Eliminating Microarchitectural Dependency from Architectural Vulnerability

Vilas Sridharan and David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
{vilas, kaeli}@ece.neu.edu

Abstract

The Architectural Vulnerability Factor (AVF) of a hardware structure is the probability that a fault in the structure will affect the output of a program. AVF captures both microarchitectural and architectural fault masking effects; therefore, AVF measurements cannot generate insight into the vulnerability of software independent of hardware. To evaluate the behavior of software in the presence of hardware faults, we must isolate the software-dependent (architecture-level masking) portion of AVF from the hardware-dependent (microarchitecture-level masking) portion, providing a quantitative basis to make reliability decisions about software independent of hardware.

In this work, we demonstrate that the new Program Vulnerability Factor (PVF) metric provides such a basis: PVF captures the architecture-level fault masking inherent in a program, allowing software designers to make quantitative statements about a program's tolerance to soft errors. PVF can also explain the AVF behavior of a program when executed on hardware; PVF captures the workload-driven changes in AVF for all structures. Finally, we demonstrate two practical uses for PVF: choosing algorithms and compiler optimizations to reduce a program's failure rate.

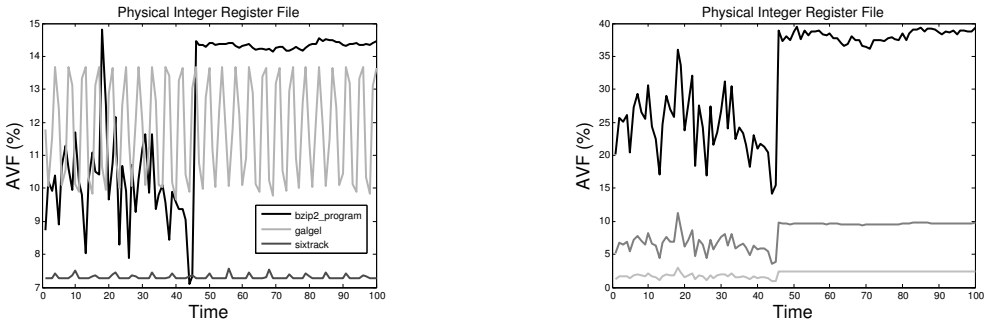
1. Introduction

Reliability is now a first-class design constraint for systems from high-end mainframes to commodity PCs, primarily due to the effects of transient faults caused by particle strikes [1]. To meet reliability goals, microprocessor vendors typically set a failure rate (FIT) target for each design and perform significant pre-silicon analysis to ensure a design adheres to this target [2]. Therefore, accurately modeling a system's failure rate is crucial to understanding its performance relative to the reliability goals.

One facet of fault modeling is determining the amount of fault masking in a system: a *masked fault* does not affect correct system operation and thus does not impact a system's failure rate. Faults can be masked at many levels; properly assessing the impact of each masking level is important to achieve accurate failure rate estimates. *Device-level masking* occurs when a fault does not propagate to the inputs of a device within the device's window of timing vulnerability; this can be quantified using Seifert et al.'s *Timing Vulnerability Factor* [3]. *Microarchitecture-level masking* occurs when a microarchitecturally-visible fault (i.e., a fault not masked at a device level) does not propagate to architectural state. *Architecture-level masking* occurs when an architecturally-visible fault does not affect correct program output. Mukherjee et al.'s *Architectural Vulnerability Factor* quantifies both microarchitecture-level and architecture-level fault masking [2] [4].

As shown in Figure 1, this means that AVF is dependent on software (architecture) as well as hardware (microarchitecture). As a result, software designers cannot use AVF to make hardware-independent statements about the reliability of a program (i.e., the level of fault masking in a program). An accurate method to quantify architecture-level fault masking would yield a better understanding of the link between software and reliability and allow the development of hardware-independent reliability techniques in (for example) a compiler or programming language. Reducing the vulnerability of software should directly translate into improved availability and/or reliability when the program is executed on hardware.

In this work, we demonstrate that a new microarchitecture-independent metric, the *Program Vulnerability Factor* (PVF) [5], quantifies the architecture-level fault masking inherent in a program. PVF can be calculated for any architecturally-visible resource and is a property of a dynamic execution of a program. Therefore, PVF is impacted only by changes to the binary or to input data and not by changes in hardware: it can be used to reason about the vulnerability of a program without any knowledge of the target microarchitecture. We show that PVF



(a) Constant Microarchitecture / Variable Workload

(b) Variable Microarchitecture / Constant Workload

Figure 1. The AVF of a Physical Register File when running three programs on one microarchitecture and one program (*bzip2*) on three microarchitectures. The AVF depends on both workload and hardware configuration.

analysis can be used to identify highly-vulnerable regions of a program; this allows software designers to determine which sections of code are most likely to fail due to a hardware error. This is useful in circumstances where the level of hardware fault tolerance is unknown to the programmer or where hardware protection is inadequate to meet the software developer's requirements.

We also show that the PVF of an architectural resource can be used to explain the AVF behavior of an underlying hardware structure. Architecture-level masking dominates the behavior of some structures, and PVF variation explains nearly all of the AVF variation in these structures. Microarchitecture-level fault masking is more pronounced in other structures; here, PVF explains less of the AVF variation. We analytically distinguish these structures from each other and show that for all structures, PVF captures the workload-dependent changes in vulnerability. Finally, we conduct a case study that examines two practical ways in which programmers can use PVF: selecting algorithms and choosing compiler optimizations to reduce the failure rate of a program.

The major contributions of this paper are:

- Introduction of the *Program Vulnerability Factor* as a method to calculate the level of architecture-level masking in a program and demonstration that PVF can be used to quantify the inherent vulnerability of a program;
- A rigorous analytical and experimental study into the ability of PVF to explain changes in AVF in a variety of hardware structures; and
- An example of a practical use for PVF: choosing algorithms and compiler optimizations in order to increase the soft error tolerance of a program.

The rest of this paper is organized as follows. Section 2 provides background on AVF and introduces the Program Vulnerability Factor. Section 3 demonstrates the use of PVF in assessing program vulnerability. Section 4 provides a rigorous statistical analysis of the ability of PVF to explain AVF changes in a hardware structure. Finally, Section 5 presents a case study on reducing program vulnerability via algorithmic changes and compiler optimizations.

2. Program Vulnerability Factor

In this section, we introduce the Program Vulnerability Factor. To provide the proper context for this discussion, we first review the Architectural Vulnerability Factor (AVF). We then describe how to calculate the Program Vulnerability Factor for any architectural resource and show that PVF is a component of the AVF of a hardware structure.

