RC25220 (W905-046) May 18, 2009
Computer Science

IBM Research Report

SoftBeam: Precise Tracking of Transient Faults and Vulnerability Analysis at Processor Design Time

Michael Gschwind, Valentina Salapura, Catherine Trammell
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Sally A. McKee
Chalmers University of Technology
Göteborg, Sweden

SoftBeam: Precise Tracking of Transient Faults and Vulnerability Analysis at Processor Design Time

Michael Gschwind, Valentina Salapura, Catherine Trammell
IBM T.J. Watson Research Center
Yorktown Heights, NY

Sally A. McKee
Chalmers University of Technology
Göteborg, Sweden

***Abstract*—To study system reliability of a next-generation system, we undertake a soft error vulnerability study for a next-generation microprocessor design. Starting from design data for the entire processor, we extend the microprocessor verification methodology to study soft error propagation through microprocessor logic into the architected processor state. We use soft error injection into randomly selected latch bits to (1) identify areas for improvement, (2) derate technology susceptibility by architectural, microarchitectural, and logic masking resulting in increased soft error resilience; and (3) identify areas where microarchitectural data corruption can be tolerated as performance degradation without impact on correctness, yielding even greater soft error resilience. Based on these results, we reduce design vulnerability to soft errors by factors ranging from 2 for an execution unit to more than 32 for a memory management unit.**

I. INTRODUCTION

Soft errors, also known as transient faults or single event upsets (SEUs), are of increasing concern among computer architects. These errors manifest themselves as instantaneous inversions of the logical values held in transistors, resulting in signal corruption. Usually soft errors are generated by highly energized particles impacting the transistor substrate. The most common such high-energy particles are neutrons originating from cosmic rays and alpha particles emitted by manufacturing impurities and packaging decay. Ziegler et al. provide a comprehensive historical account of these phenomena [11].

When these energized particles interact with semiconductor atoms at a particular node, the charge distribution can be disturbed sufficiently to modify the voltage level of the node's signal. The effect is temporary and does not damage the transistor (hence the term "soft" error), but the resulting mutation in a piece of data can propagate and create a lasting inconsistency. The extent of the disturbance is largely dependent on internal device characteristics such as capacitance and voltage threshold. Larger devices with more capacitance are more robust against the creation of artificial current channels, and larger voltage ranges require a greater electron displacement to switch interpreted logic levels. As new technology generations use smaller feature sizes and lower operating voltage, a transient fault is more likely to appear. In many application areas, sacrificing performance to avoid fault manifestation is not feasible. Although the FIT (failures in time) rate of individual processors is relatively low, it compounds in multiprocessors, compute clusters, servers, and especially supercomputers, becoming nontrivial for large systems.

We investigate vulnerability of a new processor core and floating point unit, and observe how they react to transient fault injection. Most public simulators are necessarily limited by computation granularity. Fault injection in these tools can only be performed at the instruction level into large arrays such as caches and register files. Injection into individual latches is infeasible. However, large arrays are easily protected by ECC (error correcting codes) and, in fact, usually enjoy such protection in modern designs. Latches, on the other hand, such as those in the pipeline or those that hold state and configuration are equally vulnerable, and yet often unprotected. The goal of this work is to understand design resilience based on natural redundancy in microarchitectural mechanisms and design data paths, and to identify portions of the design to drive resilience improvement.

Unlike other soft error studies using generic architectural simulators, this study uses a VHDL model of the core along with industrial verification techniques. Even though the design we work with will not be a final product, our results reflect the accurate response of a real processor. The results from this study have proved valuable in ongoing processor design by capturing the propagation of SEUs into machine state, the types of errors generated, and the categories of latches with the highest susceptibility.

Our results only reflect behaviors of faults already manifested, and are completely independent of rate of fault appearance as a result of microprocessor technology and circuit designs style. We make no statement about this processor's raw likelihood of failure. Our study is a commentary on a specific design's vulnerability in the rare event that a fault occurs.

The main contributions of this work are: we (1) develop SoftBeam, a methodology that allows accurate tracking of SEU effects; (2) analyze execution failures as a result of soft errors; (3) identify soft error vulnerabilities which cause performance impact only; (4) identify opportunities for design hardening to deliver a design which is more resilient to SEU when deployed, and (5) reduce design unit vulnerability by selective hardening of the design by factors from 2 to 32.

