Open access • Proceedings Article • DOI:10.1145/1366224.1366225

Quantifying software vulnerability — Source link

Vilas Sridharan, David Kaeli

Institutions: Northeastern University

Published on: 05 May 2008

Topics: Vulnerability (computing) and Software

Quantifying Software Vulnerability

Vilas Sridharan
Department of Electrical and Computer Engineering
Northeastern University
360 Huntington Ave.
Boston, MA 02115
vilas@ece.neu.edu

David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
360 Huntington Ave.
Boston, MA 02115
kaeli@ece.neu.edu

ABSTRACT

The technique known as ACE Analysis allows researchers to quantify a hardware structure's Architectural Vulnerability Factor (AVF) using simulation. This allows researchers to understand a hardware structure's vulnerability to soft errors and consider design tradeoffs when running specific workloads. AVF is only applicable to hardware, however, and no corresponding concept has yet been introduced for software. Quantifying vulnerability to hardware faults at a software, or program, level would allow researchers to gain a better understanding of the reliability of a program as run on a particular architecture (e.g., X86, PowerPC), independent of the micro-architecture on which it is executed. This ability can provide a basis for future research into reliability techniques at a software level.

In this work, we adapt the techniques of ACE Analysis to develop a new software-level vulnerability metric called the *Program Vulnerability Factor* (PVF). This metric allows insight into the vulnerability of a software resource to hardware faults in a micro-architecture independent way, and can be used to make judgments about the relative reliability of different programs. We describe in detail how to calculate the PVF of a software resource, and show that the PVF of the architectural register file closely correlates with the AVF of the underlying physical register file and can serve as a good predictor of relative AVF when comparing the AVF of two different programs.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: [Reliability, Testing, and Fault-Tolerance

General Terms

Reliability

WREFT'08, May 5, 2008, Ischia, Italy.

Keywords

Fault Tolerance, Soft Errors, Modeling

1. INTRODUCTION

Reliability is now a first-class design constraint for most systems from high-end mainframes to commodity PCs, primarily due to the effects of soft errors from cosmic particles. The rate of soft errors in a system can be higher than that of all other errors combined, making soft errors one of the primary concerns for system reliability [1]. Microprocessor vendors typically set a soft-error rate (SER) target for each design and perform significant pre-silicon analysis to ensure a design adheres to this target. Therefore, accurately estimating the SER of a particular design early in the design cycle is crucial to measuring a design's performance against its SER target.

This type of early-design SER analysis has been greatly aided by the concept of *Architectural Vulnerability Factor* (AVF) and the introduction of *ACE Analysis* as a method to estimate a structure's AVF [6]. The AVF of a processor structure is defined as the probability that a fault in that structure will result in a visible error in the final output of a program. This is a well-defined, measurable quantity that both yields insight into the behavior of a structure and allows a simple calculation to determine its failure rate. ACE Analysis estimates a structure's AVF by determining, during each cycle, whether a fault in a bit will propagate to the program's output. This allows a designer to estimate a structure's AVF in a single, fault-free, simulation run, significantly faster than prior techniques such as software fault-injection which require hundreds of runs to achieve statistical significance [4]. As a consequence, ACE Analysis has enabled much research into the behavior of processor structures and reliability improvement techniques [9] [8] [3].

Although AVF analysis can yield an understanding of the reliability behavior of a hardware structure, no corresponding method has yet been developed to quantify the vulnerability of a *program.* to hardware faults. Such a method would allow researchers to better understand the link between program code and reliability, and could enable the development of reliability techniques at a compiler or even programming language level. For example, the following questions are currently not answerable independent of hardware: "Is *mcf* or *equake* more susceptible to faults?", or "What is the reliability impact of this compiler optimization?". A metric to quantify software vulnerability would yield answers to those questions about the *relative* vulner-

ability of two programs independent of a particular micro-architecture. Furthermore, a well-defined metric might also allow prediction of the *absolute* vulnerability of a single program when run on a system. Our work examines both of these questions and lays the foundation for such a software vulnerability metric.