2.1. Architectural Vulnerability Factor

Mukherjee et al. introduced the concept of *Architectural Vulnerability Factor* (AVF) and *ACE Analysis* as a means to estimate SER early in the design cycle [2]. AVF is a well-defined, measurable quantity that can yield insight into the vulnerability behavior of a structure: the AVF of a processor structure is the probability that a fault in that structure will result in a visible error in the final output of a program. ACE Analysis estimates a structure's AVF by determining, during each cycle, which bits are required for *Architecturally Correct Execution* (ACE). A bit that is necessary for Architecturally Correct Execution is an ACE bit; all other bits are un-ACE bits. This allows a designer to estimate a structure's AVF in a single fault-free simulation run; therefore, ACE Analysis is significantly faster than prior techniques such as software fault-injection and is well-suited to early-design stage exploration [6]. As a consequence, ACE Analysis has enabled much research into the behavior of processor structures and into reliability improvement techniques [7] [8] [9].

To avoid ambiguity with later definitions, we call a bit within a hardware structure an *m-bit*. Therefore, the

AVF of a hardware structure is the fraction of m-bits in the structure that are ACE. For hardware structure H with size B_H (in m-bits), its AVF over a period of N cycles can be expressed as follows [2]:

$$AVF_H = \frac{\sum_{n=0}^{N} (ACE\ m\text{–}bits\ in\ H\ at\ cycle\ n)}{B_H \times N} \quad (1)$$

Recent work has suggested possible sources of error when performing AVF analysis. Wang et al. show that lack of model detail in ACE Analysis can lead to overly conservative AVF estimates [10]. This was shown to not be a fundamental limitation of the technique, however, and the authors themselves conclude that AVF Analysis is an indispensable tool for early-design reliability analysis [11]. Li et al. show that FIT rates calculated using average AVF values may be inaccurate since AVF values are often not independent [12]. This concern applies only to FIT rate calculation and therefore does not diminish the ability of AVF to provide insight into a hardware structure's behavior.

2.2. Calculating Program Vulnerability

A processor is a collection of hardware structures. Similarly, a *program* can be viewed as a collection of *architectural resources*. We define an architectural resource as any architecturally-visible structure or operation. This includes, for example, architectural registers and memory as well as operations defined by the ISA (e.g. addition, subtraction, etc). Each architectural resource R has an architecturally-defined size B_R in bits; we refer to bits in an architectural resource as architectural bits, or *a-bits*. For example, the Alpha ISA defines the size of the Integer Register File as thirty-two 64-bit registers; this architectural resource contains 2048 a-bits. (A physical register file *implementation* of this resource may contain a substantially greater number of m-bits.)

We also note that an architecturally-visible definition of *time* (i.e., the ordering of and distance between events to an architectural resource) can be given by the instruction flow: instructions are ordered with respect to each other and the distance between operations is the number of intervening instructions. Therefore, we have the ability to calculate the *vulnerability* of an architectural resource; we call this the *Program Vulnerability Factor* of the architectural resource. We refer to a-bits that are needed for correct operation as *ACE a-bits*; other bits are *un-ACE a-bits*. The PVF of an a-bit is then the fraction of time (in instructions) that the bit is ACE, and the PVF of an architectural resource R of size B_R over I instructions is the fraction of ACE a-bits in the resource:

$$PVF_R = \frac{\sum_{i=0}^{I} (ACE\ a\text{–}bits\ in\ R\ at\ instruction\ i)}{B_R \times I} \quad (2)$$

2.3. PVF as a Component of AVF

Equation 2 is an architectural notion of vulnerability. In this section, we show that the AVF of a hardware structure can be expressed in terms of the PVF of the architectural resource it implements; PVF is a component of AVF. This requires rewriting Equation 1 using terms from Equation 2; therefore, we must define conversion factors from a-bits to m-bits and from instructions to clock cycles. We use the example shown in Figure 2 to explain the procedure. This figure shows a series of operations to byte b in architectural memory. The left-hand side of Figure 2 shows these operations from the program's point of view; the PVF of b can be calculated as follows:

$$PVF_b = \frac{\sum_{i=0}^{I} (ACE\ a\text{–}bits\ at\ inst\ i)}{B_b \times I}$$
$$PVF_b = \frac{(8+8+8+0+8)}{8 \times 5} = 80\%$$

The right-hand side of Figure 2 shows how these operations map to a one-entry cache in hardware. The first load (in cycle 4) is fetched from memory since the cache does not contain address b. The subsequent load and store operations hit in cache. In this example, the AVF of the cache is 50% (b is ACE for 6 out of 12 cycles).

We first define a conversion between instructions and clock cycles by defining two parameters: n_i, which represents the number of clock cycles between the events in H that correspond to instruction i and instruction $i-1$; and N_I, which represents the CPI: $N_I := \frac{N}{I}$. In Figure 2, for instance, $n_2 = 2$ since there are two instructions between the Fill corresponding to instruction 1 and the Read corresponding to instruction 2; $n_3 = 2$; and $N_I = \frac{12}{5}$. Using these terms, we can write Equation 1 as:

$$AVF_H = \frac{\sum_{i=0}^{I} (ACE\ m\text{–}bits\ at\ inst\ i \times n_i)}{B_H \times I \times N_I}$$

The second conversion is from a-bits to m-bits. First, we define m_i as the number of m-bits in H accessed by instruction i divided by the number of a-bits accessed by i. For example, the first load operation in Figure 2 accesses one byte (8 a-bits) but misses in the cache (0 m-bits are read from cache). Therefore, $m_1 = \frac{0}{8} = 0$.

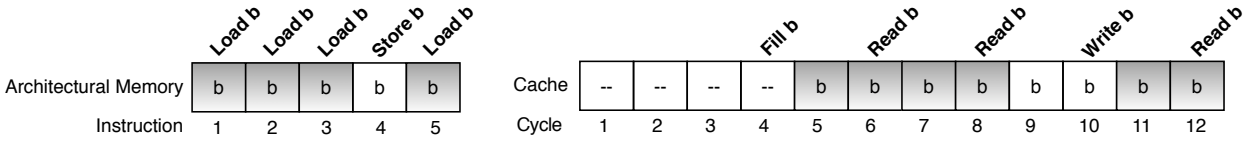


Figure 2. A set of program instructions and the resulting hardware operations. The first load instruction misses in cache; the remaining instructions hit in cache. The PVF of byte b is 80%, while the AVF of the cache line is 50%.

The second load operation also reads 8 a-bits but hits in the cache; therefore, $m_2 = \frac{8}{8} = 1$.

We also define M_H as the ratio of the size of the hardware structure to the size of the architectural resource it implements: $M_H := \frac{B_H}{B_R}$. In Figure 2, for example, $M_H = \frac{8}{8}$ since we assume both memory and cache contain only byte b.

Using this notation, we can further re-write Equation 3 as:

$$AVF_H = \frac{\sum_{i=0}^{I} (ACE\ a\text{–}bits\ at\ inst\ i \times m_i \times n_i)}{B_R \times I \times M_H \times N_I} \quad (3)$$

Equation 3 expresses AVF in terms of the three architectural parameters in Equation 2 ($ACE\ a\text{–}bits$, B_R, and I); these are the architecture-level fault masking components of AVF. For the example in Figure 2, Equation 3 evaluates as follows:

$$AVF_H = \frac{(8 \times 0 \times 4) + (8 \times 1 \times 2) + (8 \times 1 \times 2) + ...}{8 \times 5 \times \frac{8}{8} \times \frac{12}{5}}$$

$$AVF_H = \frac{48}{96} = 50\%$$

2.4. Properties of PVF

The PVF of an architectural resource will only change if either the program binary or its inputs are changed; PVF values do not depend on hardware parameters. Therefore, PVF values are directly comparable across programs independent of a particular hardware implementation, enabling insight into the reliability of software.

