Exploring the Synergy of Emerging Workloads and Silicon Reliability Trends

Marc de Kruijf   Karthikeyan Sankaralingam
Department of Computer Science
University of Wisconsin-Madison
Vertical Research Group (vertical@cs.wisc.edu)

***Abstract*—Technology constraints and application characteristics are radically changing as we scale to the end of silicon technology. Devices are becoming increasingly brittle, highly varying in their properties, and error-prone, leading to a fundamentally unpredictable hardware substrate. Applications are also changing, and emerging new classes of applications are increasingly relying on probabilistic methods. They have an *inherent tolerance for uncertainty* and can tolerate hardware errors.**

This paper explores this synergy between application error tolerance and hardware uncertainity. Our key insight is to expose device-level errors up the system stack instead of masking them. Using a compiler instrumentation-based fault-injection methodology, we study the behavior of a set of PARSEC benchmarks under different error rates. Our methodology allows us to run programs to completion, and we quantitatively measure quality degradation in the programs' output using application-specific quality metrics. Our results show that many applications have a high tolerance to errors. Injecting errors into individual static instructions at rates of 1% and higher, we find that between 70% to 95% of those instructions cause only minimal degradation in the quality of the program's output. Based on a detailed analysis of these programs, we propose light-weight application-agnostic mechanisms in hardware to mitigate the impact of errors.

I. Introduction

Advances in semiconductor manufacturing technology have enabled consistent, progressive reductions in the size of on-chip devices, resulting in exponential growth in the number and speed of devices on chip and the overall performance of microprocessors. Through the next decade and until the end of CMOS technology this device scaling is expected to continue. However, the underlying properties of these devices are radically changing, and we are entering an era of non-ideal process scaling and unpredictable silicon technology which is causing disruptive changes at many layers of the microelectronics system stack [3], [14]. Currently computer systems are designed assuming perfection at many levels, from devices, through CAD, the microarchitecture-level and ISA. Driven by scaling, however, device-level permanent and transient errors, aging, and variability become first order constraints [15]. In the future, it may be too hard to maintain this illusion of perfection. The model of hardware being correct *all the time, on all regions of chip, and forever* may become prohibitively expensive to maintain. The technology-driven question that follows this observation is: *how can we build working hardware from unpredictable silicon?*

New classes of high-performance applications are emerging that are dominated by probabilistic algorithms used for image recognition, video search, text and data mining, modeling virtual worlds, and games [6], [16]. Many of these applications share *an inherent ability to tolerate uncertainty* and provide an opportunity to creatively utilize hardware. While they require large computational capability, they provide the possibility that hardware does not have to be always correct. Hence, the key application-driven questions are: (1) *can these applications indeed tolerate hardware uncertainty* and (2) *how can we efficiently support the massive computation needs of these emerging applications?*

A rich body of literature has focused on building a fault-free machine abstraction model and implementation in the presence of device errors [13]. In contrast, we observe that emerging applications are inherently tolerant of errors and are algorithmically designed to operate on noisy data. We make a big departure from prior work and treat hardware errors that create computation errors as effectively rendering the input data noisy. Our work is motivated by the insight that exposing device errors through the system stack can become more efficient than masking these errors when devices become highly unpredictable. The cross-over point when masking become less efficient than exposing is open to question and is highly dependent on technology constraints. Figure 1a illustrates the trade-off. This work attempts to explore an architecture space where hardware is allowed to execute with errors and these errors are simply exposed to applications and the system, allowing for higher level software decisions on *managing* these errors. While error rates are manageable today, non-ideal process scaling is likely to increase error rates and hence now is the time to explore such a design space. In this paper, we motivate the need for such an abstraction by outlining technology trends and application trends, focusing on the synergy between them.