We analyze execution failures in response to soft error injections by the failure type the SEU elicits. Due to the detailed error analysis and categorization capabilities of the underlying verification environment upon which our soft error injection work is based, this affords us the first detailed look at the types of errors caused by soft errors.

The methodology presented here was developed in parallel

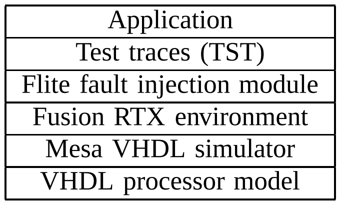

Fig. 1: The SoftBeam environment builds upon the traditional microprocessor development and extends it with the capability to simulate transient faults while the processor simulates application traces extracted from system workloads. The verification environment is used to diagnose the impact of injected faults on the correct operation of the processor.

with the development and refinement of the experimental 4-way multithreaded, 2-way superscalar in-order microprocessor with a floating-point SIMD unit which is the target of this analysis. As such, the design presented a moving target, and not a completed design that was then improved for resilience. We have attempted to capture a consistent snapshot of the design for all data shown for a point in time during the development process which reflects the methodology after it had matured during its initial use on the processor. Consequently, the design reflects many — but not all — improvements that have been made during the process of designing the processor and developing the methodology.

II. The SoftBeam Environment

To model the impact of transient faults on a microprocessor we have developed SoftBeam, a methodology and environment for design time soft error resilience exploration and hardening. We model soft errors by injecting a transient fault into latches during the simulation of the VHDL model of a processor core. Our approach mimics the post-fabrication hardware irradiation experiments performed using proton beams by Sanda et al. [8], but in a software simulation environment at design time, i.e., software-based beam experiments.

SoftBeam uses IBM's internal cycle simulator and verification methodology to provide fine-grained error detection. Using the fine-grained verification methodology to diagnose errors gives us the capability to identify exactly when a soft error injected into the logic first enters the architected state, or violates microarchitectural constraints. This allows separation of logic and microarchitectural masking from application-level masking and direct study of the impact of transient faults at the design level, unlike prior work which compares manifestations of soft errors at the chip boundary [7].

The SoftBeam environment for soft error vulnerability analysis makes use of several design verification tools, as illustrated in Figure 1. The VHDL processor model is simulated using Mesa, IBM's cycle level simulator. We use a simulation driver called RTX (runtime executable) to drive the Mesa simulator and to detect when injected transient faults cause processor state corruption. The RTX driver feeds test traces (TSTs) to drive microprocessor execution. In addition to instructions, TSTs include complete information about computed results and a snapshot of the architectural state before and after each instruction.

RTX monitors all simulation events such as state modification, cache transactions, and instruction issue and retirement. The observed simulation events are dynamically verified against the TST, which is generated by Gpro [1], a testcase generation tool for processors. Upon encountering an illegal operation or invalid data value, RTX reports the error and terminates simulation. While RTX and GPro are usually used to diagnose design errors, SoftBeam uses these tools to pinpoint the exact point where an injected fault becomes an unrecoverable error.

RTX is implemented in the Fusion environment which offers interfaces between VHDL and C, with a variety of verification support for RTX developers. We integrate our own error injection code written in Fusion to access latch signals by name and design hierarchy, read their dynamic values and write back modified values reflecting the occurrence of a transient fault.

In our SoftBeam environment, we create a unique top level simulation and error injection control file for every simulation we spawn. As Mesa is launched, the soft error injection module instructs the simulator when to suspend execution to inject a fault. When this point is reached, control temporarily yields to our transient fault injection module, where soft error injection and data collection occurs. After the fault is introduced, control passes back to the Mesa simulator and execution resumes. Figure 2 illustrates this process.

When randomizing the time of injection, we limit the range to the duration of the main loop in each program, such that no fault is ever injected during the uncharacteristic phases of the code. We divide the simulation into three phases: a warmup phase, an active injection candidate phase, and a latent error discovery and shutdown phase. We skip an initial portion of the simulation to allow the design to reach a steady state. The injection module picks a cycle during the steady state "injection candidate phase" to inject a random latch. The last section of steady state simulation is not a candidate for injection to allow possibly long term latent errors to manifest themselves. We also perform an end-of-simulation state check to compare the architected processor state against the expected simulation results.

In our environment, the soft error injection module provides generic injection and data collection capabilities under the control of a configuration file. Individual latches are often grouped to hold multi-bit data, thus "latch" generically describes such groupings of variable width. In addition to latch name, we specify the bit to corrupt within the group. Our list of fault injection candidate latches comprises all latches in the model, excluding arrays with existing soft error protection (such as register file internals) and elements that are disabled during normal operation (such as ABIST, or array built-in self test).