The remaining four parameters in the AVF equation (m_i, n_i, M_I, and N_I) are the microarchitecture-level fault masking component of AVF. These parameters behave differently within each hardware structure; if they vary substantially, PVF will not correlate well with AVF. However, if these parameters do not vary substantially, *a change in the PVF of an architectural resource will lead to a change in the AVF of the underlying hardware structure.* We explore the behavior of structures with different levels of microarchitectural masking in Section 4.

3. Analyzing Program Vulnerability

In this section, we examine the PVF profiles of several applications to generate insight into program reliability behavior. We expect to see differences in PVF behavior between programs as well as within a single program due to the well-known phase behavior of programs [13]. Walcott et al. examined the phase behavior of AVF and experimentally derived equations to predict AVF from microarchitectural state [14]; however, their models target a specific hardware implementation. Hardware-independent program vulnerability analysis allows insight based purely on architectural state. For instance, many software fault detection techniques (e.g., SWIFT [15]) more than double the dynamic instruction count while software recovery techniques can more than triple the instruction count [16]. This overhead can be reduced by adding redundancy only to highly-vulnerable regions of code, improving performance of code that is inherently fault tolerant.

3.1. Methodology

The PVF of a program can be estimated by multiple means. For instance, fault injection can be performed: a fault can be injected into a program's architectural state during its execution to determine whether the fault will be masked. Multiple injections can be performed to yield a statistically significant PVF estimate. Alternatively, PVF estimates can be generated using an ACE Analysis-like technique: a single fault-free execution of a program that determines, at each instruction, whether a bit in an architectural resource is ACE. Standard ACE Analysis techniques (e.g., marking dynamically-dead instructions as un-ACE [7]) can be applied using this method. Neither fault-injection nor ACE Analysis require any microarchitectural simulation to measure PVF; they can be performed using an architectural simulator or a dynamic binary translator such as Pin [17]. As a result, complex real-world programs can be profiled to completion, a task that is virtually impossible in many microarchitectural simulators.

3.2. Experimental Setup

In this work, we use the ACE Analysis-like methodology described in Section 3.1. We examine the PVF

Table 1. Simulated Machine Parameters

Parameter	Value
Issue / Commit Width	8 instructions
Physical Registers	256 integer / 256 Floating Point
IQ / ROB size	64 / 192 entries
Load-Store Queue	32 loads / 32 stores
L1 D-Cache	64 kB, 2 cycle access, 2-way SA write-back, allocate-on-miss
L1 I-Cache	32 kB, 2 cycle access, 2-way SA
L2 Cache	2 MB, 10 cycle access, 8-way SA write-back, allocate-on-miss
Memory Latency	200 cycles

behavior of two architectural resources: the Architectural Integer Register File and Architectural Integer ALU. We define the *Architectural Integer ALU* as the set of all architecturally-visible integer operations except multiplication and division; this definition allows apples-to-apples comparisons of PVF to the AVF of the physical ALU in our microarchitecture. We directly relate the PVF behavior of these architectural resources to regions of source code, allowing us to identify highly vulnerable functions and loops within a program. Section 5 demonstrates a practical use for this type of analysis: changing algorithms and compiler flags to lower the vulnerability of these regions of code.

Unless otherwise noted, all experiments were run using the full suite of SPEC CPU2000 benchmarks shown in Table 2 at the single early simulation points given by Simpoint analysis [18]; we typically show only a representative subset of results due to space considerations. For our experiments, we use the detailed CPU model in the M5 simulator [19]; this models an Alpha 21264-like CPU. Our baseline system configuration is shown in Table 1. We extend the CPU model to perform PVF measurements on architectural state and AVF measurements on microarchitectural state. We forgo the simulation speed benefits of architecture-only simulation in order to fairly compare AVF and PVF values. For experiments that sample vulnerability over time, we calculate vulnerability in 1M-instruction windows; therefore, a 100M instruction Simpoint consists of 100 individual measurements. For each measurement, we use a warmup period of 100M instructions plus the instructions from the start of the Simpoint to the measurement window and a cooldown period [9] of the remainder of the Simpoint plus an additional 100M instructions.

3.3. Architectural Register File

Figure 3 shows the Architectural Register File PVF for three benchmarks: *bzip2_source*, *mgrid*, and *equake*. All three benchmarks experience significant variation in PVF as they enter and exit different program phases. For example, *bzip2* enters a high-vulnerability phase after 68M instructions. This corresponds to the *generateMTFValues* function within *bzip2*; this function has nested loops and branch conditions which create significant register pressure, resulting in a high PVF. The other benchmarks exhibit similar behaviors: *equake*, for example, shows a periodic PVF behavior; each period corresponds to a single iteration of the time integration loop within the *main()* function. Although *equake* is a floating-point benchmark, it uses integer registers for loop control and memory addressing; this leads to a high PVF in portions of the loop that use many registers to hold memory addresses. The regions of low PVF correspond to loop segments that use only a small number of arrays; this requires fewer registers to hold memory addresses and decreases the vulnerability of the integer register file. A similar behavior occurs in the high-vulnerability region of *mgrid*.

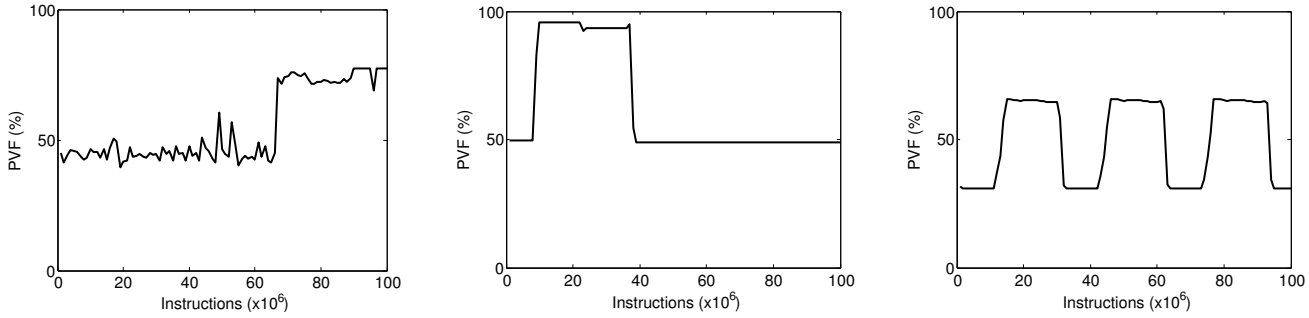
3.4. Architectural Integer ALU

Figure 4 shows the PVF of the Architectural Integer ALU for the same three benchmarks. The program phases that are evident in Figure 3 are also visible in Figure 4, although the PVF behavior of the Integer ALU differs from that of the register file. Overall, the PVF of the Integer ALU is significantly higher for the integer benchmark *bzip2* than for the floating-point benchmarks *mgrid* and *equake*; the high proportion of floating-point operations in the latter two benchmarks results in significantly reduced utilization of the ALU. The primary use of the ALU in these benchmarks is to calculate effective memory addresses from register and displacement values and to compute branch targets.