We present a comprehensive analysis and characterization of emerging workloads and attempt to quantify their tolerance to errors. We analyze a set of PARSEC benchmarks (x264, bodytrack, canneal, streamcluster, swaptions) [2] and a futuristic realtime ray-tracer called Razor [5], [11]. We use a LLVM-based [9] toolchain to probabilistically inject errors into applications and run full applications on real hardware. We also develop application-specific quality functions that measure how an application's output degrades due to hard-

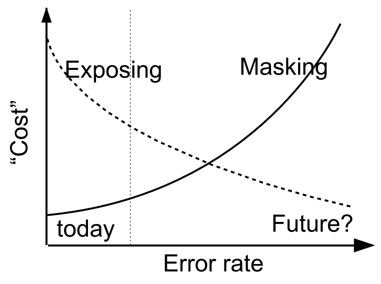


Fig. 1. The balance of masking vs. exposing errors in hardware.

ware errors. Our results show that between 70% to 95% of individual static instructions cause only minimal degradation in application output when injected with errors at rates of 1% and higher.

Application error tolerance points to several promising directions for architecture and device-level evolution. First, it presents an opportunity for building approximate microarchitectures that are designed for the common case. Second, at the device level, it expands the opportunity for incremental integration of energy-efficient analog designs. Third and more generally, allowing hardware errors can enable simplification of the overall microprocessor design process by relaxing CAD design rules, relaxing timing constraints, freeing from worst case design, and reducing foundry design rules.

The contributions of this work are:

- A scalable non-intrusive methodology to study the behavior of full applications on error-prone hardware using commodity multicore hardware.
- A detailed characterization of the tolerance to hardware errors in emerging applications.
- Simple architectural models to implement light-weight error mitigation.

The remainder of this paper is organized as follows. Section II discusses our methodology. Section III presents our application characterization. Section IV presents our architectural models for error mitigation. Section V discusses related work. Finally, Section VI concludes with implications for future systems.

II. Experimental Methodology

We simulate our applications through compiler instrumentation, using the LLVM compiler infrastructure [9]. The LLVM IR (Intermediary Representation) doubles as a RISC-like virtual ISA, and we assume a hardware machine model derived from this ISA. Our behavioral modifications are implemented through a set of compiler-based transformations that operate on the LLVM IR. In particular, we enable fault injection by inserting a conditional branch before each instruction. The branch direction is decided based on the failure probability for that instruction. On failure, a code path that contains that instruction's failure semantic is executed. Otherwise, the original instruction is executed. In both cases, control-flow merges back to the original code path.

Our approach to fault injection is significantly different from the traditional simulator-based approach, and has the following benefits and limitations:

1) *Simulation speed is extremely fast:* Instrumented applications run fast. Application slowdown is never more than 10x and is frequently in the range of 1-2x.
2) *Simulation results are architecture-agnostic:* Each LLVM instruction type is easily mapped to a particular hardware function and it is straightforward to generalize results to other hardware architectures.
3) *Parallelism and multi-core hardware:* Our infrastructure directly supports multi-threaded applications and is independent of the specific mechanisms used for parallelization.
4) *Simulations have no hardware visibility:* Speed and flexibility comes at a sacrifice to circuit- and microarchitecture-level precision.
5) *All source code is required:* A limitation is that source code is required for all regions of interest.

