Using Hardware Vulnerability Factors to Enhance AVF Analysis

Vilas Sridharan and David R. Kaeli
ECE Department
Northeastern University
Boston, MA 02115
{vilas, kaeli}@ece.neu.edu

ABSTRACT

Fault tolerance is now a primary design constraint for all major microprocessors. One step in determining a processor's compliance to its failure rate target is measuring the Architectural Vulnerability Factor (AVF) of each on-chip structure. The AVF of a hardware structure is the probability that a fault in the structure will affect the output of a program. While AVF generates meaningful insight into system behavior, it cannot quantify the vulnerability of an individual system component (hardware, user program, etc.), limiting the amount of insight that can be generated. To address this, prior work has introduced the Program Vulnerability Factor (PVF) to quantify the vulnerability of software. In this paper, we introduce and analyze the Hardware Vulnerability Factor (HVF) to quantify the vulnerability of hardware.

HVF has three concrete benefits which we examine in this paper. First, HVF analysis can provide insight to hardware designers beyond that gained from AVF analysis alone. Second, separating AVF analysis into HVF and PVF steps can accelerate the AVF measurement process. Finally, HVF measurement enables runtime AVF estimation that combines compile-time PVF estimates with runtime HVF measurements. A key benefit of this technique is that it allows software developers to influence the runtime AVF estimates. We demonstrate that this technique can estimate AVF at runtime with an average absolute error of less than 3%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault Tolerance

General Terms

Reliability

Keywords

Reliability, Fault Tolerance, Architectural Vulnerability Factor

ISCA'10, June 19–23, 2010, Saint-Malo, France.

1. INTRODUCTION

Fault tolerance is a primary design constraint for all major microprocessors. Chip vendors typically set a failure rate target for each design and strive to maximize performance subject to this constraint. To validate that a design meets the failure rate target, vendors perform extensive pre- and post-silicon analysis. This analysis measures the rate at which faults occur in a system as well as whether a given fault will cause a system failure (an *error*).

Some faults do not result in an error; these faults are said to be *masked*. Fault masking in a system can be quantified using the *Architectural Vulnerability Factor*, or AVF. The AVF of a hardware structure is the fraction of faults in that structure that affect correct program operation [11]. AVF is both simple to understand and easy to measure: *ACE Analysis* is a method to derive an upper bound on AVF using performance simulation. Therefore, recent research in the area of transient faults has used ACE Analysis to observe AVF behavior and propose novel applications for AVF [3] [9] [19] [23].

AVF is not without its limitations, however. In particular, AVF exploits very little of the abstraction that is present in modern computer systems. AVF provides device-level abstraction: it models multiple underlying hardware devices (e.g., SRAMs, latches) as basic memory elements. However, AVF provides for none of the abstractions at the higher levels of the system stack, such as the Instruction Set Architecture (ISA) or Application Binary Interface (ABI), that have been key drivers of modern computer architecture.

Some researchers have started to develop fault frameworks capable of exploiting these higher levels of abstraction. For example, Sridharan and Kaeli introduced the *Program Vulnerability Factor* (PVF) to quantify the portion of AVF that is attributable to a user program [21]. This allows a software designer to measure the microarchitecture-independent vulnerability of a program during its design phase.

Prior work does not, however, provide a method to quantify the non-PVF components of AVF, nor does it provide a method to re-compute AVF from PVF. In this paper, we introduce the *Hardware Vulnerability Factor* (HVF) to address these limitations. HVF quantifies the hardware portion of AVF, independent of program-level masking effects. AVF can then be calculated as the product of HVF and PVF.

Computing HVF has three concrete benefits which we examine in this paper. First, using HVF analysis (in conjunction with AVF analysis) provides insight to hardware designers beyond that gained by AVF analysis alone. Second, separating AVF analysis into HVF and PVF steps can

accelerate the AVF measurement process. Our results show a 2x reduction in simulation time with no loss of accuracy. Finally, runtime monitoring of HVF enables runtime estimation of AVF by combining HVF measurements with compile-time PVF estimates. A key benefit of this technique, relative to earlier AVF estimation techniques, is that it allows software developers to influence the runtime AVF estimates. We demonstrate that our technique estimates AVF with an average absolute error of less than 3%.

The rest of this paper is organized as follows. Section 2 provides background and discusses related work. Section 3 introduces the *System Vulnerability Stack*, our framework to separate AVF into hardware and software components. Section 4 demonstrates the use of HVF to provide insight to hardware designers about hardware behavior. Section 5 examines the use of HVF (in conjunction with PVF traces) to accelerate AVF modeling during processor design. Finally, Section 6 presents our technique to use HVF and PVF to estimate AVF at runtime.