Equake in particular shows an odd behavior where the PVF of the Integer ALU has a strong negative correlation with the PVF of the Register File. This is due to the semantics of the program: the regions of low Register File PVF correspond to regions with many stores to memory from floating-point registers; however, each store still uses the Integer ALU to perform Effective Address calculation. The regions of high Register PVF have a larger percentage of floating-point ALU operations and memory accesses from integer registers that do not require address calculation; thus, the Integer ALU PVF decreases in these regions.

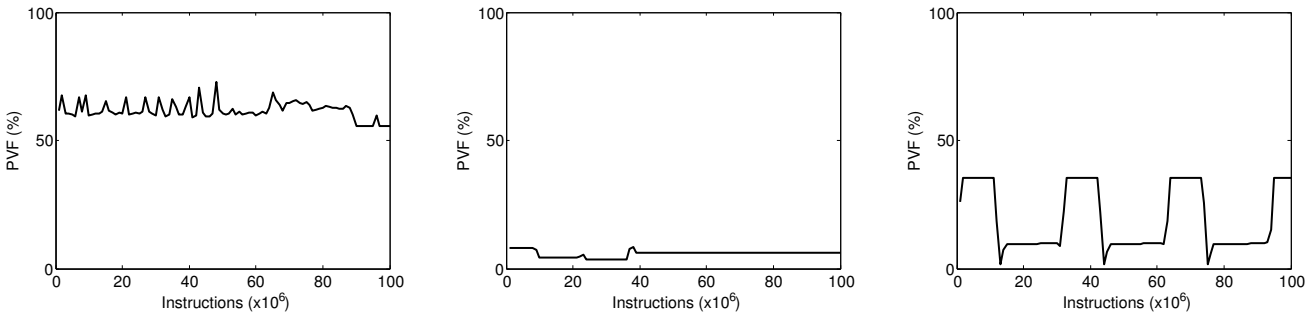
4. Using PVF to Explain AVF Behavior

The previous section demonstrated that PVF can yield insight into the vulnerability behavior of software, but did not answer a crucial question: how much does the PVF behavior of an architectural resource impact the AVF behavior of the underlying hardware structure? For example, how much of the behavioral



(a) bzip2_source (b) mgrid (c) equake

Figure 3. Architectural Register File PVF for *bzip2_source*, *mgrid*, and *equake*.



(a) bzip2_source (b) mgrid (c) equake

Figure 4. Architectural Integer ALU PVF for *bzip2_source*, *mgrid*, and *equake*.

difference in Figure 1(a) can be explained by differences in PVF? In structures with no microarchitecture-level masking, Equation 3 shows that all AVF variation will be caused by PVF variation. Conversely, in structures with significant microarchitecture-level masking, a smaller fraction of AVF variation will be explained by PVF. This section demonstrates how to analytically distinguish these structures from each other based on the behavior of the microarchitectural parameters in Equation 3.

4.1. Structures with Architecture-Level Fault Masking

PVF explains most of the AVF variation in structures where architectural masking dominates microarchitectural masking. This occurs when the microarchitectural parameters of Equation 3 or their quotient is a constant. In a register file, for example, n_i measures the CPI over the lifetime of a given register value and N_I is the CPI over the measurement window; therefore, $E[n_i] \approx N_I$. Furthermore, m_i is constant in a register file. There will be very little microarchitecture-level masking in structures that exhibit this type of behavior; their AVF will correlate strongly with PVF. Structures in this category include both register files and single-level (cache-less) memory subsystems.

We ran correlation experiments on the Integer Register File to test this hypothesis; Figure 5 shows the results for three benchmarks; results for other benchmarks are qualitatively similar. The PVF and AVF values span different ranges but clearly exhibit a high correlation. Table 2 shows the Pearson correlation coefficients between AVF and PVF for all benchmarks; the correlation is generally above 0.95. The square of the correlation coefficient (the *coefficient of determination*) is a measure of the amount of AVF variation "explained" by the correlation; this value is over 0.90 for most benchmarks, implying that over 90% of the AVF variation is explained by PVF variation. For benchmarks with low correlation (e.g., *mesa*, *perlbmk_makerand*, and *vpr_route*), a visual examination of the data (Figure 6) reveals nearly-constant AVF and PVF values. As a result, the correlation is susceptible to slight variations in AVF and does not reflect the true relationship between AVF and PVF.

To ensure the robustness of these results, we performed the same experiments but varied the number of physical integer registers, the number of ROB entries, and the number of IQ entries. Results are shown in Table 3. Although the range of AVF values (not shown) differs between microarchitectural configurations, the correlation between PVF and AVF remains strong

Table 2. Correlation between Architectural Register File PVF and Physical Integer Register File AVF.

Benchmark	Corr. Coef.	Benchmark	Corr. Coef.
ammp	0.983	bzip2_program	0.962
applu	0.999	bzip2_source	0.976
art110	0.967	crafty	0.971
equake	0.993	eon_rushmeier	0.910
facerec	0.682	gcc_166	0.944
fma3d	0.957	gzip_graphic	0.955
galgel	0.989	gzip_source	0.963
mesa	0.370	mcf	0.977
mgrid	0.998	perlbmk_makerand	-0.541
sixtrack	0.997	twolf	0.863
swim	0.853	vortex2	0.971
wupwise	0.972	vpr_route	0.259

Table 3. Register file AVF and PVF correlation across microarchitectures. Results for other benchmarks are similar.

PRF / ROB / IQ Size	bzip2_program	ammp	sixtrack
1024/1024/1024	0.952	0.988	0.996
256/192/64 (baseline)	0.962	0.983	0.997
256/256/64	0.962	0.983	0.997
128/128/64	0.962	0.983	0.997
64/16/64	0.985	0.979	0.999
64/16/32	0.985	0.979	0.999
64/16/16	0.985	0.979	0.999

across microarchitectural variation, showing less than a 4 percentage point difference across all configurations.