A. Failure Instrumentation and Analysis

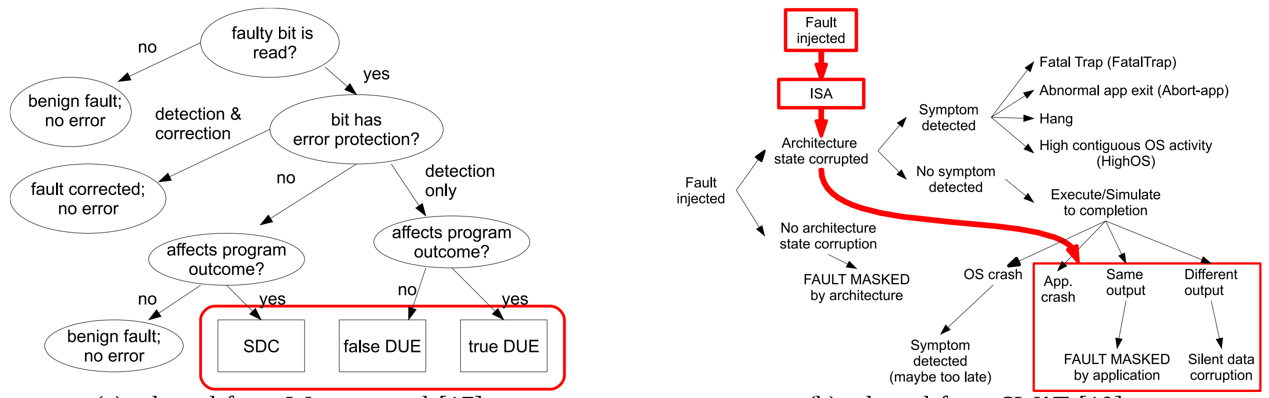
The applications we analyze are listed in Table I. For each of these applications, we perform two sets of automated analyses: one analysis, *instruction-local analysis,* looks at the failure behavior of individual static instructions, and the second analysis, *whole-program analysis*, looks at the failure behavior of groups of dynamic instruction executions. Finally, through manual code inspection we apply the data from these analyses to infer high-level conclusions about the application as a whole.

Instruction-local analysis: We analyze all instructions that constitute the top 95% of program execution time (counts are shown in column 5 of Table I), and sample the instructions that constitute the remaining 5%. We look at specific LLVM instruction types, namely: arithmetic, logical, compare, cast, branch, select, load, store, and getelementptr (a LLVM-specific instruction for indexing data structures). We label program crashes and timeouts *failures*. For non-failure executions, we feed the application output to an application-specific *evaluation function*, which produces a quality-metric that assesses the quality of the output relative to fault-free output. This quality metric is normalized to a scale of 0 to 100 for all applications, where 100 represents no degradation in quality (shown in column 7 of Table I).

Whole-program analysis: Here, we group non-failing static instructions together for simultaneous fault-injection. We ran experiments grouping all functional, memory, and control instructions together.

B. Failure Models

In our experiments, we explore two different models: *bit-flip* and *assisted*. A failure for an instruction under the bit-flip model causes a random output bit to flip from 0 to 1 or vice versa, while the assisted model assumes a best-effort attempt by the hardware to minimize failure by ensuring a deterministic failure semantic. To give two examples, under



(a) adapted from Weaver et. al [17]   (b) adapted from SWAT [10]

Fig. 2. This work in the context of other work.

Application	**Description**	**Input**	**Instruction count**			**Evaluation metric**
			Dynamic	**Static**	**Top 95% Static**	
x264	Video encoding	simlarge	457 Bil	39788	853	change in PSNR relative to raw input
razor	Real-time raytracing [8], [5]	courtyard	235 Bil	59760	15707	PSNR relative to fault-free output
bodytrack	Motion tracking in 3D space	simlarge	14.3 Bil	7274	1378	change in body pose coordinates
streamcluster	Clustering in n-dimensional space	simlarge	48.3 Bil	1539	12	change in sum of squared distances
canneal	Simulated annealing	simlarge	3.11 Bil	573	303	change in numeric output
swaptions	Monte Carlo-based partial differential equation solver	simlarge	11.2 Bil	735	187	change in numeric output

TABLE I
OUR SIX APPLICATIONS, THEIR SIZE IN TERMS OF STATIC AND DYNAMIC INSTRUCTIONS, AND THE OUTPUTS OF THEIR EVALUATION FUNCTIONS.

the assisted model a failed add outputs the first operand and a failed load outputs zero.

In the context of prior work, many previous studies have examined hardware errors using microarchitecture-level and RTL-level studies focusing on fault models targeted at levels below the ISA. Our study is unique in that it focuses on ISA-level fault analysis. We concluded that injecting faults at this level was sufficiently accurate since multiple simultaneous bitflips caused qualitatively the same types of application-level behavior. As a result, many device-level fault models including stuck-at faults, bridging faults, and timing faults are simply captured by our bit-flip model.

