gemV: A Validated Toolset for the Early Exploration of System Reliability

Karthik Tanikella*, Yohan Ko†, Reiley Jeyapaul‡, Kyoungwoo Lee† and Aviral Shrivastava*
*School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Arizona, US
Email: {karthik.tanikella, aviral.shrivastava}@asu.edu
†Department of Computer Science, Yonsei University, Seoul, Korea
Email: {yohan.ko, kyoungwoo.lee}@yonsei.ac.kr
‡ARM Research, Cambridge, UK
Email: reiley.jeyapaul@arm.com

***Abstract*—Decades of technology scaling has brought the threat of soft errors to modern embedded processors. Though several methods have been proposed to protect systems from soft errors, their effectiveness in ensuring error-free computing cannot be guaranteed; without accurate and quantitative estimation of system reliability. The metric *vulnerability* – which defines the likelihood of device failure by accurately evaluating the time it is exposed to soft errors – provides the most effective means to perform early design space explorations to estimate system reliability in the presence of transient soft errors. In this paper, we present gemV – the first accurate and comprehensive vulnerability estimation toolset, which is configurable and extendible to analyse future/novel architecture and microarchitecture designs. Some of the key features of gemV are: (1) all possible microarchitecture components that store bits, even temporarily, are modeled for their vulnerability in the gem5 cycle-accurate simulation platform, (2) its models have been validated (<3% correlation error with 90% statistical confidence) through exhaustive bit-level fault injection experiments, (3) the analytical models have incorporated microarchitecture-level masking effects like speculative executions, flushes, and etc. (4) the modular design of the vulnerability models make it easy to be extended and integrated when novel microarchitecture designs are explored. In addition to microarchitecture-level evaluation of system reliability, gemV provides a means to perform software-level design space explorations – that explore performance-vulnerability trade-offs of algorithm choices, compilers used, compiler optimization levels, etc. A system designer can further use gemV to explore the performance-vulnerability trade-offs of choosing different ISAs.**

I. Introduction

A soft error is a transient bit flip in the processor caused by a number of sources both internal and external to the chip. Among the sources of errors, charge carrying cosmic particles (alpha particles, protons, and even low energy neutrons) have been shown to significantly impact technology nodes sub 45nm [1], [2]. Technology scaling below 10nm dimensions, only exacerbates the vulnerability of devices to such charge carrying particle strikes [3], [4]. As applications of computing systems expands from embedded safety-critical systems [5] to large HPC systems [6], the possible cost of random transient errors involves exorbitant costs in operation and sometimes even loss of life.

Over the years, researchers have proposed many techniques across design layers, to protect modern processors from soft errors [7]. A system architect would always have to make a trade-off analysis between reliability and efficiency; but there does not exist to date a standard methodology for quick and accurate reliability evaluation of system designs. Traditionally, fault injection campaigns have been used to evaluate the effectiveness of protection schemes against soft errors [8]. Faults (bit-flip) are injected into a random bit of the microarchitecture or RTL at a random time, and any deviation from expected behavior is considered a failure. For thorough examination of a processor design exhaustive fault injection experiments on all the bits in the hardware is required, which is not practically possible [9]. Previous works have used statistical methods based on probability theory to estimate failure rate with a reduced number of fault injection runs [10]. However, the accuracy and relevance of such methods, to real-life applications is dependent heavily on the type and randomness of the faults injected. In addition, a fault injection campaign is very difficult to set up correctly and is often flawed [11], [12].

An effective alternate method to estimate failure rate is the metric *vulnerability* – which defines the probability of failures in a system, as the proportion of hardware bits in the processor carrying useful data that will be used by the system [13], [14]. In other words, a datum or bit b in the processor is considered *vulnerable* in time t if an error in this datum will be consumed by the processor in any time $t + n$, with the potential to cause failures in the execution and/or program output. The vulnerability of that datum is then $n\ bit-cycles$. For a system, the sum (in bit-cycles) that bits in the processor are vulnerable during program execution, is the *vulnerability* of the system. Unlike fault injections, vulnerability analysis can be performed in a single simulation run since it is based on microarchitectural working of each component in the processor. This, therefore becomes the only viable method to perform efficient fast and early design space exploration [15], [16] across design layers – hardware specifications, software options, and even architecture choices.