4.2. Structures with Both Architecture- and Microarchitecture-Level Fault Masking

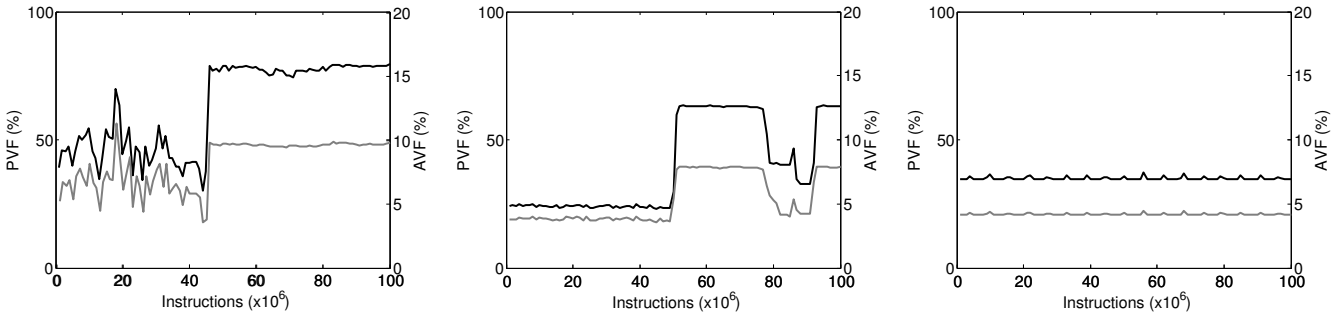
If m_i is not constant or $E[n_i] \not\approx N_I$, a structure will exhibit microarchitectural fault masking effects. In an execution unit, for example, n_i is the latency of the unit, while N_I is the CPI of the program; these values are unrelated. Therefore, PVF behavior will only partially explain changes in AVF: the AVF of an execution unit will be impacted by changes in PVF (architectural masking); changes in CPI (microarchitectural masking); and changes in latency (microarchitectural masking). However, a change in PVF will still tend to change AVF if it does not also result in a change in CPI (N_I) or latency (n_i). Structures that exhibit both architectural and microarchitectural masking include execution units as well as multi-level (caching) memory subsystems.

We ran correlation experiments on the Integer ALU to test this hypothesis. Our machine contains six fixed-latency hardware ALUs using round-robin scheduling; therefore, the AVF of each ALU is identical. Column 2 of Table 4 shows the AVF-PVF correlation over all the benchmarks; Figure 7 depicts the results for *bzip2_program*, *galgel*, and *sixtrack*. The correlation is high for benchmarks that exhibit minimal CPI variation across the simulation interval, including $ammp$, $applu$, and $wupwise$, and low for those that exhibit significant CPI variation such as $galgel$ and gcc. In $galgel$, the architectural ALU's PVF is nearly uncorrelated to the AVF of the hardware ALU; Figure 8 demonstrates that this is due to high variation in the CPI, meaning CPI effects dominate the AVF behavior. The PVF of $bzip2$ correlates more strongly to the AVF because the CPI is relatively constant for the second half of the execution window. In this region, the AVF behavior is dominated by changes in PVF, while the AVF in the first half of its execution is dominated by CPI variation. Finally, $sixtrack$'s CPI is nearly constant across the entire simulation interval; therefore the correlation between its AVF and PVF is quite high. Column 3 of Table 4 shows the correlation between PVF and AVF times CPI; the correlation is nearly perfect for all benchmarks. This indicates that changes in CPI and PVF are the sole contributors to the Integer ALU's AVF behavior, which is to be expected since our machine uses fixed-latency ALUs. (*Perlbmk* has nearly-constant CPI, AVF, and PVF; thus the low correlation does not reflect the true relationship between the variables.)

4.3. Structures with Microarchitecture-Level Fault Masking

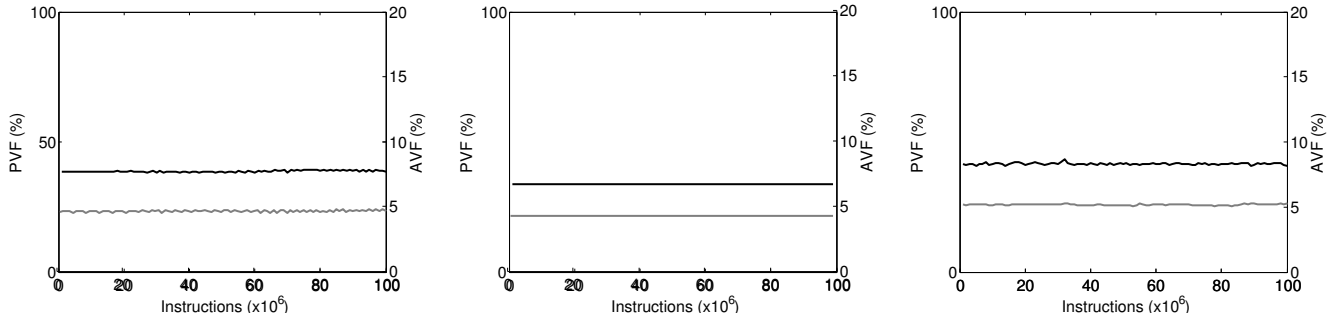
Many modern microprocessors contain structures that do not directly correspond to an architectural resource. However, we can also use Equation 3 to reason about the causes of AVF variation in these structures as a whole. Using the ROB as an example, and assuming every instruction is allocated a ROB entry: N_I is the average CPI; n_i is the ROB occupancy (in cycles) of instruction i; and the number of ACE a-bits per instruction is the number of non-dynamically dead bits in the instruction. Prior work has shown that these dynamically-dead bits in an otherwise ACE instruction have a small impact on ROB AVF relative to microarchitectural effects such as ROB occupancy [8] [10] [14]. Thus, the ROB AVF is dominated by the microarchitectural fault masking effects caused by changes in CPI and occupancy.

The principle of PVF allows us to quantify this intuition, but we can also take this conclusion a step further: since ROB AVF is dominated by microarchitectural masking, *it is impossible to deterministically reduce the AVF of the ROB in a completely microarchitecture-independent manner* (i.e., by reducing the PVF of some architectural resource). A program must have knowledge of the target microarchitecture in order to reduce the AVF of the ROB.



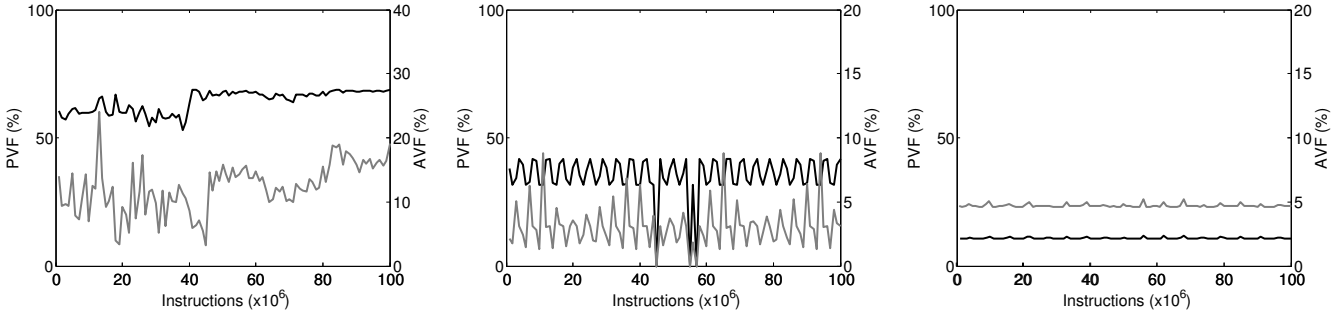
(a) bzip2_program (b) ammp (c) sixtrack

Figure 5. Physical Register File AVF (in gray) and Architectural Register File PVF (in black) for *bzip2_program*, *ammp*, and *sixtrack*.