Figure 2a places our work in the context of basic definitions of fault injections. The focus of this study is silent data corruption and detected unrecoverable errors which have small or no effect on the output of an application. Figure 2b shows the coverage of this work in the context of the SWAT work which is the closest related [10]. Their system assumptions are driven from a different perspective of attempting to build near-perfect hardware, while we assume the opposite.

III. Application Analysis

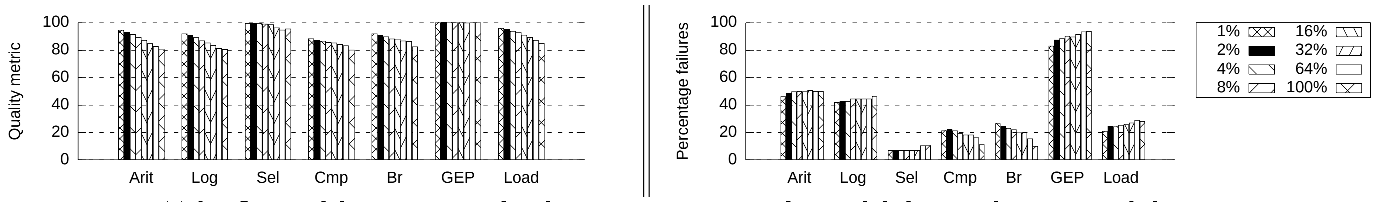
This section presents our application analysis. We take x264 as a case study and examine its error tolerance behavior in detail while making several key observations which apply to all our application. We provide an analysis of all applications in the Appendix.

Bit-flip model: Figure 3a shows the quality output on the left and percentage of instructions causing failure (crashes or timeouts) on the right assuming the bit-flip failure model, broken down by instruction type. *Observation 1:* This data confirms that bit-flips are often catastrophic and software-only recovery implies frequent intervention.

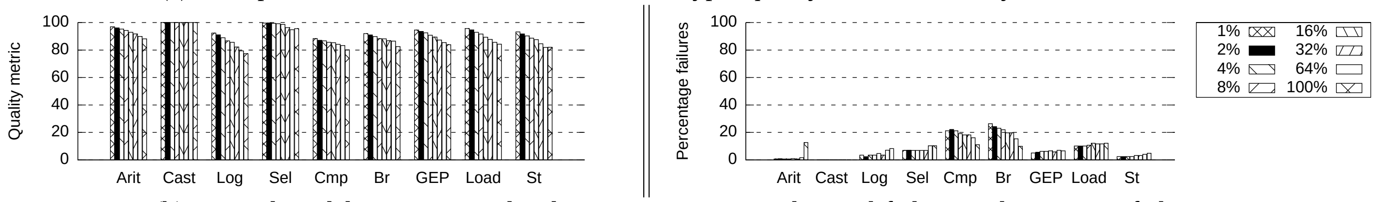
Assisted model: Figure 3b shows the quality output and instructions causing failure assuming the assisted model. It shows that there are substantially fewer failures. Interestingly, over 99% of arithmetic instructions result in no failures, and reduce program quality by no more than a PSNR (Peak Signal to Noise Ratio) difference of 20. *Observation 2:* Light-weight assistance from hardware can dramatically improve the safe execution of error-tolerant application regions.

As shown by the detailed breakdown of instructions by their execution frequency in the right column of Figure 3c, failure instructions tend to be those that are less-frequently executed. *Observation 3:* Most of the frequently executing instructions are tolerant to errors with few program failures and little reduction in program quality.