Researchers have presented methods to estimate the vulnerability based on cycle-accurate simulators [14], [17], [18]. However, there is still a need for an validated, comprehensive, and flexible vulnerability modeling: (a) prior vulnerability estimation tools are incomprehensive as they evaluate only a subset of the microarchitectural components, (b) most of the existing tools cannot provide configurable vulnerability modeling (e.g., varying ISA, full-system execution, and multi-core systems) and accurate vulnerability modeling (e.g., register renaming unit pipeline queues) which is due to the underlying simulation platform used, and (c) the accuracy of vulnerability modeling in previous tools are not validated through extensive fault injection campaigns or other schemes.

In this work, we present gemV: a tool for accurate and comprehensive vulnerability estimation based on gem5 [19] – a popular cycle-accurate system-level simulation platform [20]. For instance, gem5 explicitly models most of the microarchitectural components of an out-of-order processor, various ISAs, multicore processors, and system calls. And, extensive fault injection experiments validate gemV to 97% accuracy with 90% confidence. gemV also comprehensively models the vulnerability for all the microarchitectural components of out-of-order processors. gemV presents the most effective toolset for early design space exploration of reliability in the presence of soft error failures. It enables us to answer key design questions such as: (i) Can we decrease the vulnerability by just changing hardware configurations with comparable performance? (ii) Can software engineers improve the hardware-level reliability against soft errors? In a program, the algorithm, the optimization-level of the compiler can also affect the runtime and vulnerability. (iii) System designers can alternate ISAs for better performance, but how can they ensure that protection mechanisms for the previous ISA still works for alternative ISA? The trade-offs between runtime and vulnerability can now be answered rapidly and accurately using the gemV toolset.

In our demonstrations of the capabilities of gemV, we perform a wide range of design space explorations, and observe that vulnerability varies by changing architectural parameters like the number of entries in reorder buffer (ROB), instruction queue (IQ), load store queue (LSQ), and pipeline queues. Among configurations, there is an interesting design configuration with 82% less vulnerability at most 1% performance penalty. A software designer can also use gemV to find the least vulnerable algorithm for a program. For example, we show that switching from a selection sort to a quicksort algorithm can affect the system vulnerability by 91% with the fixed configurations. With the perspective of system designers, it is interesting that the distribution of vulnerabilities among microarchitectural components is sensitive to the ISA. While protecting register rename map and register file will be the most effective in SPARC architecture (more than 75% vulnerability reduction), but the protection will only reduce the vulnerability by 21% in ARM architecture. In contrast, protecting history buffer and IQ will be the most effective in ARM architecture in our study.

II. gemV: Comprehensive and Validated Vulnerability Estimation

Vulnerability has been used as an alternative metric for the failure rate of architectural components against soft errors. A bit b in a microarchitectural component at the specific time t during execution time is vulnerable if a soft error into (b, t) may result in system failure. Otherwise, (b, t) is not vulnerable. Vulnerability is the sum of these vulnerable bits in microarchitectural components of a processor. The unit of vulnerability is bit $\times$ cycle in order to consider both time and space domains. Assume that 2 bits in a microarchitectural component are vulnerable for 5 cycles. The vulnerability of this microarchitectural component is 10 bit $\times$ cycles (= 2 bits $\times$ 5 cycles). In a processor, a bit which may induce failures should be tracked to estimate the vulnerability based on behaviors of microarchitectural components. Note that the vulnerability estimation can be performed in just one simulation run.