2. BACKGROUND AND RELATED WORK

Mukherjee et al. introduced the *Architectural Vulnerability Factor* (AVF) and *ACE Analysis* as a means to estimate a processor's failure rate from transient faults early in the design cycle [11]. The AVF of a hardware structure is the probability that a fault in that structure will result in a visible error in the final output of a program. The authors call bits required for correct operation *ACE bits*; bits not required for correct operation are *unACE bits*. The AVF of a hardware structure is the fraction of bits in the structure that are ACE. For hardware structure H with size B_H, its AVF over a period of N cycles can be expressed as follows:

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

AVF quantifies full-system vulnerability. Some previously-introduced metrics can quantify the vulnerability of individual system components such as devices or user programs. The most relevant of these metrics is the *Program Vulnerability Factor* (PVF), which computes the vulnerability of a user program [21]. PVF can be computed across ISA-defined architectural resources (e.g., the architectural register file) and uses the dynamic instruction stream to measure time. The PVF of an architectural resource is the probability that a fault in that resource will result in a visible error in the final output of the program. For architectural resource R with size B_R, its PVF over a period of I instructions can be expressed as follows[1]:

$$PVF_R = \frac{\sum_{i=0}^{I} (Activated\ and\ exposed\ faults\ in\ R\ at\ instruction\ i)}{B_R \times I} \quad (2)$$

Other related work includes Seifert and Tam's *Timing Vulnerability Factor* (TVF) [16]. TVF quantifies device-level fault masking but does not extend to other system layers. Sanda et al. report experimental AVFs of a Power6 processor in *machine derating* and *application derating* components, but do not provide a method for *a priori* calculation of vulnerability [15]. Finally, Weaver et al. calculate DUE AVF (Detected Unrecoverable Error AVF), which excludes architectural sources of unACEness such as logical masking, and is similar to HVF in some respects [25]. DUE AVF incorporates structure-specific information about error handling, however, and is not guaranteed to match up with the non-PVF components of AVF. Furthermore, HVF and DUE AVF serve different purposes: HVF can guide decisions about redundancy, while DUE AVF uses existing redundancy to calculate a structure's DUE rate.

[1]Note that we formulate PVF using terminology from Section 3 of this paper.

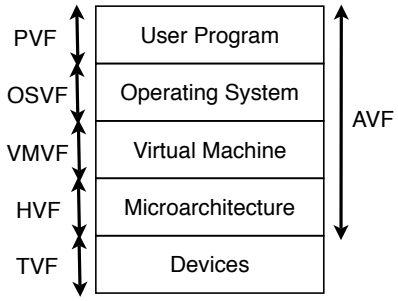


Figure 1: We calculate a vulnerability factor at every layer of the system. A bit is assigned a vulnerability at every layer to which it is visible. If it is vulnerable at every layer of the system, it is vulnerable to the system (i.e., its AVF is 1).

3. THE SYSTEM VULNERABILITY STACK

In this section, we introduce the *System Vulnerability Stack*, our method to separate AVF into individual vulnerability factors measured at each level of the system stack. We first introduce the basic concepts of the stack. We then explain how we use the stack to compute HVF, and how to re-compute AVF from its individual components.

3.1 Definitions and Concepts

We define a *fault* as a raw failure event such as a single-bit flip in a hardware structure. In this work, we refer to faults using the notation (b, n): b is the location of the fault (e.g., the bit in which it occurs), and n is the time of the fault (e.g., the cycle at which it occurs).

The basic underpinning of the vulnerability stack is the calculation of a vulnerability factor for each layer of the system stack (see Figure 1). A layer's vulnerability factor is the fraction of faults that cause incorrect operation of that layer. We define incorrect operation of a system layer as any disruption of the interface being implemented by that layer; we refer to this as an *error* in the layer. For instance, a hardware error is defined as any deviation in the semantics of the ISA being implemented. This includes faults that are propagated to ISA-visible state such as an architectural register, faults that entirely halt ISA function (e.g., a core deadlock), and any other behaviors that corrupt ISA state or disrupt program execution.

Not all faults in a system can cause an error in every system layer. For instance, a fault in a free physical register cannot cause a user program error without first propagating to program state. To determine the set of potential faults that can cause an error within a given system layer, we use the concept of *bit visibility*. A *visible bit* is a bit that is observable (accessible) by a particular system layer. For

instance, a bit in a free physical register is visible to the microarchitecture, but not to a user program. A bit in a valid cache line, on the other hand, is visible to both the microarchitecture and the user program. For a fault to cause an error within a given system layer, it must occur within a bit visible to that layer. Therefore, a program-visible fault can occur only in a bit that is visible to the architected state of the program. On a given system, the set of bits visible to an upper layer of the stack (e.g., the user program) is a subset of the bits visible to a lower level of the stack (e.g., the microarchitecture).