The rest of this paper is organized as follows. Section 2 sets out the characteristics that a software vulnerability metric should possess. Section 3 reviews the application of ACE Analysis to measure the AVF of hardware structures. Section 4 then applies this methodology to software, introducing a metric we call the *Program Vulnerability Factor*. Section 5 discusses the relationship between PVF and AVF, and Section 6 presents a preliminary evaluation of PVF.

2. PROPERTIES OF A SOFTWARE VULNERABILITY METRIC

In this section, we discuss several attributes that a software vulnerability metric should possess. These attributes allow a metric to be both general-purpose in its ability to provide insight, but also of practical use in drawing conclusions about program reliability. First, in order to be general-purpose, any vulnerability metric must be *micro-architecture independent*. Therefore, *a software vulnerability metric must be a function of only architectural (software-visible) parameters*, since these are the only features guaranteed not to change across different implementations of a given architecture. This requirement precludes a crucial feature of ACE Analysis: the use of clock cycles as a measure of time. We discuss a solution to this issue in Section 4. This requirement also implies that the vulnerability metric can be modeled in such a way that its calculation does *not* require detailed micro-architectural simulation of the program; vulnerability estimates should be derivable through program analysis techniques such as offline examination of an instruction trace or architectural simulation of the program.

Second, a software vulnerability metric should be able to predict the failure rate of a system upon which the program is executing. For instance, if system X consists of program S executing on hardware H, the failure rate of the system (FIT_X) should be expressable as a function of the vulnerability factor of program S (VF_S):

$$FIT_X = f_X(VF_S) \tag{1}$$

The function f_X applies VF_S to a set of system-specific parameters (e.g., the raw failure rate per bit) to compute an overall failure rate. Similarly, the failure rate of a system is also expressable as a function of the hardware's AVF:

$$FIT_X = g_X(AVF_H) \tag{2}$$

This implies that a software vulnerability factor will have a well-defined relationship with AVF:

$$AVF_H = g_X^{-1}(f_X(VF_S)) \tag{3}$$

Therefore, knowledge of both VF_S and certain parameters of system X can allow us to predict the AVF of hardware H when running this program. We discuss this relationship in Section 5.

This also implies that knowledge of the vulnerability of two programs A and B (for example, $VF_A > VF_B$), will enable a meaningful statement about the failure rate of program A relative to the failure rate of program B *when run on the same hardware*. This is an important consideration when discussing the utility of a vulnerability metric.

3. BACKGROUND ON AVF AND ACE ANALYSIS

To develop a software vulnerability metric, we first review the concept of AVF. The Architectural Vulnerability Factor of a processor structure is defined as the probability that a fault in that structure will result in a visible error in the final output of a program [6]. A bit in which a fault will result in incorrect execution is said to be necessary for *architecturally correct execution*; these bits are termed ACE bits. All other bits are un-ACE bits. An individual bit may may be ACE for a fraction of the overall execution cycles and un-ACE for the rest. Therefore, the AVF of a single bit can be defined as the fraction of cycles that the bit is ACE.

We refer to a cycle in which a bit is ACE as an *ACE bit-cycle*. One can then define the AVF of a hardware structure as the fraction of bit-cycles in the structure that are ACE. For hardware structure H with size B_H (in bits), its AVF over a period of N cycles can be expressed as follows [6]:

$$\begin{aligned} AVF_H &= \frac{\sum_N ACE\ bits\ in\ H}{B_H \times N} \\ AVF_H &= \frac{ACE\ bit\text{–}cycles\ in\ H}{B_H \times N} \end{aligned} \tag{4}$$

The average AVF of an entire *processor* can be computed as the weighted average of the AVFs of each structure for systems of reasonable size [5].

4. PROGRAM VULNERABILITY FACTOR

Using the methodology presented in the previous section, we now develop a definition for a program's vulnerability. We term our vulnerability metric the *Program Vulnerability Factor*, or PVF. First, however, we must define the quantities that we can use in computing PVF; as stated earlier, these are limited to architecturally-visible quantities to preserve micro-architecture independence.