In order to estimate the vulnerability, prior works have exploited cycle-accurate and software-based simulators. Mukherjee et al. [14] proposed AVF (Architectural Vulnerability Factor) based on Asim [21] that simulates Itanium2-like IA64 processors. Li et al. [17] proposed SoftArch models the error generation and propagation based on the probabilistic theory in Turandot simulator [22]. Sim-SODA [18] has been proposed to estimate the vulnerability of microarchitectures based on Sim-Alpha simulator [23]. However, previous works are incomprehensive, unavailable for public use, inextensible, and not validated.

First off, gemV is comprehensive as it models the vulnerability of all major hardware structures in an out-of-order processor. Previous vulnerability modeling techniques are incomprehensive since they estimate the vulnerability of just a small subset of the microarchitectural components of the processor. In [14], [17], they do not model the vulnerability estimation for register files, memory hierarchy, and pipeline structures. Sim-SODA considers more components than the other estimation tools, but it still does not model the vulnerability for pipeline queues and renaming units. We model the all the out-of-order components and complete register renaming unit. Comprehensiveness is an important quality to study the breakdown of vulnerabilities of a specific hardware structure as a percentage of the total processor vulnerability. This is useful in studying the effectiveness of new protection mechanisms and also in designing new protection mechanisms to target the hardware structure contributing the highest percentage of the overall system vulnerability. Fig. 1 shows the breakup of processor vulnerability in the default configuration of gem5 ARM out-of-order processor running *stringsearch* benchmark. More than half of the total system vulnerability (54%) that we model has not been modeled in previous works (i.e., pipeline queues and renaming unit). Thus, gemV can provide the entire system-level vulnerability rather than just sum of vulnerabilities of a few microarchitectural components.

Secondly, gemV can provide the accurate and flexible vulnerability modeling due to gem5 simulator. Since vulnerability analysis is based on simulated behaviors of each component, the accuracy and configurability of simulator affects that of vulnerability modeling. Previous tools are inflexible and inaccurate due to the limitations of simulators they

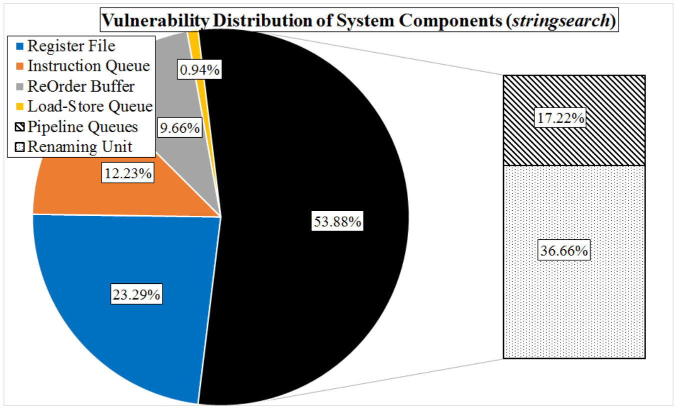

Fig. 1. About 54% of the vulnerability (i.e., vulnerabilities of pipeline queues and register renaming units) has not been considered in previous works

TABLE I
GEMV VALIDATION AGAINST FAULT INJECTION. 300 FAULTS INJECTED PER COMPONENT FOR EACH OF THE FOLLOWING 10 BENCHMARKS: *matrix multiplication*, *hello world*, *stringsearch*, *perlbench*, *gsm*, *qsort*, *jpeg*, *bitcount*, *fft*, AND *basicmath*