There are three possible consequences of a visible fault within a system layer. First, the fault can propagate to an interface implemented by the layer, thus becoming visible to another system layer. For instance, a microarchitecture-visible fault that propagates to ISA-visible state becomes visible to the user program. This fault has been *exposed* to the user program.

Second, a visible fault can create an error without being exposed to another system layer. For instance, a fault in an instruction scheduler unit might cause the unit to deadlock, freezing the system. This fault is not exposed to the user program, but still causes an error in the microarchitecture[2]. We refer to this fault as *activated* within the microarchitecture.

Finally, a visible fault that is neither activated within a layer nor exposed to another layer will not cause an error. We refer to these faults as *masked*.

A fault is either exposed, activated, or masked within every system layer to which it is visible. At a given layer, only exposed faults are made visible to higher system layers. For instance, masked and activated faults within the hardware are not visible to the operating system or user program. It is possible for a fault to be both activated within a layer and exposed to the next layer. However, we assume without loss of generality that an activated fault is not also exposed; that is, it is not visible to any higher system layers.

A fault's ACEness to the system will be determined by the highest layer to which it is exposed. If the fault is activated within that layer, it is ACE to the system. If the fault is masked within that layer, it is unACE to the system.

Two examples can help to illustrate the way the vulnerability stack treats a typical system. First, consider a bit in a free physical register whose contents will be overwritten when the register is mapped to an architected register. A fault in this bit will be masked within the microarchitecture and will not be visible to any higher system layers. Therefore, this fault will be unACE to the system.

Second, consider a bit in a register that is used as a load address. Further, assume that the load is dynamically-dead (i.e., its result is never used) [5]. A microarchitecture-visible fault in this bit will be exposed to the next layer of the system, typically the operating system. Assume the operating system does not examine the address, but allows the load to proceed normally. Thus, the OS exposes the fault to the user program. Finally, since the load is dynamically-dead, the fault is masked within the user program. As a result, the bit is unACE to the system.

Now instead assume that the operating system first performs a bounds check on the address and crashes the system on an out-of-range address. If the fault causes the address to be out-of-range, the fault will be exposed to the OS by the hardware, activated within the operating system, and not visible to the user program. Therefore, in this case the bit will be ACE to the system.

To compute the vulnerability of a system layer, activated and exposed faults are assigned a vulnerability factor (VF) of 1, while masked faults are assigned a VF of 0. In the next section, we show how to use these definitions to calculate the vulnerability of the hardware layer.

3.2 Computing Hardware Vulnerability

The *Hardware Vulnerability Factor* (HVF) of a hardware structure is the fraction of faults in the structure that are either activated within the hardware layer or exposed to a higher layer. The HVF of hardware structure H with size B_H over N cycles can be represented as follows:

$$HVF_H = \frac{\sum_{n=0}^{N} (Activated\ and\ exposed\ faults\ in\ H\ at\ cycle\ n)}{B_H \times N} \quad (3)$$

A hardware-visible fault is exposed to the user program if it will corrupt committed state. In a register file, for example, a fault within a physical register with a valid architectural mapping will be exposed to the user program. In instruction-based structures (e.g., Issue Queue, Reorder Buffer), faults are often exposed if they occur in an entry that eventually commits, or if the fault causes the instruction to incorrectly overwrite committed state. For instance, a fault in the destination register address of a squashed instruction is exposed if the faulty address points to an architected register and causes the instruction to overwrite a committed register value.

Whether a fault within a particular bit is activated is highly dependent on the details of the microarchitecture. For instance, a fault in a valid bit of an instruction scheduler might always be activated, while a fault in a valid bit of a load buffer might be activated only when the corresponding entry is occupied.

3.3 Computing System Vulnerability

The system vulnerability (AVF) of a bit is the product of its vulnerability factors in all system layers to which it is visible. In this section, we compute the AVF of a register in a system that consists of a microarchitecture and a user program. For simplicity, we assume that the VFs of all software layers of the system (e.g., virtual machine, user program) are captured as part of the PVF value.