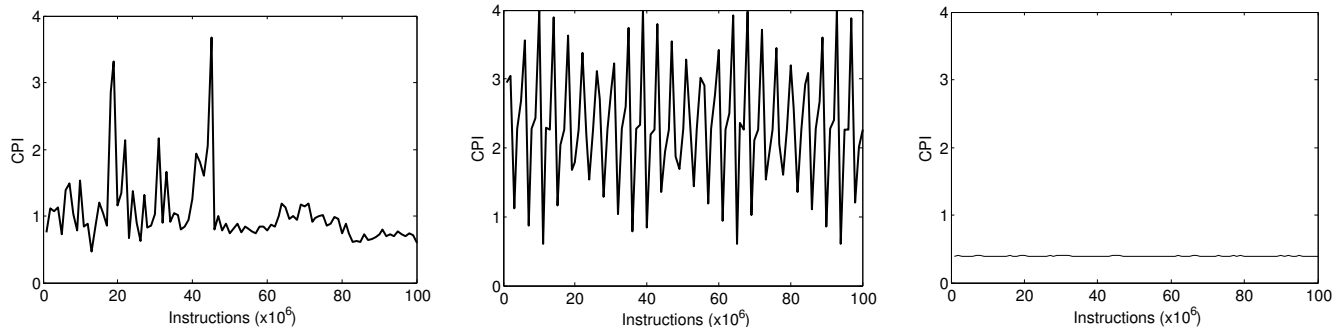
(a) mesa (b) perlbmk_makerand (c) vpr_route

Figure 6. Register File AVF (in gray) and PVF (in black) for *mesa*, *perlbmk_makerand*, and *vpr_route*.



(a) bzip2_program (b) galgel (c) sixtrack

Figure 7. Integer ALU AVF (in gray) and PVF (in black) for *bzip2_program*, *galgel*, and *sixtrack*.



(a) bzip2_program (b) galgel (c) sixtrack

Figure 8. CPI for *bzip2_program*, *galgel*, and *sixtrack*.

Table 4. Correlation coefficients between ALU PVF and AVF (Column 2) and PVF and AVF*CPI (Column 3).

Benchmark	PVF-AVF	PVF-AVF*CPI	Benchmark	PVF-AVF	PVF-AVF*CPI
ammp	0.875	0.999	bzip2_program	0.480	0.929
applu	0.999	1.000	bzip2_source	-0.704	0.816
art110	0.884	1.000	crafty	0.305	0.986
equake	0.982	1.000	eon_rushmeier	0.978	0.999
facerec	0.585	0.999	gcc_166	-0.885	0.997
fma3d	0.508	0.999	gzip_graphic	-0.757	0.904
galgel	-0.082	1.000	gzip_source	-0.374	0.938
mesa	0.865	1.000	mcf	0.589	1.000
mgrid	0.855	1.000	perlbmk_makerand	-0.948	0.286
sixtrack	0.962	0.998	twolf	0.475	0.806
swim	0.011	1.000	vortex2	-0.786	0.956
wupwise	0.990	1.000	vpr_route	-0.359	0.997

5. Case Study: Reducing Program Vulnerability

The previous sections have shown that PVF can yield meaningful insight into the soft error tolerance of a program and that reductions in PVF will tend to reduce AVF. In this section, we demonstrate two practical methods by which software developers can use PVF to assess and/or improve the robustness of their applications: a source code example that evaluates the reliability of several implementations of a sorting algorithm; and a compiler example that examines the impact of two different optimizations on error tolerance. The latter study is similar to recent work by Jones et al. [20], but their study was performed on a specific microarchitecture and there is no clear method to generalize their results to other hardware implementations. PVF analysis addresses this limitation.

The overall reliability of an algorithm is a function of both PVF and runtime: if algorithm A has the same PVF but twice the runtime of algorithm B, A is twice as susceptible to error. Therefore, we evaluate both PVF and *cumulative vulnerability*; an algorithm with a larger cumulative vulnerability is more susceptible to error regardless of runtime. To preserve microarchitectural independence, we use the number of dynamic instructions to estimate runtime; a hardware-specific algorithm could impart more detail by, for example, deriving CPI estimates based on the instruction mix.

We limit our study to the integer register file. This is not intended as a complete fault tolerance method, but rather as a demonstration of the opportunities provided for by PVF: a complete method would examine the vulnerability of all architectural resources. Limiting our study to the register file serves to highlight the basic approach that one would take in this endeavor.

5.1. Algorithm Implementation: Quicksort

Our first example examines the reliability of three publicly-available implementations of quicksort: an iterative implementation (*Quick-1*); a recursive implementation (*Quick-2*); and an iterative implementation that uses insertion sort on small sub-arrays (*Quick-3*). Each version is implemented in C and compiled using *gcc* version 4.1.0 with optimization level *–O*3. The dataset is a randomly-generated array of 1 million integer values between 0 and 1 million.

The PVF and cumulative vulnerability of each quicksort implementation are shown in Figures 9(a) and 9(b). The profiles of *Quick-1* and *Quick-2* are similar, but *Quick-1* executes fewer instructions and has a slightly lower cumulative vulnerability. *Quick-3*, however, executes 20% fewer instructions but has a 40% higher average PVF (52% versus 35% for *Quick-1*). This is due to nested loops in the insertion sort pass of *Quick-3*, which lead to higher utilization of the register file. As a result, the cumulative vulnerability of *Quick-3* is 20% higher than that of *Quick-1*.

This analysis highlights a tradeoff: a developer must decide whether to use the faster *Quick-3* algorithm or the more reliable *Quick-1* algorithm. Without PVF analysis, quantifying this tradeoff would require AVF analysis on every (potentially unknown) target microarchitecture. PVF analysis requires only one simulation to quantify this tradeoff on all microarchitectures.

To confirm that microarchitectural simulation supports the results of our PVF analysis, we analyze the AVF of each algorithm on our baseline microarchitecture. The results, in Figure 10, closely track the PVF analysis: *Quick-3* has a 15% larger cumulative AVF than *Quick-1*. The difference between analyses (15% versus 20%) is a result of the microarchitecture. *Quick-3* has poor cache behavior between cycles 150M and 200M, a period of low vulnerability. The IPC drops during this period, so its relative influence is larger in the AVF calculation. This is reflected as a "knee" in *Quick-3*'s cumulative AVF curve at cycle 150M.