Component	**Faults Injected**	**Matched Results**	**Mismatched Results**	**Accuracy (in %)**
Register file	3,000	2,899	101	96.63
Rename map	3,000	2,748	252	91.60
History buffer	3,000	2781	219	92.70
Instruction queue	3,000	2,978	22	99.27
Reorder buffer	3,000	2,760	240	92.00
Load-store queue	3,000	2,979	21	99.30
Fetch queue	3,000	2,890	110	96.33
Decode queue	3,000	2,902	98	96.73
Rename queue	3,000	2,827	173	94.23
I2E queue	3,000	2,959	41	98.63
IEW queue	3,000	2,873	127	95.77
Overall Accuracy				96.78

use. Vulnerability estimation techniques in [14], [17] use the proprietary and private tools which model Intel's Itanium 2-like processor and IBM's Power-PC, respectively. Sim-SODA estimates the vulnerability based on publicly available Sim-Alpha simulator, but it is limited to ALPHA and single-core processors. And, Sim-Alpha has been shown to be up to 43% inaccurate in runtime estimations [24] as compared to real hardwares. Further, Sim-Alpha does not model the floating point pipeline execution accurately.

We use gem5 simulator to implement the vulnerability modeling since gem5 can provide up to 99% accuracy as compared to the real hardware board [20]. Moreover, gem5 simulator is updated actively since it is based on open source infrastructure. gemV is also flexible in its support for multiple ISAs, multi-cores, and system call simulation. Due to this, gemV offers several advantages in vulnerability estimation over previous works. gemV can estimate vulnerability irrespective of the underlying ISA. This can be used in estimating vulnerability of the same program across different ISAs such as X86, ARM, SPARC, and ALPHA as demonstrated in Fig. 4. gemV can estimate the vulnerability of a program running on out-or-order processors in both single core and multi-core configurations.

Lastly, we perform extensive fault injection campaigns in all the microarchitectural components in gem5 as listed in Table I in order to validate our vulnerability estimations in gemV. For each microarchitectural component, we inject a single bit-flip in a microarchitectural bit chosen at random, at randomly selected cycle per each execution of a program in gem5. we inject 300 faults per component for each of ten benchmarks from MiBench [25] and SPEC CPU2006 [26] for the comprehensive simulations. Note that gem5 simulator shares the same information of instructions among ROB, LSQ, and IQ, i.e., a bit flip into one component can affect the behaviors of all these three components. Thus, we modify gem5 by duplicating fields in order to observe single bit flip's impact on a specific component exclusively.

In our fault injection campaigns, we run 300 simulations per component. Theoretically, 300 simulations are large enough to observe the statistical fault injection based experiments regardless of an initial population size with the 90% confidence level [27]. Experimentally, we also validate that 300 runs can provide the stable results for all the components as compared to validation results with more than 300 runs. We have injected 1 through 2,000 single-bit flips randomly by incrementing 1 for each microarchitectural component into a benchmark. If the number of runs is smaller than 300, the accuracy is unstable. However, if the number of runs is equal to or larger than 300, the accuracy is stable. Indeed, the difference is less than 2% among all the runs over 300 in our simulations. Thus, 300 runs per microarchitectural component should be large enough to validate our gemV with fault-injection campaigns.

Table I lists the results of our fault injection experiments for each microarchitectural component. The results show that component vulnerability estimated using gemV is about 97% accurate. Benchmarks are chosen to minimize the effects of software-level masking effects on error propagation. The result of a validation run is declared as a match if the result of the fault injection agrees with the prediction made by gemV. For example, if gemV predicts that a bit is vulnerable, then the corresponding fault injection run should result in an incorrect output or program failure. As shown in Table I, we observe 2,899 matched ones and 101 mismatched results for the register file, giving us an accuracy of 96.63%.

III. GEMV: TOOL FOR FAST AND EARLY DSE

The value of gemV is in making possible fast and early DSE or Design Space Exploration. Radiation testing requires developers to build a fully working prototype before evaluating the reliability, and even register-transfer level fault injection requires developers to bring down the design to synthesizable form before reliability can be quantified. As opposed to these, gemV allows hardware architects, software engineers, and system designers to evaluate the reliability at a very early high-level design stage.