An AVF value can be computed precisely for each micro-architectural bit, or *m-bit*, in a hardware structure. These values are then summed across all bits in the structure to yield the AVF of that structure, and across all structures in a processor to yield an average AVF for that processor. In this way, AVF calculations treat a processor as a collection of hardware structures. Similarly, a *program* can be viewed as a collection of *software resources*. We define a software resource as an independently-addressable architectural structure. For example, an architectural register (or a byte within the register, if each byte is addressable) is a software resource; the architectural register file is a collection of individual software resources. Each software resource has an architecturally-defined size in bits; to avoid confusion, we refer to these as architectural bits, or *a-bits*. Therefore, we can precisely compute a vulnerability value for each a-bit within a software resource, and these values can be summed across all a-bits in the resource to yield the vulnerability of the entire resource. As with hardware structures, the value of a software resource (or of a subset of a-bits within the resource) may or may not be required for architecturally

correct execution at a given point in time. We refer to a-bits that are needed for correct operation as *ACE a-bits*; other bits are *un-ACE a-bits*.

To complete the definition of PVF, we require an architectural definition of *time*. AVF typically measures time quanta in clock cycles; events that occur within a single clock cycle are usually not separable. Clock cycles are not an architectural concept, however, and cannot be used when computing PVF. PVF calculations require some other measurement of the relative ordering and temporal distance between events. In a sequential programming model, an architecturally visible quantity that allows these measurements is the instruction flow. Instructions are ordered with respect to each other, and the distance between operations can also be given as the number of intervening instructions. Therefore, our definition of PVF uses a (dynamic) instruction as a single time quantum. This choice will vary by architecture. For example, some architectures such as Itanium execute several instructions (instruction *bundles*) at the same "time". For these architectures, the time quantum used for PVF calculations should be an instruction bundle. In addition, certain operations within an instruction (or bundle) are also often ordered with respect to one another; for instance, one instruction can read and write the same register. Although these events occur at the same "time", the event ordering defined by the architecture must be preserved.

We now have all the concepts needed to precisely calculate a Program Vulnerability Factor. The PVF of an a-bit is the fraction of time (in instructions) that the bit is ACE; we refer to an instruction during which an a-bit is ACE as an *ACE bit-instruction*. The PVF of an entire software resource is then the fraction of bit-instructions in the resource that are ACE. For a particular software resource R with size B_R, its PVF over I instructions can then be expressed as follows:

$$PVF_R = \frac{\sum_I ACE\ a\text{–}bits\ in\ R}{B_R \times I}$$
$$PVF_R = \frac{ACE\ bit\text{–}instructions\ in\ R}{B_R \times I} \quad (5)$$

The average PVF of an entire program can then be computed as the weighted average of the PVFs of each software resource within the program.

4.1 Example Calculation

To demonstrate a practical example of calculating PVF using Equation 5, we calculate the PVF of register r1 in the following assembly code:

```
1:      ld r1 = [r2]
2:      ld r3 = [r2]
3:      ld r4 = [r2]
loop:
4:      add r3 = r3, r4
5:      sub r1 = r1, 1
6:      br loop, r1 > 0
7:      st [r2] = r3
8:      ret
```

This assembly code initializes the values of registers r1, r3, and r4, executes a loop with exit condition $r1 \leq 0$, saves the final value of r3, and returns. There are eight static instructions in this code; the number of dynamic instructions is determined by the initial value of r1. To calculate the PVF of r1, we must identify the operations that occur to register r1 and the dynamic instruction at which these operations occur. At the beginning of code execution, the load instruction on line 1 causes a write to register r1. During every iteration of the loop, the subtract on line 5 causes both a read from r1 and a write to r1. Although they occur at the same dynamic instruction, the semantics of the architecture define the read as occurring "before" the write, since the read result is the value of r1 prior to the write. Lastly, the branch on line 6 causes a final read to r1.