Reflecting that soft error upsets occur as isolated incidents, we inject one transient fault per simulation and allow the program to run to completion. This lets us identify a solid cause-and-effect between each injected fault and any resulting

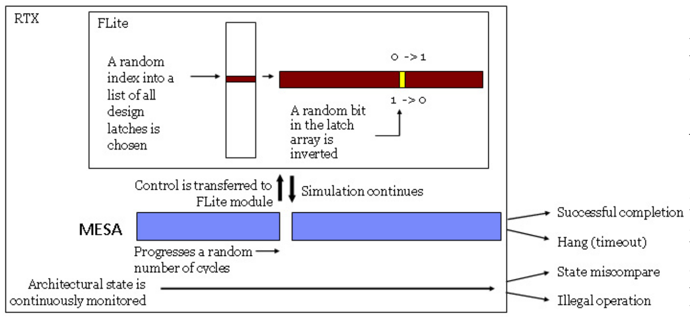
RTX
FLite
A random index into a list of all design latches is chosen
0 -> 1
1 -> 0
A random bit in the latch array is inverted
Control is transferred to FLite module
Simulation continues
MESA
Progresses a random number of cycles
Architectural state is continuously monitored
Successful completion
Hang (timeout)
State miscompare
Illegal operation

Fig. 2: Error injection is performed during VHDL simulation by gaining control during VHDL simulation and inverting the contents of one or more latch bits in the simulated model.

TABLE I: TST instruction mix

Application	Control	Floating Point	Integer	Memory
QCD	6%	78%	12%	4%
DGEMM	2%	55%	15%	28%

simulation error. We run over 20,000 simulations for each application, ensuring statistical significance accurate to 0.7% with 95% confidence.

III. Experimental Setup

Using the main loop kernels for two major floating point applications, we create four distinct TSTs reflecting our target workloads with representative processor usage patterns and different levels of compute intensity. Because TSTs typically do not represent real applications, but are generated by a randomizing test case generator, no infrastructure exists to transform applications to TST format. We convert applications into two TSTs: one with a single thread of execution, and one with four threads. The multithreaded version in each case is a four-way replication of the code on different data set. Translating full applications would yield TSTs over 800K lines. We therefore use only the main application kernel, and we execute the most significant portions of these applications within a few thousand instructions.

Our selected applications are QCD and DGEMM, two floating point intensive kernels representative of key system workloads. QCD models lattice quantum chromodynamics, a mainstream benchmark with a typical workload. We chose DGEMM, a dense matrix multiply kernel, as a worst-case application, with efficient pipeline utilization and above average throughput — a “stress test” to explore the impact of soft error upsets on the design under heavy utilization. QCD is about twice as large as DGEMM, offering more varied bypass scenarios and dependence scenarios. Table I provides the relative contributions of instruction categories found in each application. Loops have already been unrolled. Although targeted for a floating point study, there is no shortage of integer instructions in either TST, and they both sufficiently tax the entire processor to obtain a comprehensive set of results.

When control passes to the fault injection module, we can probe the processor state to examine conditions surrounding a fault injection point, to aid fault propagation analysis. As example, we probe clock gating information, so we know whether a latch which had an error injected was active. Soft error injection into idle clock-gated latches should never impact the correctness of a computation, whereas soft error injection in a functionally gated latch (i.e., a latch gated to hold a defined value for several cycles) often reflects corruption to key state, such as the state in an architected register.[1]

We randomize injections uniformly across latches, ignoring width. In the presented data, we weight latches by their width to prevent skewing of results in favor of smaller latches. This assumes that latches exhibit homogeneous behavior for all bits; since they are grouped for the very reason that they share a common function, this is a reasonable compromise between broader coverage over different latch groups and assuring proper representation of large latches.

The simulation seed is always set to the same constant, ensuring that hardware design variables such as memory latencies and cache miss rates are identical across simulations. This also ensures determinism and reproducibility in our results.

IV. Results

Figure 3 provides a distribution of over 20,000 simulations of fault injections for each of the four test cases. We classify the results by clocking state of the injected latch at the time of the fault injection, and by the outcome of the simulation run, i.e., whether the fault injection caused state corruption.