A. *gemV for Hardware Implementation*

Extending hardware configuration to a larger design space, one interesting question is, that given an existing processor configuration, and performance leeway, how can we change

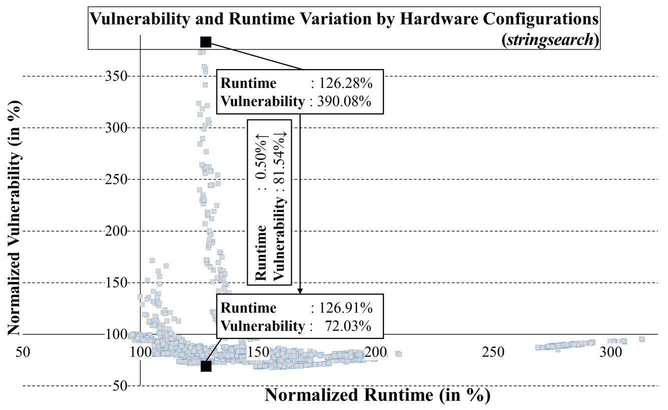

Fig. 2. Different hardware configurations generates interesting design space in terms of runtime and vulnerability. Vulnerability can be reduced by up to 81% with less than 1% runtime overhead by varying hardware configurations.

some hardware configurations to minimize the vulnerability. This can be answered with gemV by plotting design points for runtime against vulnerability. In this experiment, we vary the total number of entries in ROB, LSQ, IQ, and pipeline queues to plot a design space for a benchmark *stringsearch* in MiBench suites [25] as shown in Fig. 2. We establish a baseline runtime and vulnerability with sizes of 192, 64, and 8 entries for ROB, LSQ, and IQ respectively. A hardware designer can use this design space to choose the required hardware configuration as dictated by runtime and vulnerability bounds. Given a certain runtime target, the hardware designer can now find several design points for vulnerability as shown by the grey band in Fig. 2. In this example, for a runtime overhead of 1%, it is possible to find a design point with 81% less vulnerability. Given any runtime or vulnerability overhead it is now possible to find alternate design points with lower vulnerability or runtime with gemV.

B. *gemV for Software Development*

gemV can also be used by the software engineer to find alternate design points with lower vulnerability or runtime. Alternate design points can be realized with software changes in either the algorithm, the compiler used or the level of optimization. For example, given the choice of two sorting algorithms - such as quicksort and insertion sort - which would be the optimal choice for the best trade-off between runtime and vulnerability? gemV can be used to study the design space for runtime and vulnerability due to changes in software. To study such changes, we perform an experiment by establishing a baseline runtime and vulnerability for an insertion sort algorithm compiled with *gcc* at the highest (O3) level of optimization. Fig. 3 presents the normalized runtime and vulnerability for various combinations of algorithms, compilers and optimization levels. We consider an array sorting application with five sorting algorithms (bubble, quick, insertion, selection, and heap sorting), two compilers (GCC and LLVM [28]), and four optimization levels (no optimization, O1, O2, and O3). We note that vulnerability can be reduced by up to 91% without additional runtime overhead with software changes. The software engineer can use this design space to choose optimal design points to meet runtime and vulnerability requirements. In this example, switching from a selection sort algorithm at O1 level of optimization to quicksort at O3 level of optimization can reduce runtime by 53% and vulnerability by 91%.

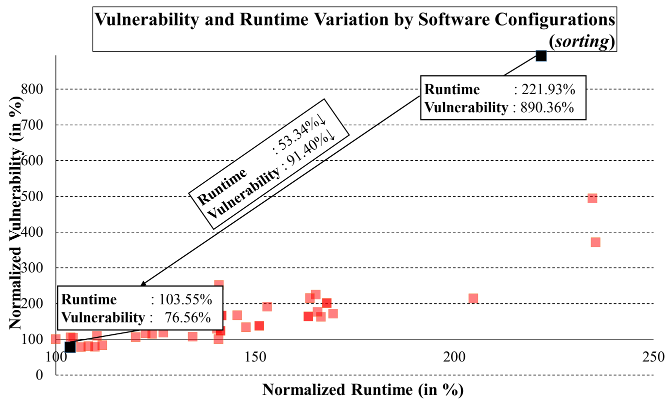

Fig. 3. Different software configurations can generate interesting design space in terms of vulnerability on the same hardware. Vulnerability can be reduced by 91% without runtime overhead by software changes.