Figure 2 shows a sequence of machine instructions and the corresponding events in physical register $P1$ of the microarchitecture. During cycles 4-6 and 12, $P1$ is not mapped to an architectural register, and a microarchitecture-visible fault during these cycles will be masked. Therefore, the HVF of all bits of $P1$ is 0. In the remaining cycles, $P1$ is mapped to architectural registers $R1$ and $R2$, and a microarchitecture-visible fault during these cycles will be exposed to the user program. In these cycles, the HVF of all bits of $P1$ is 1.

If a program-visible fault in $R1$ or $R2$ is masked by the user program, the bit is assigned a PVF of 0. For instance, a fault during cycle 14 will be masked if it does not change the result of the compare operation in cycle 15. PVF uses dynamic instructions to mark time. Therefore, the PVF

[2]Although this fault is not visible to the user program, the resulting error obviously still affects the program.

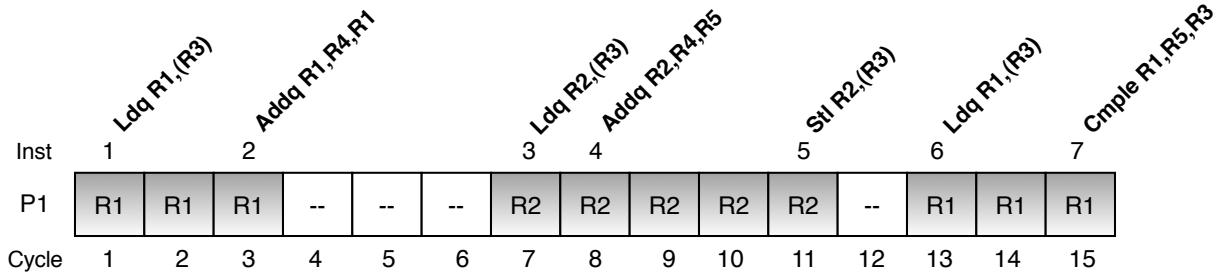


Figure 2: Physical register P1 is mapped to architectural registers R1 and R2 over 15 cycles. A microarchitecture-visible fault during cycles 4-6 and 12 will be masked in hardware. A fault during cycles 1-3, 7-11, and 13-15 will be exposed to the user program, creating a program-visible fault. The AVF of register P1 is a function of the HVF of P1 and the PVF of R1 and R2.

assigned to bit b of $P1$ during cycle n is the PVF of the architectural state a_b contained in b at the time (instruction) i_n, the instruction that consumes the value stored in $P1$ during cycle n. For example, in cycle 10, the AVF of $P1$ is the PVF of $R2$ at time 5 (instruction $Stl\ R2,(R3)$).

We can calculate the AVF of bit b in register $P1$ just using PVF values:

$$AVF_{b,1-15} = \frac{1}{15} \times (PVF_{R1_b,i_1} + PVF_{R1_b,i_2} + PVF_{R1_b,i_3} + PVF_{R2_b,i_7} + PVF_{R2_b,i_8} + PVF_{R2_b,i_9} + PVF_{R2_b,i_{10}} + PVF_{R2_b,i_{11}} + PVF_{R2_b,i_{13}} + PVF_{R1_b,i_{14}} + PVF_{R1_b,i_{15}})$$

We can replace $R1_b$ and $R2_b$ in the equation above with a_b, the architectural state assigned to b. Then we can compute $AVF_{b,1-15}$ by multiplying the HVF and PVF values assigned to b during each cycle, and summing over all 15 cycles:

$$AVF_{b,1-15} = \frac{1}{15} \sum_{n=1}^{15} HVF_{b,n} \times PVF_{a_b,i_n}$$

More generally, we can compute the AVF of a hardware structure H with size B_H over N cycles as:

$$AVF_{H,N} = \frac{1}{N \times B_H} \sum_{b=1}^{B_H} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{a_b,i_n} \quad (4)$$

3.4 Multi-Exposure Faults

Some faults that are exposed by the hardware to a user program will cause multiple program-visible faults when corrupted; we call these *multi-exposure faults*. Figure 3 gives an example: a fault within the Physical Source Register Index field of the IQ will cause an incorrect source operand to be fetched for the computation, resulting in up to 64 program-visible faults in the destination register.

The presence of multiple faults can change the behavior of a fault. Therefore, the PVF of a bit calculated using a single-fault model cannot be used to compute the AVF of a bit with a multi-exposure fault. Instead, we must calculate the AVF of a bit b with a multi-exposure fault as follows:

$$AVF_{b,N} = \frac{1}{N} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{A_{b,n}} \quad (5)$$

In this equation, $A_{b,n}$ is the set of all program-visible faults resulting from a microarchitecture-visible fault in bit b at cycle n. $PVF_{A_{b,n}}$ is 1 if and only if the set of program-visible faults $A_{b,n}$ causes the program to produce incorrect output. Calculating a precise value for $PVF_{A_{b,n}}$ requires modeling multiple simultaneous program-visible faults. This is tractable for a small set of faults, but is not feasible to evaluate in general.