Runs that fail when un-clocked represent the fraction of injections into state-holding latches where RTX detects their presence immediately. Our results show that between 83% and 91% of all faults have no impact on architected state. However, even simulations that fail RTX may not cause corrupt output: a corrupt state-holding element may never be used, and thus may never propagate its fault.

Multithreaded applications cause a significant increase in failure rates compared to singlethreaded versions. DGEMM and QCD experience an increase in failure rate by 46% and 66% percent, respectively. Comparing the failure rates between singlethreaded versions of QCD and DGEMM, their failure rates are remarkably similar despite the difference in their CPI. Thus, thread-level parallelism affects error rates more than instruction-level parallelism for both of these applications. This is a result of higher overall activity of latches when running multiple threads compared to a single thread. On the other hand, higher activity due to higher instruction-level parallelism (with a 34% better CPI for DGEMM compared to QCD) results only in a 2.4% difference in failure rates.

We also break down the results according to major units of the processor: the instruction unit, the integer execution unit, the memory management unit, the floating point unit, and the pervasive unit (primarily for debugging). Table II lists these

[1]Aggregate statistics on clock gating also allow to track progress towards power reduction goals as the design evolves. Clock gating statistics presented here do not reflect the final clock gating achieved, and hence are not a reflection of activity in the processor units.

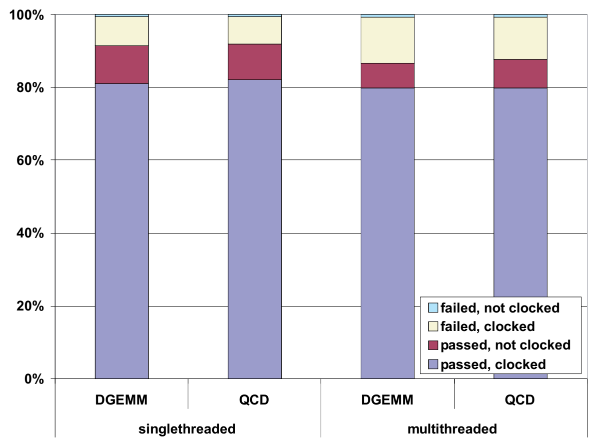

Fig. 3: Transient fault injection propagation rates into architected and microarchitected state and operations, by clock activity status at time of fault injection.

TABLE II: Processor Units

Name	Function	% Latches	% Latch Bits
IU	Instruction Unit	26	23
XU	Integer Execution Unit	56	44
MMU	Memory Management Unit	10	11
FPU	Floating Point Unit	7	20
PCU	Pervasive Unit	1	2

processor units and their abbreviated names. The table gives the relative size of each processor unit as a percentage of both total number of latches and total number of latch bits (i.e., instantiated latches weighted by bits in that latch). The pervasive unit PCU is small, and we selectively report results due to space constraints.

Figure 4 breaks down the total injections by unit and by whether the injection caused an execution failure. The integer execution unit XU (comprising both fixed point execution and load/store functions, including the data cache) dominates latch bit count in the core — it accounts for 56% of all latches and 44% of overall latch bits in the processor core. With a dominant share of latch bits, the execution unit is the statistically likeliest target for soft errors to occur, or the selection of a latch bit to inject at random. Consequently, most errors are injected in the execution unit, and it dominates the percentage of passed and failed runs in Figure 4. Similarly, other units appear proportionally to their latch bit count, as SoftBeam uniformly sprays soft errors into the processor core, much in the same way that physical chips are affected by cosmic particles.

A. Failure Rates by Component

Figure 5 plots the percentage of runs that fail due to injections in the specified processor unit. These results were collected after a number of design improvements were made, as discussed below. Failure percentages refer to the subset of runs that had a fault injection in the specified unit, not of all failed runs across all units. Data presented have been weighted

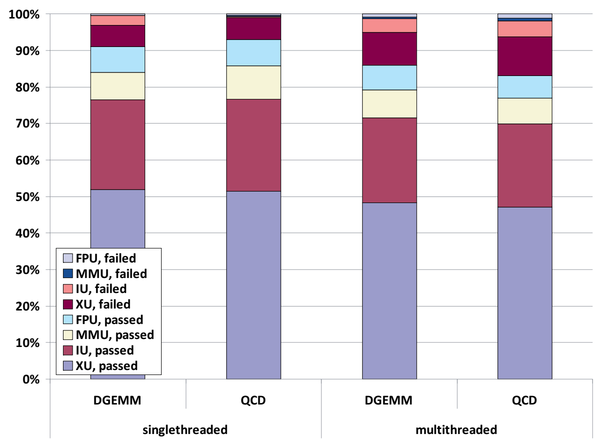

Fig. 4: Distribution of fault injections by function unit, and impact of SEU on application kernels.

