show the average power consumption of processors with and without error tolerance. The processor with complete error tolerance (Argus, TCSC, and checkpoint/recovery) has a power overhead of approximately 18%, which is comparable to the area overhead. This similarity indicates that the activity factor of the error tolerance hardware is comparable to that of the baseline hardware.

IX. Applicability to Other System Models

We cannot quantitatively evaluate every possible baseline, nor do we (as academics) have access to a suitable current industry processor, thus we apply the results more broadly.

A. *Core Model*

The biggest changes due to switching to a large high-performance core would be the impact on Argus and checkpointing. Argus is provably able to detect errors, and thus we anticipate little impact on error detection. Argus's hardware is a function of the *architectural* state of a core, but not its *micro*-architectural state. Thus Argus's hardware overheads are proportionally less for more complicated microarchitectures. Argus's performance impact would also probably be less for a wider core that is more likely to have open slots for fetching/decoding signature NOPs.

A large high-performance x86 core has somewhat more architectural state to checkpoint and has a longer latency to drain its pipeline. This could be ameliorated with a higher-performing but more costly checkpointing scheme to match the larger core with proportionally the same overhead.

B. *Coherence Protocol*

Our baseline cache coherence protocol is a simple, bus-based MOSI snooping protocol. The current trend, however, is towards directory-like protocols, even for small numbers of cores (e.g., recent chips from Intel [31] and AMD [32]).

TCSC's implementation and activity are a function of the coherence permission changes that occur and not a function of *how* those coherence permission changes occur. Our TCSC implementation would behave identically with a directory protocol, even at greater throughput.

C. *Number of Cores*

If we continue to assume a single error model, then having more cores would have negligible impact on error detection capability, because (a) TCSC's error detection capability is not a function of the number of cores and (b) Argus is implemented independently on a per-core basis.

X. Related Work

There is a vast body of prior work in fault tolerant computer architecture [33], and we cannot cover it all. Instead, we focus on the most relevant prior work.

Core error detection. Some promising approaches include DIVA [34], redundant multithreading [1][2], and software redundancy [35]. Any of these schemes could be used to functionally replace Argus in our processor, but at higher cost.

Memory system error detection. One other approach is dynamic verification of memory consistency (DVMC) [36][37], which is similar to TCSC. For a system that supports sequential consistency, DVMC is equivalent to TCSC; for other consistency models, DVMC is more complete and can detect errors missed by TCSC. The other approach is to re-design the cache coherence protocol such that all errors can be detected using a set of timeouts [38].

Error recovery. We based our checkpoint/recovery scheme on two prior approaches with straightforward implementations, CARER [22] and ReVive [23]. More complicated schemes, such as SafetyNet [39], exist for streamlining certain aspects of checkpoint/recovery.

XI. Conclusions

We have developed a proof-of-concept processor in RTL to show that an academic group can build a multicore processor that tolerates soft errors across the entire processor at low cost, with overheads that are less than those of other processor-wide approaches. Furthermore, we expect that an industrial engineering team could achieve better results with a more optimized implementation and a larger baseline processor.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-111-5367. We thank Jeremy Walch for his build, simulation, and synthesis tool flow.

References

[1] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *Proc. of the 29th Int'l Symp. on Fault-Tolerant Computing Systems*, 1999, pp. 84–91.

[2] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Implementation of Redundant Multithreading Alternatives," in *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, 2002, pp. 99–110.

[3] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[4] P. S. Magnusson, et al., "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[5] M.-L. Li, et al., "Accurate Microarchitecture-level Fault Modeling for Studying Hardware Faults," in *Proc. of the Fourteenth Int'l Symp. on High-Performance Computer Architecture*, 2009.

[6] M.-L. Li, et al., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proc. of the Thirteenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] D. Lampret, *OpenRISC 1200 IP Core Specification, Rev. 0.7*. 2001.

[8] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way Multithreaded SPARC Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Apr. 2005.

[9] M. Shah, et al., "UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC," in *Proc. of the IEEE Asian Solid-State Circuits Conf.*, 2007, pp. 22–25.