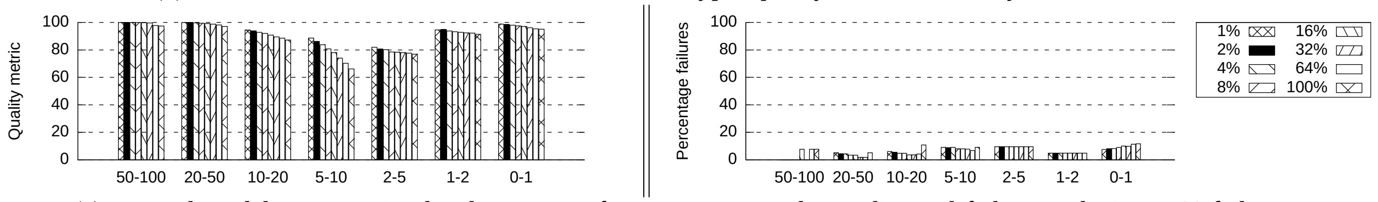
Whole-program analysis: As shown in Figure 3d, when multiple instructions are allowed to fail independent of each other, programs have little tolerance to control-flow errors. On the other hand, none of the memory or functional unit instructions, even in the presence of errors, cause failures



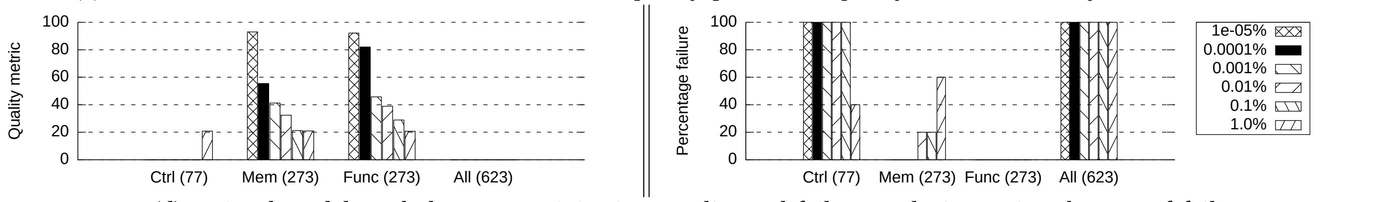
(a) bit-flip model – instruction-local instruction type: quality and failure analysis at 1% failure rate



(b) assisted model – instruction-local instruction type: quality and failure analysis at 1% failure rate



(c) assisted model – instruction-local execution frequency percentile: quality and failure analysis at 1% failure rate



(d) assisted model – whole-program injection: quality and failure analysis varying the rate of failure

Fig. 3. Graphs showing fault injection data for x264.

in combination. Furthermore, at very low error rates, they continue to have a small effect on quality. *Observation 4:* Non-control flow instructions are highly tolerant to failures on error prone hardware and do not require strong correctness guarantees.

Code inspection: By mapping the individual failing instructions back to source code, we observed that the number of those instructions in the hot region of the code is quite small. *Observation 5:* If static or dynamic code profiling techniques can identify instructions with a high probability of causing failure, targeted heavy-weight techniques can be applied to just these instructions.

Examining the application code, we found that arithmetic, select, logic, and store instructions were the most tolerant. Arithmetic instructions are heavily used to compute the sum of absolute differences between macroblocks, which is an extremely error tolerant computation. All select instructions are also used in this computation, to take the absolute value, i.e. they conditionally move a value or its negative. They are tolerant for the same reason. Logic instructions include shifts for averaging, which are highly tolerant. Shifts and ORs are sometimes used to manipulate pointer indices as well, for which they are less tolerant, and can sometimes cause a protection violation. Finally, most store instructions write data to intermediary pixel block structures used in a part of the application called motion estimation. In general, silent stores are preferable to outright data corruption for these structures.

We found conditional branch, compare, and load instructions to be the least tolerant instruction types. Virtually all conditional branch instructions are looping branches. When these loops overflow, the program often crashes due to a bounds violation. Compare instructions are used to direct conditional branch and select instructions. Hence, the impact of failing compare instructions correlates well with these instructions. Finally, load instructions are biased by their heavy use in the sub-pixel refinement motion estimation code, in which failures have a relatively high impact on quality.