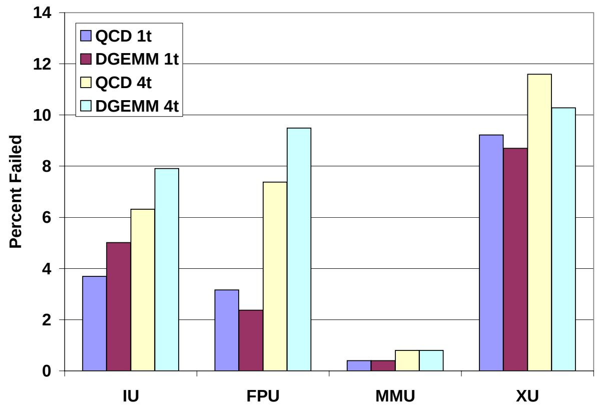

Fig. 5: Comparison of unit failure rates

by latch width.

The MMU is the most resilient unit, displaying no more than 3% error for any TST. The XU, on the other hand, is the largest source of error, with an average fail rate of 10%. The IU and FPU also reveal substantial propagation of injected soft errors, although these are much more dependent on the application and number of threads.

Since the XU is the unit most susceptible to failure, we looked more closely at unit internals. Figure 6 plots failure rates of sub-components within this unit. The XU has a group of unprotected special purpose registers that mostly fail when not clocked. Within the XU, the completion logic latches in XU_CPL and the special purpose register logic in XU_SPR are the most vulnerable subunits.

Other processor units display similar results when comparing all injections to only clocked ones, each with the exception of a single sub-component comprised of unprotected special purpose registers.

B. Error Detection Times

As part of validating our methodology, we explored the time it takes for an error result triggered by a soft error injection to surface in the processor state. This curve allows us to

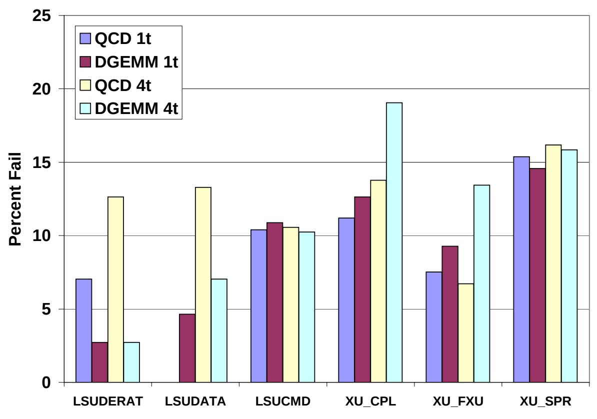

Fig. 6: Execution Unit (XU) subunit failure rates

establish how long to run experiments to be confident that all – or substantially all – errors that will eventually manifest themselves have surfaced. In addition to giving us confidence that detection of latent errors has converged, these experiments also give interesting insights into error propagation behavior and the behavior of possible latent long term corruption.

Figure 7 plots cumulative histograms of runs that detect errors in each time interval. The data include only errors detected before program completion, omitting hangs and end-of-test miscompares (both of which are always diagnosed at the end of the test run). The figures shown here focus only on injections into clocked (active) latches. The majority of errors are detected soon after injection. Eighty percent of failures are discovered within 12-25 cycles for singlethreaded and multithreaded DGEMM, and within 49-108 cycles for singlethreaded and multithreaded QCD. The data demonstrate that the probability of error discovery at any given point is application dependent. The fault injection runs with the QCD kernel exhibit more latent long range manifestation of faults, with 5% of QCD injection-induced corruptions still not having manifested themselves after 2500 cycles.

C. Latch Nonuniformity

Thus far we have assumed all bits within a latch grouping perform identically, or close to it. While this represents the majority of latches (e.g., instructions, integer data, cache busses and so forth), we wanted to explore the behavior of latches in the design that may not conform to this assumption. For instance, floating point data may be subject to rounding, and latch bits carrying least significant portions of the data will be less vulnerable, since corruptions to those bits will be lost along with the rest of the discarded precision. To test this hypothesis, we execute a separate set of simulations, this time injecting only into a single latch suspected of exhibiting nonuniformity. This latch holds the 110-bit result of a multiplication operation in a fused-multiply-add datapath that is eventually reduced to a 52-bit output mantissa, resulting in a large portion of the latch contributing only to rounding.