[10] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C–28, no. 9, pp. 690–691, Sep. 1979.

[11] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proc. of the 40th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[12] A. Meixner and D. J. Sorin, "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures," in *Proc. of the Twelfth Int'l Symp. on High-Performance Computer Architecture*, 2007, pp. 145–156.

[13] X. Delord and G. Saucier, "Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors," in *Proc. of Int'l Test Conference*, 1991, pp. 936–945.

[14] E. Borin, C. Wang, Y. Wu, and G. Araujo, "Software-Based Transparent and Comprehensive Control-Flow Error Detection," in *Proc. of the Int'l Symp. on Code Generation and Optimiztion*, 2006.

[15] N. J. Warter and W.-M. W. Hwu, "A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking," in *Proc. of the 20th Int'l Symp. on Fault-Tolerant Computing Systems*, 1990, pp. 442–449.

[16] J. Carretero, et al., "End-to-End Register Data-flow Continuous Self-test," in *Proc. of the 36th Annual Int'l Symp. on Computer Architecture*, 2009, pp. 105–115.

[17] A. Meixner and D. J. Sorin, "Error Detection Using Dynamic Dataflow Verification," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2007, pp. 104–115.

[18] F. F. Sellers, M.-Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company, 1968.

[19] P. J. Eibl, A. Meixner, and D. J. Sorin, "An FPGA-Based Experimental Evaluation of Microprocessor Core Error Detection with Argus-2," in *Proc. of ACM SIGMETRICS*, 2011.

[20] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token Coherence: Decoupling Performance and Correctness," in *Proc. of the 30th Annual Int'l Symp. on Computer Architecture*, 2003.

[21] M. M. K. Martin, et al., "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[22] D. B. Hunt and P. N. Marinos, "A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique," in *Proc. of the 17th Int'l Symp. on Fault-Tolerant Computing Systems*, 1987.

[23] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," in *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, 2002, pp. 111–122.

[24] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," Department of Computer Science, Carnegie Mellon University, CMU-CS-96-181, Sep. 1996.

[25] S. M. Z. Iqbal, Y. Liang, and H. Grahn, "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems," *IEEE Comput Arch. Lett*, vol. 9, no. 2, pp. 45–48, Jul. 2010.

[26] C. Constantinescu, "Experimental Evaluation of Error-Detection Mechanisms," *IEEE Trans. on Reliability*, vol. 52, no. 1, Mar. 2003.

[27] C. Constantinescu, "Using Physical and Simulated Fault Injection to Evaluate Error Detection Mechanisms," in *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 1999, pp. 186 –192.

[28] A. Pellegrini, et al., "CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework," in *Proc. of the IEEE Int'l Conf. on Computer Design*, 2008.

[29] Nangate Development Team, "Nangate 45nm Open Cell Library." 2012.

[30] S. Thoziyoor, D. Tarjan, and N. P. Jouppi, "Cacti 4.0," 2006.

[31] R. Singhal, "Inside Intel Next Generation Nehalem Microarchitecture," in *Hot Chips 20*, 2008.

[32] P. Conway, et al., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, Apr. 2010.

[33] D. J. Sorin, *Fault Tolerant Computer Architecture*. Morgan & Claypool Publishers, 2009.

[34] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proc. of the 32nd Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 1999, pp. 196–207.

[35] G. A. Reis, et al., "SWIFT: Software Implemented Fault Tolerance," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2005, pp. 243–254.

[36] A. Meixner and D. J. Sorin, "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2006, pp. 73–82.

[37] K. Chen, S. Malik, and P. Patra, "Runtime Validation of Memory Ordering Using Constraint Graph Checking," in *Proc. of the Thirteenth Int'l Symp. on High-Performance Computer Architecture*, 2008.

[38] R. Fernandez-Pascual, et al., "A Fault-Tolerant Directory-Based Cache Coherence Protocol for Shared-Memory Architectures," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2008.

[39] D. J. Sorin, et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, 2002, pp. 123–134.