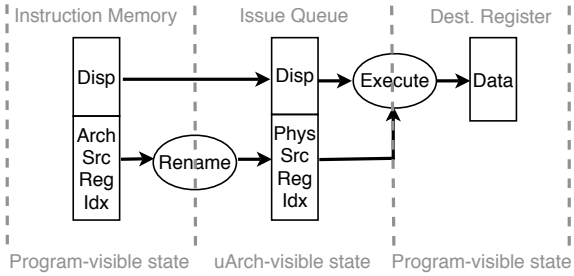


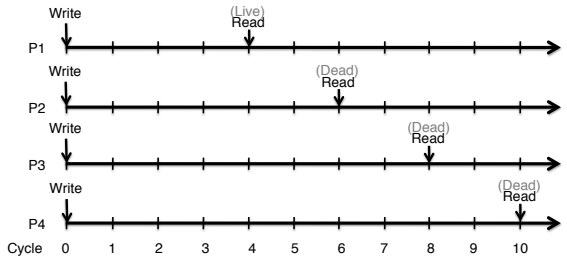
Figure 3: A single microarchitecture-visible fault that causes multiple program-visible faults is a *multi-exposure fault*. For instance, a fault in a bit in the Physical Source Register Index in the Issue Queue can cause multiple program-visible faults (e.g., in the destination register). To precisely calculate the AVF of a multi-exposure fault requires evaluating the impact of all the program-visible faults simultaneously.

In practice, there are many cases where we can conclusively determine the PVF of a multi-exposure fault. For example, a bit in the Physical Source Register Index field of the IQ will have a PVF of 0 (i.e., a fault in this bit will be masked within the user program) if the destination register is dead, logically masked by a subsequent instruction, or otherwise not needed for program correctness. In this work, unless we can conclusively prove that a multi-exposure fault is masked within the user program, we treat the bit as activated within the hardware.

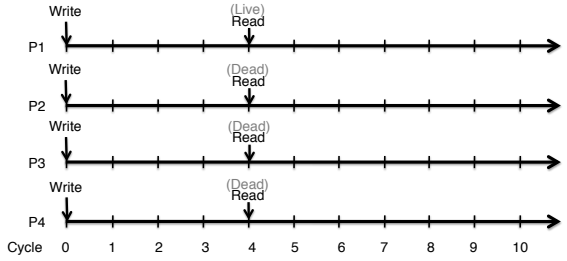
4. USING HVF FOR PROCESSOR DESIGN

Currently, architects use AVF to guide decisions such as where or when to add redundancy. For runtime reliability techniques (e.g., IQ squashing [25] or dynamic RMT [23]), this is appropriate. However, using AVF alone to evaluate the impact of design-time microarchitectural decisions (e.g., structure sizing) is suboptimal because AVF is not sensitive to operations that are masked at a program level, which can obscure changes in hardware behavior.

Figure 4 presents an illustrative example. The figure shows the behavior of four physical registers, P1 through P4, over 10 cycles, using two different scheduling algorithms. In Figure 4a, each register is written at cycle 0, and a sequence of reads occurs in cycles 4, 6, 8, and 10. We assume that the reads of $P2$, $P3$, and $P4$ do not affect program output due to logical masking later in the program's execution (i.e., the reads are dead). Since $P1$ is only ACE for 4 cycles, the AVF of $P1$-$P4$ over these 10 cycles is 10%.

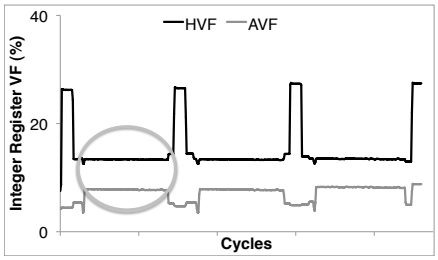


(a) Baseline

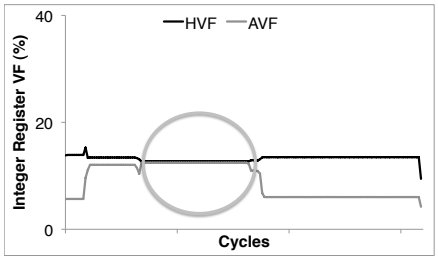


(b) Scheduler Change

Figure 4: The behavior of four physical registers over 10 cycles. In the baseline case, the registers have an HVF of 70% and an AVF of 10%. After a change to the scheduling algorithm, the registers have an HVF of 40% and an AVF of 10%. In this case, AVF does not capture the significant change to register behavior caused by the scheduler change. In contrast, the change in HVF will alert the designer to a change in potential vulnerability behavior.