Figure 8 depicts the behavior of a heavily used floating point pipeline latch, capturing failure propagation rates for fault injections when the latch is clocked. Failure rates for each bit in the latch are plotted with the most significant bit on the left. The plots show that the SER is not uniform across this particular latch type. The most significant half of the latch has, on average, a 10.4% clocked error propagation rate for singlethreaded QCD and a 23.0% clocked error propagation rate for multithreaded QCD (propagation rates are 1.9% and 9.7%, respectively, when considering injections without regard to clocking status of the latch), but fault injections into the least significant half of this latch do not propagate. There is a well defined boundary between the two halves of the latch, corresponding to the point where the floating point intermediate result is commonly rounded.

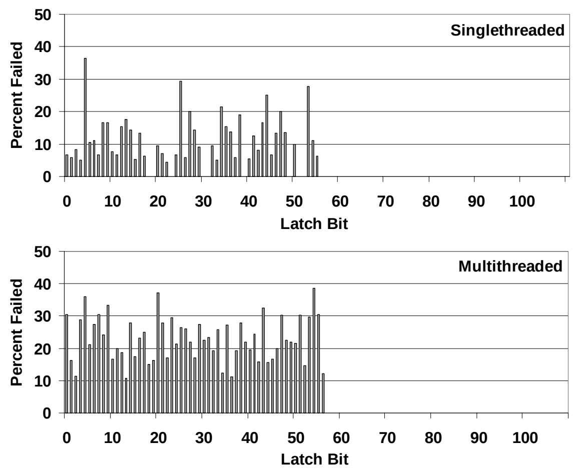

Fig. 8: Failure rates across floating point pipeline latch MAD0.FADD.EX4_RES_LO_LAT for injections while latch is clocked, for singlethreaded and multithreaded QCD. Fault injections into low-order mantissa bits are unlikely to impact correct floating point datapath operation because they only impact result rounding.

V. Failure Analysis and Design Improvements

A. Failure Types

We analyze SEU-induced execution failures by the failure type elicited. Due to the detailed error analysis and categorization capabilities of the underlying verification environment upon which our soft error injection work is based, this affords us the first detailed look at the types of errors caused by soft errors.

In the course of our experiments, we found that injections into clock gated latches produce either an almost instantaneous failure, by corrupting vital state (either architected state or configuration state), or no failure when injecting clock-gated, unused latches. Figure 9 depicts the most common errors that occur during the execution of each application in response to injections into active (i.e., not clock-gated) latches only.

The most significant class of failures are IBI (“instruction by instruction”) miscompares, that compare the architected state of the microprocessor, as each instruction completes. Thus, soft errors causing the data path to generate an incorrect result

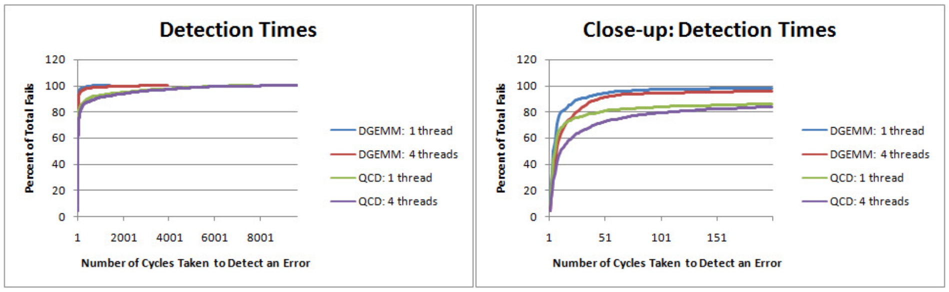

Fig. 7: Cumulative histograms: time from fault injection to detection of error manifestation