5.2. Compiler Optimizations: Scheduling

Compiler optimizations can also have a significant effect on PVF. Figure 11 contains three *gcc*-generated versions of a loop from *bzip2*. Figure 11(a) shows the loop compiled using *–O*3; Figure 11(b)

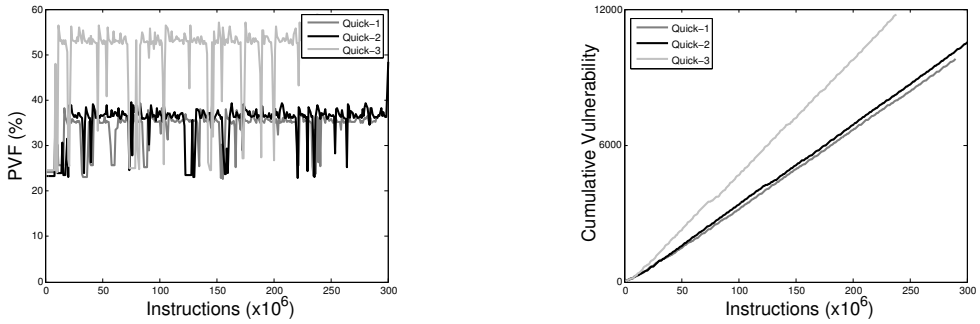


Figure 9. PVF and cumulative vulnerability (PVF * instructions executed) for three implementations of quicksort. *Quick-3* executes 20% fewer instructions but has a 40% higher PVF than *Quick-1*, leading to a 20% higher cumulative vulnerability.

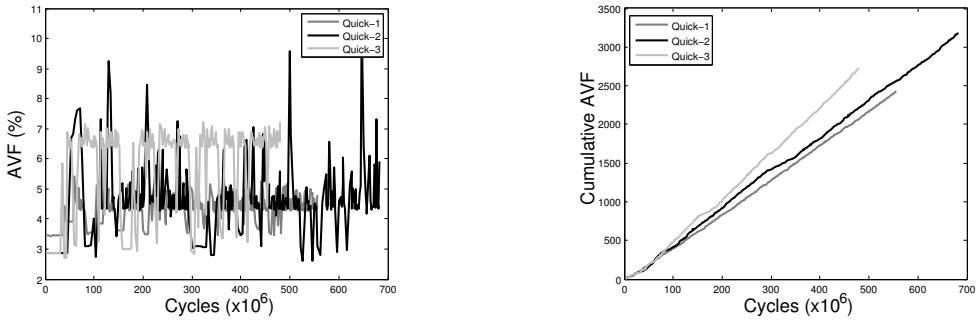


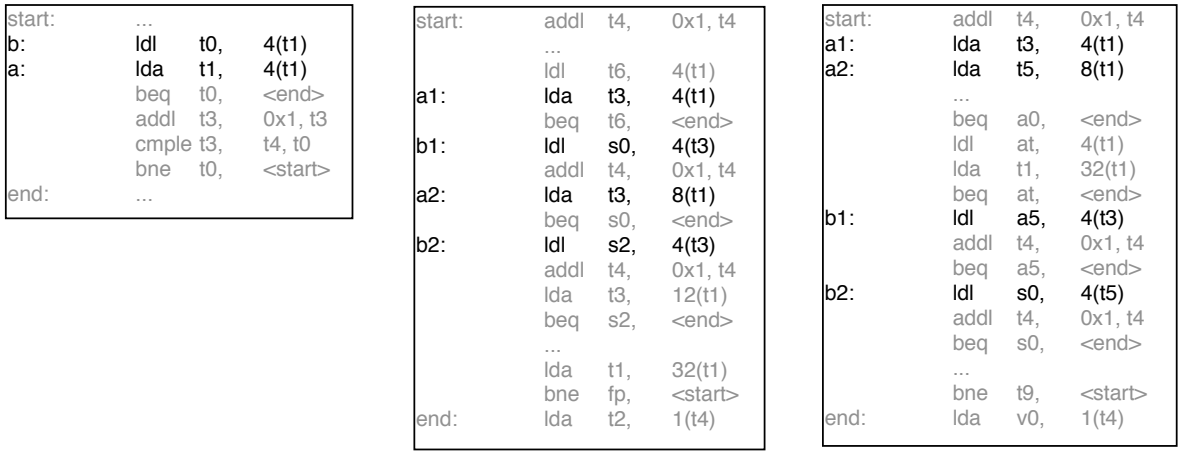
Figure 10. AVF and cumulative AVF distributions for each quicksort implementation on our baseline microarchitecture. The analysis leads to the same conclusion as the PVF analysis: *Quick-3* is 15% more susceptible to error than *Quick-1*.

used loop unrolling but without speculative scheduling (*–O3 –funroll–loops –fno–sched–spec*); and Figure 11(c) used unrolling with speculative scheduling (*–O3 –funroll–loops*). Speculative scheduling allows the compiler to move instructions across a branch boundary. This can improve performance, but also may result in the instruction being executed unnecessarily.

In each code fragment, the load-address instruction labeled a increments the value in $t1$; the destination register is eventually consumed by load instruction b. In Figure 11(a), register $t1$ is both source and destination and is vulnerable for the entire loop body. In Figure 11(b), register $t3$ is the destination; there are sections of code where this register is not vulnerable (e.g., between instructions $b1$ and $a2$). In Figure 11(c), however, the compiler has scheduled producer instructions $a1$ and $a2$ at the top of the unrolled loop for performance purposes. As a result, the instructions write separate destination registers ($t3$ and $t5$) which are both vulnerable for most of the loop body.

The original and unrolled loops have PVF values of 26.56% and 25.43%, respectively. The unrolled loop executes 15% faster, however, and therefore has a 16% lower cumulative vulnerability. The speculatively-scheduled loop, on the other hand, executes only 0.1% faster than the unrolled loop but has a PVF of 36.56%, a 43% increase over unrolling alone. As a result, speculative scheduling serves to increase the cumulative vulnerability of the loop by more than 40%.

The preceding example demonstrates that instruction scheduling can have a large effect on program reliability, especially in frequently-executed loops. Thus, we broaden our study to examine the PVF impact of disabling two compiler optimizations: speculative scheduling (*–fno–spec–sched*), and the instruction schedule optimization pass (*–fno–schedule–insns*). We perform our experiments on the C, C++, and Fortran 77 benchmarks listed in Table 2 using compiler setting *–O3 –funroll–loops* as our baseline. The results are shown in Figures 12 and 13. Disabling speculative scheduling results in an average PVF reduction of less than 2%. The PVF decrease is slight even in *bzip2* because the loop in Figure 11 only accounts for 3% of the benchmark's execution time. Therefore, although speculative scheduling has a large local effect, its overall impact is modest. The large increase in overall vulnerability for *mgrid* and *sixtrack* is caused by the large performance reduction when disabling speculative scheduling in these benchmarks. Figure 13 shows that disabling instruction scheduling altogether



(a) Original loop (b) Unrolled loop (c) Speculatively scheduled loop

Figure 11. A loop from *bzip2* compiled by *gcc* with optimizations: (a) *-O3*; (b) *-O3 -funroll-loops -fno-spec-sched*; and (c) *-O3 -funroll-loops*. In this example, allowing instructions $a1$ and $a2$ to be speculatively scheduled at the top of the loop increases the register file PVF by 43% without significantly improving performance.

reduces PVF by 5% on average; unfortunately, this also comes at a large performance cost. Therefore, the overall vulnerability increases significantly on average.