(a) equake



(b) mgrid

Figure 5: The AVF and HVF of the register file for *equake* and *mgrid*. The circled areas indicate regions of similar register behavior, with similar HVFs (15%). Due to program masking, however, *equake* has a much lower AVF (8%) than mgrid (15%). If *equake* is run during processor design, a designer using only AVF will underestimate the potential vulnerability of this behavior. Therefore, it is important to understand the causes of high HVF in a hardware structure.

Now assume we make a change to the instruction scheduler, which results in the behavior indicated by Figure 4b. We wish to examine the vulnerability impact of this design change. Unfortunately, the AVF of Figure 4b is still 10%, since the design change only altered the behavior of dead reads.

In the field, a different workload (or the same workload with different input data) may execute the same sequence of operations, but a different subset of the reads may be dead, resulting in a change in AVF. In this case, a designer would like to know that the scheduler change will have a potential impact on vulnerability. Unfortunately, AVF is not sensitive to the placement of dead reads, and does not provide insight into this behavior change.

If, instead, the designer also measured the HVF of the registers, he/she would see that the HVF in Figure 4a is 70%, while the HVF in Figure 4b is 40%. This change in HVF alerts the designer to the significant change in behavior. This is an example of a situation in which HVF gives us insight into potential vulnerability changes that AVF does not. Therefore, we suggest that HVF is a useful metric to examine in conjunction with AVF.

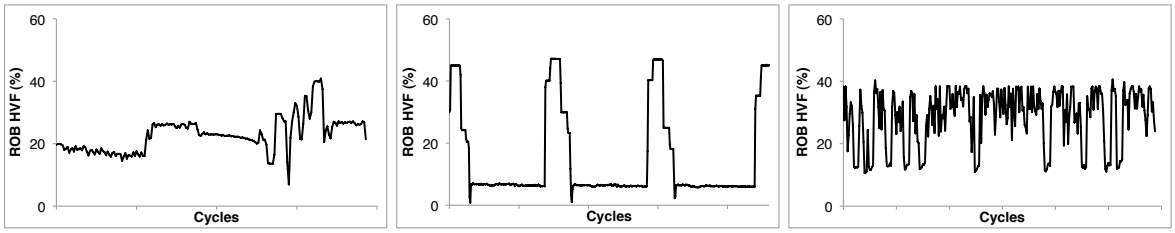
A similar real-world example is presented in Figure 5, which shows the AVF and HVF of *equake* and *mgrid*. In the circled regions, the two workloads perform a similar series of register operations. This is reflected in their similar HVFs of approximately 15%. However, due to program masking characteristics, the AVF of *equake* is only 8%, while the AVF of *mgrid* is 15%. If a hardware designer runs *equake* in simulation, he/she will significantly underestimate the potential AVF of this register behavior in the field. Thus, it is important to understand behaviors that cause high HVF, regardless of the AVF of a particular workload.

Finally, a comparison to performance evaluation is also instructive. Assume we have a workload whose goal is to maximize transactions executed per cycle (TPC). There are two components to TPC: the transactions-per-instruction rate (TPI), and the number of instructions per cycle (IPC). Increases in both TPI and IPC will increase TPC, but IPC is the only metric within the control of a hardware designer. Therefore, microarchitects focus on improving IPC during processor design. Similarly, when looking to maximize reliability, HVF is the only metric within the control of the hardware designer, and microarchitects should focus on improving HVF during processor design. (Note that, just as the ultimate goal of performance evaluation is to improve TPC, our ultimate goal is to improve AVF.)

4.1 Measuring HVF

Both activated and exposed faults can be identified using fault injection or ACE Analysis, in either a performance model or in RTL. In this work, we measure the HVF of a hardware structure using a modified version of ACE Analysis: we assume that all faults are activated unless we can prove they are exposed to another layer or masked. Determining that a fault is exposed to another layer is, at maximum, as difficult as determining that a bit is ACE during AVF computation. Determining that a fault is exposed is often quicker than determining it is ACE since a value need only be tracked until it is visible to the architected state.