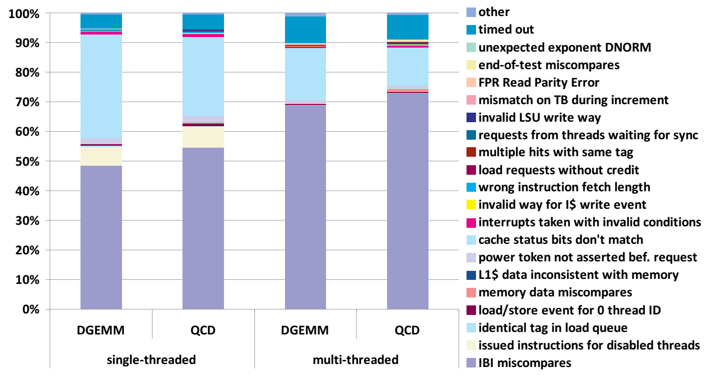

Fig. 9: Failure type distribution

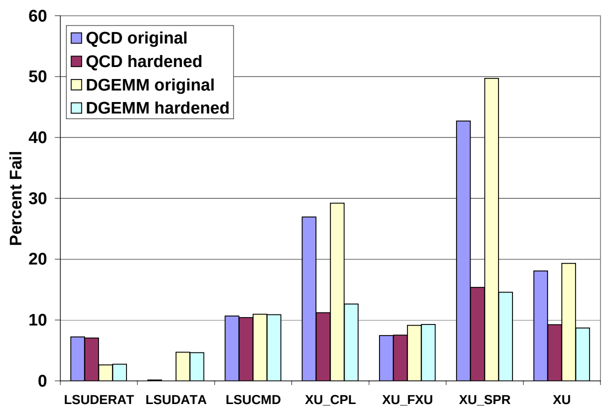

Fig. 10: Design hardening achieved a reduction of XU soft error vulnerability by more than a factor of 2.

(by corrupting the input data, the intermediate results during computation, or the output) typically manifest themselves as IBI faults.

The second significant failure category results from discrepancies found in caches indicated by the “cache status bits don’t match” failure.. Since this analysis only covers soft errors in latches, and not memory arrays (where well known detection and correction techniques such as ECC can be used), this category reflects corruption in cache management information held separately in latches, such as valid or lock bits. Because of the higher utilization of data paths — and the concomitant higher IBI error rate — in the multithreaded execution runs, the relative contribution of these errors is smaller for the multithreaded execution scenarios.

We have found that using a write-through policy can significantly reduce the impact of SEUs on valid bits. In operational systems, caches tend to be full with predominantly valid lines. Thus, SEU upsets will predominantly trigger valid bit changes that turn valid data into invalid data. This will cause spurious cache misses — however, if the cache is a write-through cache, the data can be safely retrieved from the lower level caches without any corruption of the computed results.

The singlethreaded TSTs are the only ones issuing instructions for disabled threads. This is understandable, because the multithreaded TSTs utilize all four threads, and can not try to issue instructions to disabled threads. An attempt to issue instructions to a dormant thread will most likely not result in the processor failure, but will be ignored.

Similarly, a multithreaded TST can attempt to issue instructions for the wrong thread, but we cannot detect the problem at instruction issue, since all threads are enabled. Instead, these errors manifest themselves as state miscompares or hangs, which comprise a greater percentage of total failures in the multithreaded TSTs.

B. Design Hardening

The soft error vulnerability methodology described here pinpointed critical design areas that were particularly vulnerable. Based on our injection experiments, we generated detailed statistics of the impact of a soft error of each latch in the design. Based on these results, designers were able to mitigate vulnerability by selecting applicable SEU mitigation strategies, from protecting data with ECC and/or duplication, using write-through caches to allow recovery from incorrect cache line invalidations due to SEUs, and so forth.

As an example, one area for improvement we identified was common configuration and state latches. These included configuration latches with high fanout and architected state that was not kept in register file arrays, and hence did not benefit from protection techniques benefitting the data stored in these arrays.

To increase reliability of the entire design, we selectively and iteratively replaced latches which had high propagation into architected state with more robust latches with a hardened design style until we reached an aggregate reliability threshold. Figure 10 shows an example of the improvements achieved by applying the design improvements described here to key resources in the XU (but not accounting for the resilience due to valid bits flipping from valid to invalid in write through caches), for singlethreaded DGEMM and QCD benchmarks. By identifying critical latches and selectively hardening these latches, we were able to cut the failure rate more than in half. We achieved similar or better results in several other units, such as the MMU, where hardening a few key memory management registers resulted in decreased unit vulnerability to fault injection from approximately 16% to under 0.5%.

VI. Related Work

Wang et al. [10] perform fault injection into latch outputs of an Alpha-based Verilog processor model. They verify microarchitectural state at the *end* of program execution, detecting the presence of propagated errors only when they appear on processor pins. Pin checking does not allow separation of microarchitectural from application resilience, nor detect fault conditions that degrade performance but do not introduce failures, such as branch mispredictions.