Results for other compiler flags that we examined ($-fsched\text{–}interblock$, $-fsched\text{–}spec\text{–}load$, $-O2$, $-O1$) are similar: none has a consistent effect on PVF or overall vulnerability. This potentially discouraging result is in some ways expected: current compiler techniques are designed to increase performance, not reliability. Unsurprisingly, these transformations have inconsistent effects on vulnerability. This points toward potential future work: compiler-based algorithms to increase reliability at minimal performance cost.

6. Conclusions

This work has established that the Program Vulnerability Factor (PVF) can generate insight into the reliability behavior of software. PVF is a microarchitecture-independent method of quantifying the architecture-level fault masking inherent to a program. Prior techniques are unsuitable for programmers with no access to a microarchitectural model or who wish to evaluate the vulnerability of their applications on *any* microarchitecture. PVF depends only on a binary and its inputs and allows programmers to quantify the vulnerability of a program. This analysis can highlight regions of code with high vulnerability and guide software designers in making judgments about where and how to add fault tolerance to their applications. We demonstrated that PVF quantifies the workload-dependent component of a structure's AVF and can explain much of the AVF behavior for certain hardware structures. We examined the Integer Register File and ALU in detail; both show significant workload-based AVF variation that is quantified by PVF. Finally, we presented a case study on reducing program vulnerability by examining three different implementations of the quicksort algorithm to determine which had the lowest overall vulnerability. We also examined the impact of compiler settings and demonstrated that certain code optimizations can cause large changes in PVF. This points the way towards the future work of investigating PVF-reduction algorithms at the compilation stage in order to increase program tolerance to transient errors. We believe that PVF represents an important next step in helping to develop more robust applications and can stimulate software-based fault tolerance research, an area that has been underrepresented in the literature to date.

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, Sept. 2005.

[2] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *International Symposium on Microarchitecture (MICRO-36)*, 2003.

[3] N. Seifert and N. Tam, "Timing vulnerability factors of sequentials," *Device and Materials Reliability, IEEE Transactions on*, vol. 4, no. 3, pp. 516–522, Sept. 2004.

[4] M.-L. Li *et al.*, "Understanding the propagation of hard errors to software and implications for resilient system

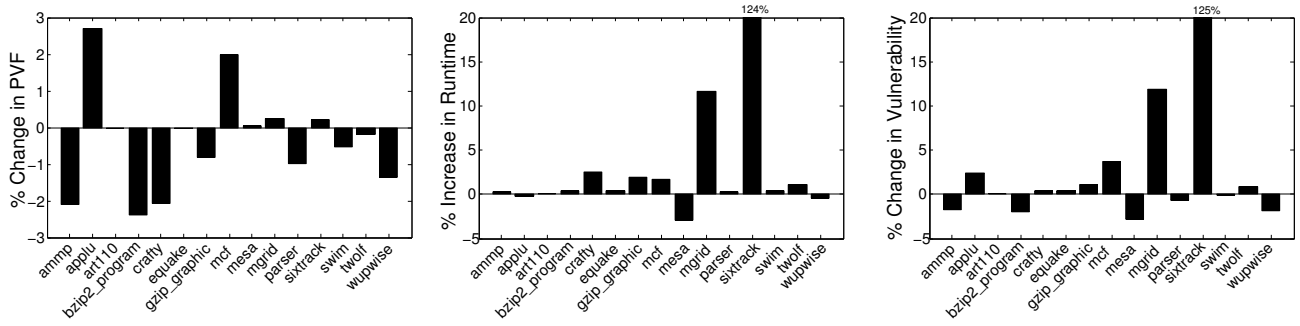


Figure 12. Percent change in PVF, runtime, and cumulative vulnerability when disabling speculative scheduling.

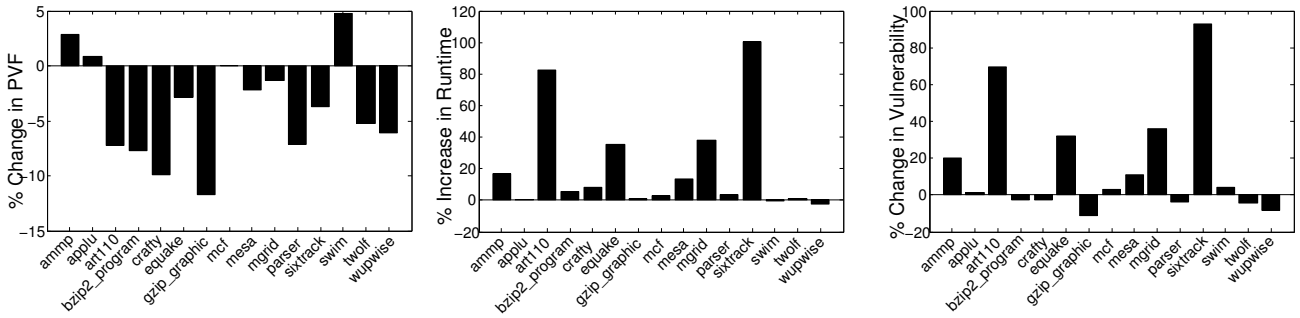


Figure 13. Percent change in PVF, runtime, and cumulative vulnerability when disabling instruction scheduling.

design," in *International conference on architectural support for programming languages and operating systems (ASPLOS-XIII)*, 2008.

[5] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WREFT-1)*, 2008.

[6] S. Kim and A. K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," in *International Conference on Dependable Systems and Networks (DSN-32)*, 2002.

[7] C. Weaver *et al.*, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *International Symposium on Computer Architecture (ISCA-31)*, 2004.

[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] A. Biswas *et al.*, "Computing architectural vulnerability factors for address-based structures," in *International Symposium on Computer Architecture (ISCA-32)*, 2005.

[10] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *International Symposium on Computer Architecture (ISCA-34)*, 2007.

[11] A. Biswas *et al.*, "Computing accurate AVFs using ACE analysis on performance models: A rebuttal," *IEEE Comput. Archit. Lett.*, vol. 7, no. 1, pp. 21–24, 2008.

[12] X. Li *et al.*, "Architecture-level soft error analysis: Examining the limits of common assumptions," in *International Conference on Dependable Systems and Networks (DSN-37)*, 2007.

[13] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Iternational Symposium on Computer Architecture (ISCA-30)*, 2003.

[14] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," in *International Symposium on Computer Architecture (ISCA-34)*, 2007.

[15] G. A. Reis *et al.*, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization (CGO '05)*, 2005.

[16] G. Reis, J. Chang, and D. August, "Automatic instruction-level software-only recovery," *Micro, IEEE*, vol. 27, no. 1, pp. 36–47, Jan.-Feb. 2007.

[17] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation (PLDI '05)*, 2005.

[18] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *International Conference on Parallel Architectures and Compilation Techniques (PACT-12)*, 2003.

[19] N. Binkert *et al.*, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, July-Aug. 2006.

[20] T. M. Jones, M. F. P. O'Boyle, and O. Ergin, "Evaluating the effect of compiler optimizations on AVF," in *Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-12)*, 2008.