We examine the HVF of the Issue Queue (IQ), Reorder Buffer (ROB), Load Buffer (LDB), Store Buffer (STB), and Integer Register File (RF). Within each structure, we split each entry into multiple subfields and measure the HVF of each subfield independently. For example, an Issue Queue entry contains several fields, including a destination register



(a) ammp (b) equake (c) mcf

Figure 6: The HVF of a hardware structure is the fraction of faults that are activated within the layer or exposed to another layer. The HVF behavior of microarchitectural structures such as the Reorder Buffer varies substantially across benchmarks, increasing when the structure is full of correct-path instructions, and decreasing when the structure is empty or contains mostly wrong-path instructions.

Parameter	Value
Issue / Commit Width	8 instructions
IQ / ROB / LDB / STB	64 / 192 / 32 / 32 entries
Physical Registers	256 Int. / 256 FP
L1 I-Cache	32 kB, 2 cycle access 2-way SA
L1 D-Cache	64 kB, 2 cycle access, WB 2-way SA, allocate-on-miss
L2 Cache	2 MB, 10 cycle access, WB 8-way SA, allocate-on-miss
Memory Latency	200 cycles

Table 1: Simulated machine parameters used in this paper.

address and control information for the functional units. We measure the HVF of each field separately and calculate a weighted average to compute IQ HVF.

Figure 6 plots the HVF of the Reorder Buffer for three typical benchmarks from the SPEC2000 suite. The dramatic changes in HVF represent shifts in system behavior. Typically, the high HVF phases are the result of memory stalls or other events that cause the structure to fill with correct-path instructions. Low HVF phases (e.g., in *equake*) will result when the structure is relatively empty (e.g., as a result of low fetch bandwidth due to instruction cache misses). Low HVF can also be caused by a high incidence of flushes (e.g., from mispredicted branches) when the structure contains mostly wrong-path instructions.

4.2 Using HVF for Design Exploration

In this section, we present the results of a microarchitecture exploration study performed using HVF. For all experiments, we use the detailed CPU model in the M5 simulator modeling an Alpha 21264-like CPU [2]; our baseline system configuration is shown in Table 1. All experiments were run using the SPEC CPU2000 benchmarks at the single early simulation points given by Simpoint analysis [12]. We use a 100M-instruction warmup window and a 100M-cycle cooldown window for all simulations [3].

Assume that the goal of our study is to choose the optimal number of store buffer entries for our microarchitecture, subject to performance and reliability constraints. Therefore, we vary the number of store buffer entries while monitoring the HVF of several large structures and the average CPI of the workloads. Figure 7 shows the results averaged over all benchmarks. A 32-entry Store Buffer increases performance by approximately 2% over a 16-entry Store Buffer (STB). However, this is accompanied by a 16%, 25%, and 30% increase in the HVF of the Load Buffer (LDB), Reorder Buffer (ROB), and Issue Queue (IQ), respectively. In addition, the HVF decrease in the Store Buffer (from 7.2% to 3.2%) is offset by the 4x increase in its size. Therefore, by choosing a 16-entry Store Buffer, we can derive most of the performance benefit of a 32-entry Store Buffer but substantially improve overall reliability.

In general, our experiments have shown that the AVF of microarchitectural structures with architectural equivalents (such as register files) shows low correlation to HVF, while the AVF of structures with no architectural equivalent (e.g., issue queue, reorder buffer) shows good correlation to HVF.