Sanda et al. [8] study soft error resilience of the IBM POWER6 processor using four experiments: proton beam irradiation, fault injection into architected state of the Mambo simulator, statistical fault injection (SFI) into latches of a hardware-emulated RTL model, and neutron beam irradiation. The SFI experiment, which most closely resembles our own methodology, is described in detail by Ramachandran et al. [6]. Using hardware acceleration sacrifices the fine-grain error detection capability of our infrastructure.

Blome et al. [3] study logic masking rates and error propagation using a Verilog model of an ARM processor. Bronevetsky and de Supinski [4] study algorithm-level fault tolerance with checkpointing. Other work has studied architectural vulnerability factors (AVF) of various structures within a processor [5], [2], [9] to assess probabilities of architectural fault masking.

VII. Conclusions

We have introduced the SoftBeam methodology to analyze logic and microarchitectural vulnerability to soft errors based on a microprocessor design time verification environment. This approach allows separate analysis of soft error derating factors at the circuit level, the microarchitectural/logic level, and the application levels.

SoftBeam is uniquely capable of capturing the microarchitectural behavior of soft errors in detail, and we demonstrate its results in a new processor core. We use a verification tool combined with detailed, verification-oriented forms of real applications to observe design behavior in the presence of individual soft errors at latch outputs. We find that less than 20% of faults propagate into the microprocessor state.

We have found that the integer execution unit is the most vulnerable processor component, and that multithreaded execution is more susceptible to SEUs than singlethreaded executions. We have found cache directories of write-through caches to offer SEU resilience by converting many corruptions in a directory into cache misses and corrective reloads. Thus, soft errors in the cache directory together with branch prediction represent a class of performance degrading SEU impacts.

Finally, this work has documented the use of transient fault injection during processor design and verification as part of a holistic approach to improving transient fault resilience in a microprocessor. Using SoftBeam, we were able to project soft error vulnerability of a processor, and make design for transient fault resilience an integral part of the design process. Based on this integrated methodology, we have selectively hardened key design parts to significantly improve overall processor resilience, with vulnerability of key units being improved by factors from 2 to over 32.

The authors would like to thank Emanuel Gofman, Anatoly Koyfman, Mark Kupferschmidt, Thomas Roewer, Brian Thompto, and Charles Wait for their help with GPro test case generation, the RTX environment, soft error injection with Fusion and the FLite environment, and with model build.

This work has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the U.S. Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331.

References

[1] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov. Industrial experience with test generation languages for processor verification. In *Proc. ACM/IEEE 41st Design Automation Conference*, pages 36–40, June 2004.

[2] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing Architectural Vulnerability Factors for Address-Based Structures. In *Proc. 32th IEEE/ACM International Symposium on Computer Architecture*, pages 532–543, June 2005.

[3] J. Blome, S. Mahlke, D. Bradley, and K. Flautner. A Microarchitectural Analysis of Soft Error Propagation in a Production-Level Embedded Microprocessor. In *Proc. First Workshop on Architectural Reliability*, Nov. 2005.

[4] G. Bronevetsky and B. de Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proc. 22nd ACM International Conference on Supercomputing*, pages 155–164, June 2008.

[5] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. IEEE/ACM 37th Annual International Symposium on Microarchitecture*, pages 29–42, Dec. 2003.

[6] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. Statistical Fault Injection. In *Proc. International Conference on Dependable Systems and Networks*, pages 122–127, June 2008.

[7] G. P. Saggese, N. J. Wang, Z. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6), 2005.

[8] P. Sanda, J. Kellington, P. Kudva, R. Kalla, R. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. Jones. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development*, 52(3):275–284, May 2008.

[9] K. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic Prediction of Architectural Vulnerability from Microarchitectural State. In *Proc. 34th IEEE/ACM International Symposium on Computer Architecture*, pages 516–527, June 2007.

[10] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *Proc. International Conference on Dependable Systems and Networks*, pages 61–70, July 2004.

[11] J. Ziegler, H. Curtis, H. Muhlfeld, C. Montrose, B. Chin, M. Nicewicz, C. Russell, W. Wang, L. Freeman, P. Hosier, L. LaFave, J. Walsh, J. Orro, G. Unger, J. Ross, T. O'Gorman, B. Messina, T. Sullivan, A. Sykes, H. Yourke, T. Enger, V. Tolat, T. Scott, A. Taber, R. Sussman, W. Klein, and C. Wahaus. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, Jan. 1996.