C. *gemV for System Design*

A system designer can also use gemV to make design choices in several interesting ways. In this experiment, we will demonstrate two such examples. (i) Given a choice of processors running different ISAs, which one offers the best trade-off in runtime or vulnerability? We have run this experiment by changing the ISA within gemV while keeping all hardware sizes constant. Fig. 4 shows vulnerability and runtime under different ISAs such as ARM, SPARC, x86, and ALPHA for the *stringsearch* benchmark, with no change in hardware and software configurations. Baseline vulnerability and runtime are established on the ARM ISA. A benchmark, *stringsearch*, running on an ALPHA is 38% less vulnerable than that on SPARC. The system designer can choose the ARM ISA for minimum runtime or the ALPHA for minimum vulnerability.

(ii) The system designer can study the breakdown of vulnerability to individual hardware components. This can be used to design protection techniques targeting specific components. Fig. 4 shows the detailed breakdown of each component such as HB (history buffer), RM (rename map), LSQ, IQ, IEWQ (IEW queue), I2EQ, RQ (rename queue), DQ (decode queue), FQ (fetch queue), RF (register file), and ROB. History buffer and IQ take up the highest fraction (50%) of the vulnerability in an ARM processor while the Rename Map and Register File contribute the most in case of SPARC and ALPHA respectively. In this example, a protection mechanism such as ECC can be applied to the register file on the SPARC processor. However, the same protection is not very useful on the ARM processor as the RF contributes only 21% to the system vulnerability.

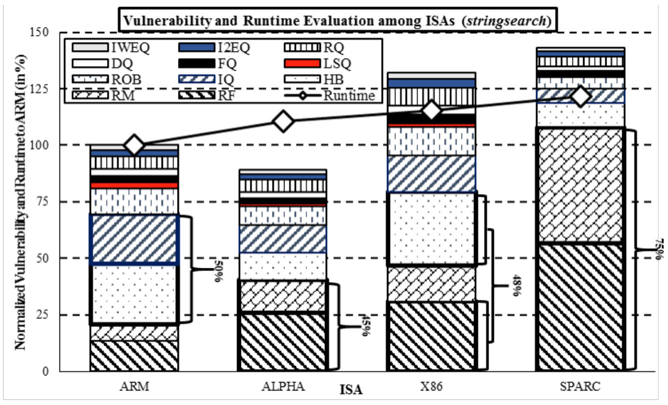

Fig. 4. Variation in runtime and vulnerability for *stringsearch* under different ISAs. Bars show vulnerability and diamond points indicate runtime

IV. Conclusion

Several protection techniques against soft errors have been proposed ever since reliability became an important design concern. The need to quantitatively study the effectiveness of such protection techniques have led to several vulnerability estimation tools be proposed. However, previous vulnerability estimation tools are incomplete, inaccurate, and inflexible. In this paper, we presented gemV, a comprehensive and accurate vulnerability estimation based on the cycle-accurate simulator gem5. We also showed that our tool has been validated against fault injection experiments. To demonstrate the value in gemV as a design space exploration tool, we performed several experiments useful to hardware and software engineers. For the hardware designer, we showed the effects of microarchitectural changes on runtime and vulnerability. For the software designer, we showed the effects of the algorithm, compiler and optimization level on runtime and vulnerability. We also demonstrated the usefulness of gemV to a system designer in designing component specific or ISA specific soft-error protection techniques. In the future, gemV will also model the effects of software level masking. This will improve the accuracy and comprehensiveness of our tool even further.