If the initial value of r1 is 1, then the loop executes exactly once. In this case, r1 is written at instruction 1, read at instruction 5, written at instruction 5, and read at instruction 6. Therefore, r1 is vulnerable between instructions 1 and 5 and between instructions 5 and 6. Since a total of 8 dynamic instructions are executed, the PVF of r1 for this section of code can be calculated as:

$$PVF_{r1} = \frac{(5-1)+(6-5)}{8}$$
$$PVF_{r1} = 62.5\%$$

If we assume the loop executes 100 times, r1 is continually written and read by the subtract on line 5 and read by the branch on line 6. Each instance of the branch occurs 1 dynamic instruction after the subtract, and the subsequent subtraction occurs 2 dynamic instructions after the branch. Therefore, the PVF calculation for r1 becomes:

$$PVF_{r1} = \frac{(5-1)+(6-5)+(99*1)+(99*2)}{305}$$
$$PVF_{r1} = \frac{302}{305}$$
$$PVF_{r1} = 99\%$$

Calculating the PVF for a set of software resources over this section of code requires computing a weighted average of the PVF of each software resource. For example, if this architecture has eight available logical registers of 32 bits each, the PVF of the architectural register file can be given as:

$$PVF_{ARF} = \frac{\sum_{i=1}^{8} 32 \times PVF_{r_i}}{8 \times 32}$$
$$PVF_{ARF} = \frac{ACE\ bit\text{–}instructions\ in\ ARF}{8 \times 32 \times I}$$

In this way, an approximation of the PVF of the entire program can be constructed from its constituent parts.

For clarity, there are many effects that we do not include in the previous discussion, such as masking of values (when only a subset of bits in the software resource affect the outcome) and the effect of dynamically-dead instructions (instructions whose results are unused). These effects can easily be incorporated in a detailed PVF calculation.

5. PREDICTING AVF FROM PVF

With our PVF model, it is possible to derive an equation to compute a hardware structure's AVF given the PVF behavior of the software resources that use the hardware structure and knowledge of the micro-architectural parameters of the structure. This requires *unique attribution* of

every ACE bit-cycle in a hardware structure to an ACE bit-instruction in a software resource. Every ACE bit-cycle in a hardware structure must be attributable to at most one ACE bit-instruction. The converse is not true: an ACE bit-instruction may have many ACE bit-cycles attributed to it.

To do this attribution, we need to determine, for each ACE bit-instruction: (1) the number of m-bits per a-bit; and (2) the number of cycles per instruction. The number of m-bits per a-bit for bit-instruction i can be expressed as m_i. The number of cycles per instruction for each bit-instruction i can be expressed as n_i. For a set of instructions I, these values can be expressed as M_I and N_I, respectively. Given these values, we can then construct an equation to calculate AVF_H:

$$AVF_H = \frac{\sum ACE\ a\text{–}bits\ in\ R \times m_i \times n_i}{B_R \times M_H \times I \times N_I} \quad (6)$$

For many structures, $m_i = 1$. For example, architectural register values for a given instruction are typically present in only one location in a physical register file. M_H is typically also constant for a given structure. For instance, a register file that uses 128 physical 64-bit registers to implement 32 architectural 64-bit registers will have $M_H = 128/32 = 4$.

N_I and n_i both reflect a conversion from instructions to cycles, and can be viewed as the throughput of this particular program. N_I is the average CPI of the program, while n_i is the instantaneous CPI of a given instruction. While there is significant variation in n_i between instructions, we leave this area of exploration to future work and assume that $n_i \approx N_I$ for every bit-instruction. Therefore, Equation 6 can be simplified to:

$$AVF_H = \frac{ACE\ bit\text{–}instructions\ in\ R}{B_R \times M_H \times I}$$
$$AVF_H = \frac{PVF_R}{M_H} \quad (7)$$

Equation 7 can be used to predict a hardware structure's AVF when running a program with a known PVF.

Our other goal was to compare the relative vulnerability of two programs. We can do this by using Equation 7 to predict the relative AVF of programs A and B as follows:

$$\frac{AVF_A}{AVF_B} = \frac{PVF_A}{PVF_B} \quad (8)$$

6. PRELIMINARY EVALUATION

For our evaluation, we used the M5 simulator [2] with the parameters shown in Table 1. We ran the SPEC CPU2000 benchmarks using the single early simulation points given by Simpoint analysis [7]. Our goals are to determine whether PVF values are good predictors: first, of the relative vulnerability of two programs; and second, of the absolute vulnerability of a single program.