A. Results in Context

Our error injection analysis with the bit-flip model showed that when executing full applications, the bit flip model is too restrictive, even for the most error-tolerant applications. More than 40% of the static instructions caused program failure. While these results may seem a little counter-intuitive or appear to contradict previous results, these are indeed consistent when placed in context. For instance, our error injection at a 0.0001% probability injects 450000 errors for one execution run for x264, executing the application to completion. Other studies have focused on a handful of single bit-flips in hardware structures and the effect on a relatively short phase of the program. **Our study provides a comprehensive analysis of what future error prone hardware may look like and attempts to study how unmodified software will perform in such an environment.**

IV. Architectural Models

In this section, we propose simple architectural models and associated abstractions and mechanisms to enable efficient hardware.

Assisted model: Architectures that implement this model can execute with errors, but must detect when errors occur and correct errors to a relatively simple default. The pipeline implementation of the assisted model requires the propagation of errors detected through every pipeline stage. The basic idea is that errors can occur anywhere in the pipeline, but must be detected and some meaningful output must be committed to the architecture state. This model can be implemented using several existing mechanisms: Razor [7], DIVA [1], and Argus [12]. Across all instruction types, this assisted hardware model reduces the number of failure instructions by 2X to 10X.

Assisted memory model: The assisted model only has a local, instruction-granularity view of errors and does not propagate error information to dependent instructions. As a result, even with the assisted model a large number of instructions cause failures. The main cause of these failures is illegal memory accesses and control-flow violations. These failures come about because an instruction that is affected by hardware errors will "safely" execute with the assisted model, but its results are not guaranteed to cause safe execution of dependent instructions. For example, a load that is dependent on a load that executed with an error, will receive as its input 0 and will invariably cause a protection violation. The assisted memory model attempts to provide for this case by including a poison bit with every register and memory location. This poison bit is propagated with each instruction execution. With the assisted memory model, stores and loads are preventing from causing a memory access violation; they default to silent stores and return zero, respectively, if the address operand is poisoned.

Assisted memory & control flow model: A third model, we call the assisted memory & control flow model, adds support for control flow recovery, by terminating functions any time a poison bit propagates to a branch instruction, as this means potentially faulty control flow. Our preliminary design for this model is for the function to immediately return when this happens and re-execute the function. A hardware implementation of this model adds moderate complexity to the processor pipeline.

V. Related Work Overview

Mukherjee presents a good overview of device reliability and its interaction with the architecture and microarchitecture [13]. Finally, Bruer and Gupta have proposed the pure hardware analog of this work, an idea they call intelligible testing, which certifies ASIC chips to be partially correct [4]. The software-hardware co-designed resilient system design [10] is the most closely related effort to our approach to allow errors in hardware. Their research attacks the slightly different problem of resilient system design with high-level detection mechanisms.

VI. Summary and Implications

Our core findings are that 1) emerging applications have a very high level of local instruction-level error tolerance, 2) with no error detection at all (i.e. assuming the bit-flip failure model) applications are highly prone to failure, 3) simple light-weight error detection that enforces a deterministic failure semantic significantly enhances failure tolerance and provides high-quality execution, and 4) when errors are injected at a whole program scale, error tolerance drops but is still high.

We believe exposing hardware errors to applications can trigger ISA specialization similar to short-vector extensions that specialized ISAs for multimedia computation. Following this, a well-defined model of the probabilistic guarantees that hardware provides for correctness can allow development of tailored software with high quality guarantees. Our assisted models are a start in the direction of specifying such a model for hardware uncertainty. These models can trigger detailed exploration at the microarchitecture and device level to exploit opportunities for efficiency.

Finally, the growing synergy between device-level unpredictability and application error tolerance can enable innovations in applications. Exposing hardware errors will help build application-specific error recovery mechanisms that are likely to be even more effective than general-purpose architecture mechanisms.

References

[1] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO '99*, pages 196–207, November 1999.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. L. i. The PARSEC Benchmark Suite: Characterization and Architectural Impli cations. Technical Report TR-811-08, Princeton University, January 2008.

[3] S. Y. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[4] M. A. Breuer and S. K. Gupta. Intelligible Testing. In *Proceedings of 2nd IEEE International Workshop on Microprocessor Test and Verification*, 1999.