Acknowledgment

This research was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and future Planning (NRF-2015R1A2A1A15053435), by Next-Generation Information Computing Development Program through the NRF funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065522), and by funding from National Science Foundation grants CCF 1055094 (CAREER), and CNS 1525855.

References

[1] R. Baumann, “Soft errors in advanced computer systems,” *Design Test of Computers, IEEE*, vol. 22, no. 3, 2005.

[2] A. Dixit and A. Wood, “The impact of new technology on soft error rates,” in *Reliability Physics Symposium (IRPS), 2011 IEEE International*, 2011, pp. 5B.4.1–5B.4.7.

[3] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda, and M. A. Aguirre, “Compiler-directed soft error mitigation for embedded systems,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 2, pp. 159–172, 2012.

[4] N. Mahatme, N. Gaspard, S. Jagannathan, T. Loveless, B. Bhuva, W. Robinson, L. Massengill, S.-J. Wen, and R. Wong, “Impact of supply voltage and frequency on the soft error rate of logic circuits,” *Nuclear Science, IEEE Transactions on*, vol. 60, no. 6, pp. 4200–4206, 2013.

[5] J. Yoshida, “Toyota case: Single bit flip that killed,” *EE Times*, vol. 8, 2013.

[6] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, “Understanding the propagation of transient errors in HPC applications,” in *SC*. ACM, 2015, pp. 72:1–72:12.

[7] I. Lee, M. Basoglu, M. Sullivan, D. H. Yoon, L. Kaplan, and M. Erez, “Survey of error and fault detection mechanisms,” *UT Austin, Tech. Rep*, 2011.

[8] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, “Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection,” *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 313–322, 2012.

[9] H. T. Nguyen and Y. Yagil, “A systematic approach to SER estimation and solutions,” in *IRPS*. IEEE, 2003.

[10] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, “Design and evaluation of system-level checks for on-line control flow error detection,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 6, pp. 627–641, 1999.

[11] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, “Quantitative analysis of control flow checking mechanisms for soft errors,” in *DAC*, 2014.

[12] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design,” in *DAC*, 2013, pp. 1–10.

[13] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *HPCA*. IEEE, 2005.

[14] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Micro*. IEEE/ACM, 2003.

[15] A. Biswas, P. Racunas, J. Emer, and S. S. Mukherjee, “Computing accurate AVFs using ACE analysis on performance models: A rebuttal,” *Computer Architecture Letters*, vol. 7, no. 1, pp. 21–24, 2008.

[16] R. Jeyapaul and A. Shrivastava, “Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors,” in *CASES*, 2011.

[17] X. Li, S. Adve, P. Bose, and J. Rivers, “SoftArch: An architecture-level tool for modeling and analyzing soft errors,” in *DSN*, 2005.

[18] X. Fu, T. Li, and J. Fortes, “Sim-SODA: A unified framework for architectural level software reliability analysis,” in *MoBS*, 2006.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[20] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, “Accuracy evaluation of gem5 simulator system,” in *ReCoSoC*. IEEE, 2012, pp. 1–7.

[21] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert *et al.*, “Asim: A performance model framework,” *Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[22] M. Moudgill, P. Bose, and J. H. Moreno, “Validation of Turandot, a fast processor model for microarchitecture exploration,” in *IPCCC*. IEEE, 1999.

[23] R. Desikan, D. Burger, S. W. Keckler, and T. Austin, “Sim-Alpha: a validated, execution-driven Alpha 21264 simulator,” *UT Austin, Tech. Rep.*, 2001.

[24] R. Desikan, D. Burger, and S. W. Keckler, “Measuring experimental error in microprocessor simulation,” in *ISCA*. ACM, 2001.

[25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in *WWC*, 2001.

[26] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[27] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, “Statistical fault injection: Quantified error and confidence,” in *DATE*, 2009.

[28] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*. IEEE, 2004, pp. 75–86.