To determine whether PVF is a good predictor of relative vulnerability, we measured the PVF of each program's Architectural Integer Register File (ARF) and compare these values to the AVF of our processor's implementation of this resource, the Integer Physical Register File (PRF). For PVF to serve as a good predictor, we must be able to determine two programs' relative AVF values based solely on their PVF. Figure 1 shows the ARF PVF and the PRF AVF for each benchmark. The AVF of the physical register file tracks closely with the PVF of the architectural register file: the PVF and AVF values have a correlation coefficient of 0.98. This high correlation coefficient implies that our model is a good predictor of relative AVF values from PVF values. For example, $PVF_{applu} = 86.2\%$ and $PVF_{apsi} = 60.42\%$. From this, we would expect AVF_{applu} to be 143% of AVF_{apsi}, which closely matches the actual value of 139%. There are a few benchmark pairs for which this prediction would fail: for example, since $PVF_{apsi} = 60.4\%$ and $PVF_{equake} = 48.9\%$, one would predict that AVF_{equake} would be roughly 80% of AVF_{apsi}. However, AVF_{equake} is actually slightly larger than AVF_{apsi}. This is due to the divergent performance of the two benchmarks: *equake* has a CPI of 4, while *apsi* has a CPI of less than 1. Since m_i is constant for the register file (each architectural register has only one *currently* live counterpart in the physical register file), this implies that *equake* has larger n_i values on average, leading to a higher AVF. A detailed analysis of the difference would enable an even better correlation between PVF and AVF values; we reserve this for future work.

Parameter	Value
Issue Width	8 instructions
Commit Width	8 instructions
Physical Registers	256 integer / 256 Floating Point
Issue Queue	64 entries
Re-Order Buffer	192 entries
Load-Store Queue	32 loads / 32 stores

Table 1: Simulated Machine Parameters

We would also like to determine whether PVF is a good predictor of absolute AVF for a single program. Figure 2 compares the actual AVF for the PRF and the absolute AVF we would predict using the approximation in Equation 7. As can be seen, Equation 7 generates a consistent under-prediction of the actual AVF values. This is due to our assumption that $n_i \approx N_I$ for all ACE bit-instructions (i.e., that every instruction has the same instantaneous CPI). In reality, instantaneous CPI values vary based on a number of factors, such as the type of instruction and the program's control and data dependences. The under-prediction is relatively consistent, however, leading us to believe that a better estimate of the CPI parameters might allow us to refine this prediction further.

7. CONCLUSION AND FUTURE WORK

This work has developed a starting point to quantify software vulnerability. We have introduced the *Program Vulnerability Factor* as a method to yield insight into a program's micro-architecture-independent reliability behavior. A metric such as PVF can open many avenues of future research: for example, hardware techniques to exploit trends in software vulnerability or compiler techniques to reduce program vulnerability.

Overall, our data show that PVF may serve as a good predictor of the relative AVF of two programs, but more work is needed to refine PVF as a predictor for absolute AVF. However, we must confirm that PVF is general-purpose enough to serve as a good predictor across micro-architectural variation and for other types of software resources (such as func-