[5] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark. Razor: An architecture for dynamic multiresolution ray tracing. *Accepted to ACM Transactions on Graphics. http://www-csl.csres.utexas.edu/gps/publications/razor_tog07/index.html*.

[6] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Intel Technology Magazine*, February 2005.

[7] D. Ernst, N. S. Kim, S. Pant, S. Das, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Micro '03*, pages 7–18, December 2003.

[8] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. Toward a Multicore Architecture for Real-time Ray-tracing. In *Proceedings of the 41st Annual International Symposium on Microarchitecture, MICRO*, November 2008.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04,* Palo Alto, California, Mar 2004.

[10] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 265–276. ACM, 2008.

[11] W. R. Mark and D. Fussell. Real-time rendering systems in 2010. Technical Report TR-05-18, The University of Texas at Austin, Department of Computer Sciences, May.

[12] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[13] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, 2008.

[14] J. Segura and C. Hawkins. *CMOS Electronics: How It Works, How It Fails*. John Wiley & Sons, 2004.

[15] T. Skotnicki, J. A. Hutchby, T.-J. King, H.-S. P. Wong, and F. Boeuf. The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance. *Circuits and Devices Magazine, IEEE*, 21:16–26, Jan-Feb 2005.

[16] A. Szalay and J. Gray. 2020 computing: Science in an exponential world. *Nature*, 440:413–414, March 2006.

[17] C. T. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Reducing the soft-error rate of a high-performance microprocessor. *IEEE Micro*, 24(6):30–37, 2004.

Appendix

In this section, we examine common behavior across applications, followed by a brief summary for each of the applications. Our experiments examine error tolerance for failure rates of 1% for instruction-local analysis and 0.0001% for whole-program analysis.

A. Common Behavior Across Applications

First, all applications behave similarly under the *bit-flip* model's semantics. Also, the light-weight assisted semantics are effective at reducing failures when injecting errors in individual static instructions.

Second, there is little correlation in general between the frequency that instructions execute and both their qualitative impact and their likelihood to cause program failure. This appears counter-intuitive because we would expect instructions that execute more frequently to have more opportunity to degrade program quality or cause outright failure. However, instructions that execute more frequently are also more likely to be data-intensive and not involved in control-intensive program setup or orchestration, improving their error tolerants. Our results show that these two aspects tend to balance each other out for most applications.

Finally, most programs respond better to simultaneous functional and memory instruction injections than simultaneous control injections. When instrumenting all functional and memory instructions, we almost never saw any "emergent failure" arise due to the multiple instruction injections. However, for control instructions, we did see this phenomenon of emergent failure for many applications including x264, bodytrack, and streamcluster.

B. Other Applications

streamcluster and canneal: Both streamcluster and canneal are highly tolerant to errors injected in all instruction types. For canneal, this is because the simulated annealing algorithm is incremental, inherently noisy, and has built-in redundancy. Streamcluster has similar qualities, although it has less inherent noise. Over 95% of streamclusters dynamic instruction executions occur inside of a 12 instruction basic block that performs a squared distance calculation between two values. The incremental contributions of each individual calculation are so minor that occasional errors are extremely tolerable.

Razor: Razor also shows promising results. One large basic block that is particularly tolerant computes distances between multiple rays using differential equations. Errors in these calculations due to failure in one of the instructions in the basic block results in only minor qualitative degradation. We note that Razor exhibits highly non-deterministic and scene-dependent error tolerance; tolerant instructions are intermingled with intolerant ones in a manner that is not statically identifiable.

bodytrack: For bodytrack, our results are generally favorable. Two functions that perform linear algebra computations and process the body geometry data are critical to the correctness of the algorithm, but other regions are almost completely error tolerant due to the inherent noise and redundancy in the tracking algorithm.

swaptions: Swaptions is the least tolerant of all the applications we investigated. Swaptions employs Monte Carlo simulation, and most of the execution time is spent in random number generation. In general, it appears that the quality of the random path is critical to the overall quality of the simulation.