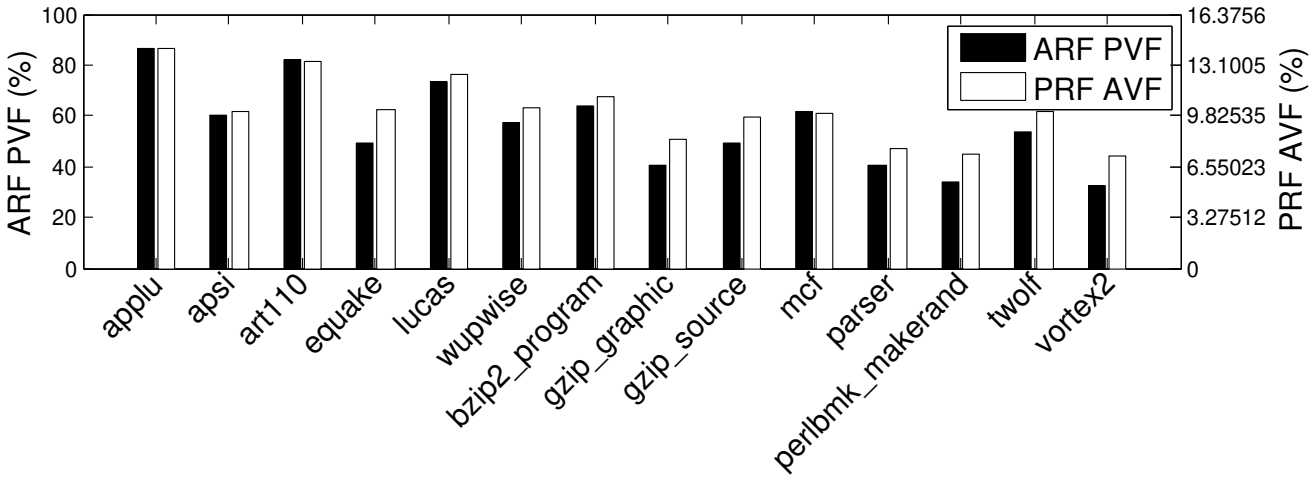

Figure 1: PVF of the Architectural Integer Register File and AVF of the Physical Integer Register File

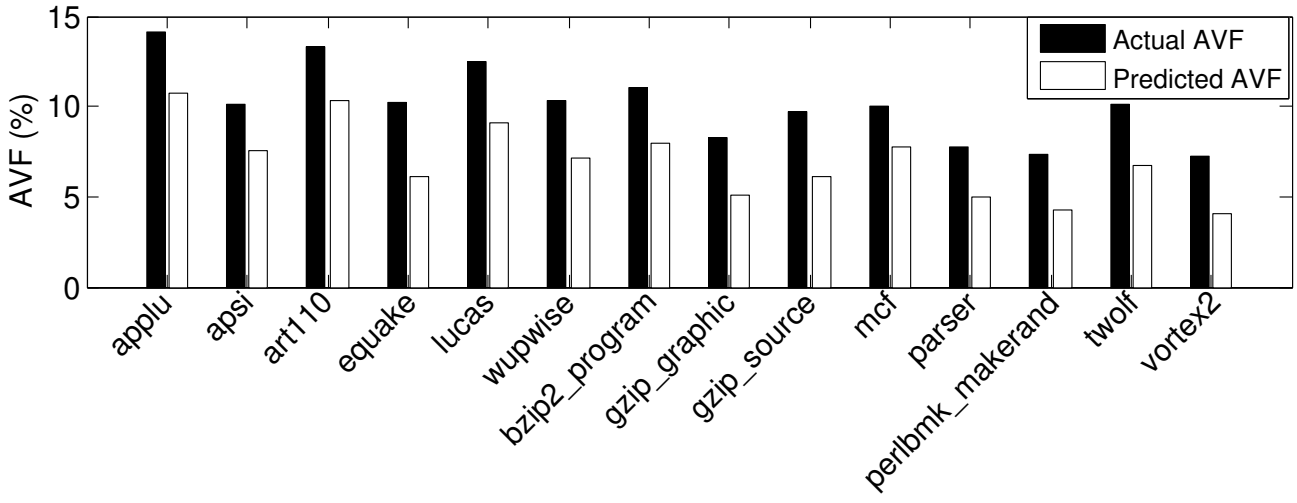

Figure 2: Actual and Predicted PRF AVF

tional units and memory) that we have not yet considered. Despite these limitations, however, we believe this work to be a strong starting point on the way to quantifying software vulnerability.

8. REFERENCES

[1] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, Sept. 2005.

[2] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July-Aug. 2006.

[3] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 532–543, Washington, DC, USA, 2005. IEEE Computer Society.

[4] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 416–428, Washington, DC, USA, 2002. IEEE Computer Society.

[5] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Architecture-level soft error analysis: Examining the limits of common assumptions. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 266–275, Washington, DC, USA, 2007. IEEE Computer Society.

[6] 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 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 29, Washington, DC, USA, 2003. IEEE Computer Society.

[7] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244, Washington, DC, USA, 2003. IEEE Computer Society.

[8] V. Sridharan, D. R. Kaeli, and A. Biswas. Reliability in the shadow of long-stall instructions. In *SELSE '07: The Third Workshop on System Effects of Logic Soft Errors*, Austin, TX, April 2007.

[9] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 264, Washington, DC, USA, 2004. IEEE Computer Society.