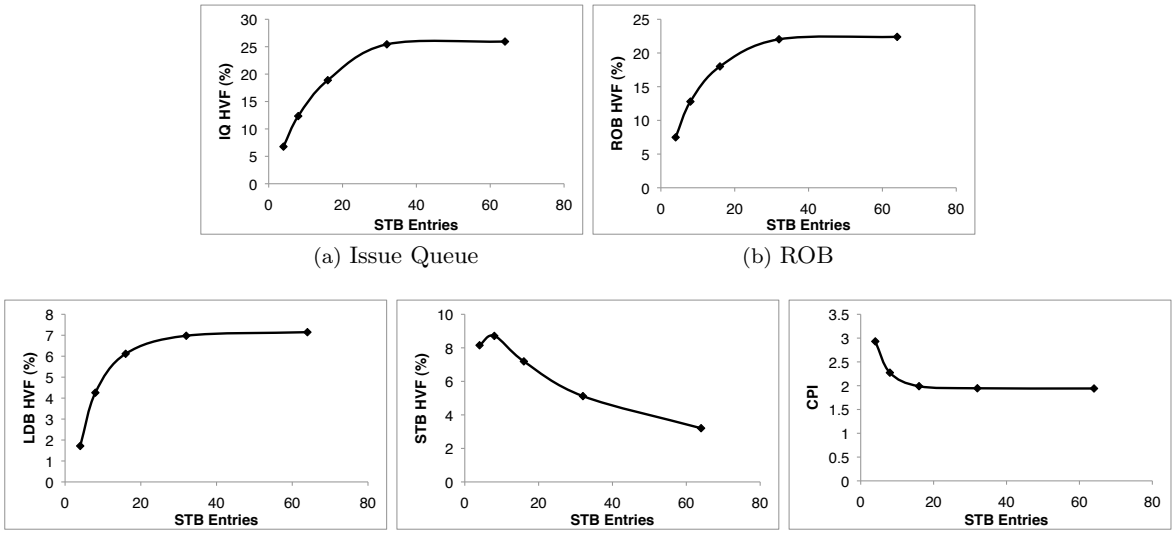
4.3 Using Occupancy to Approximate HVF

Previous studies have shown that a structure's AVF increases with its occupancy, leading to a high correlation between AVF and structure occupancy [6] [19] [23]. These studies have demonstrated reasons for this correlation based on performance statistics (events such as memory stalls that lead to high occupancy also tend to increase AVF), but no study has demonstrated the component(s) of AVF that are responsible for the correlation. We expect that the correlation between occupancy and AVF is due to a correlation between occupancy and HVF, and that significant deviations in the AVF/occupancy relationship are due to program-specific (PVF) effects.

Figure 8 shows that committed instruction occupancy is a good but conservative predictor of HVF in the ROB and IQ. The correlation coefficients between HVF and occupancy are high (greater than 0.97 for all benchmarks). However, the IQ HVF is approximately 65% of the IQ occupancy on average; for certain benchmarks, the HVF is only 50% of the average occupancy. This result has an important implication for AVF measurements in complex hardware structures. Many techniques provide an upper bound on AVF by using the simplifying assumption that all bits in an entry are ACE if any bit within the entry is ACE (e.g., [8] [9] [19]). (Generally, this is due to infrastructure limitations which result in occupancy being used as a heuristic for HVF.) While an upper bound on AVF is often desirable in design settings, our results indicate that a more detailed implementation is needed if the goal is to provide an accurate estimate, rather than a bound, for AVF.

5. BOUNDING AVF AT DESIGN TIME

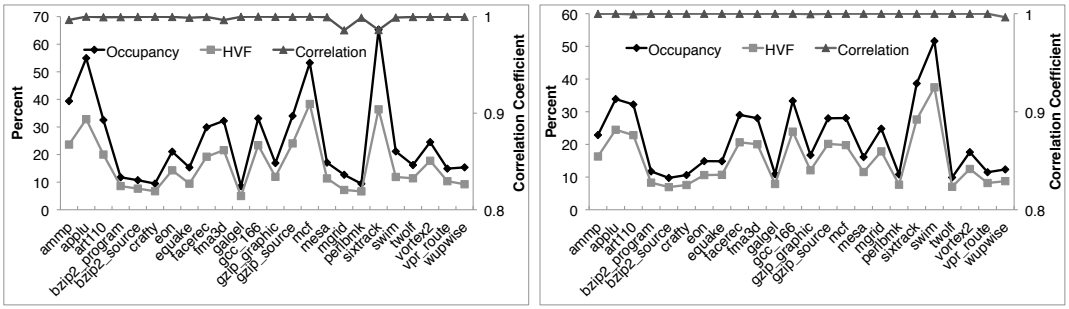
The previous section discussed the use of HVF during the processor design cycle for activities such as microarchitectural exploration. When attempting to determine a processor's soft error rate, however, designers will want to measure the AVF of the processor, including workload effects. In this section, we discuss how to use HVF to improve AVF simulation during processor design. Note that all HVF and PVF measurements are performed using full ACE Analysis (i.e., not calculated using heuristics).



(a) Issue Queue (b) ROB

(c) Load Buffer (d) Store Buffer (e) CPI

Figure 7: Microarchitectural exploration using HVF. We vary the size of the Store Buffer (STB) and compute the average HVF of the IQ, ROB, LDB, and STB, and the CPI across all benchmarks in the SPEC CPU2000 suite. Increasing the Store Buffer from 16 to 32 entries provides a 2% performance boost at the cost of a 25% increase in vulnerability.



(a) IQ (b) ROB

Figure 8: The relationship between HVF and committed instruction occupancy across benchmarks for the IQ and ROB. HVF is, on average, 65% of occupancy in the IQ and 72% of occupancy in the ROB. However, we find correlation coefficients between HVF and occupancy (using 100 samples per Simpoint) to be greater than 0.97 across all benchmarks. This indicates that committed instruction occupancy is a good but conservative heuristic for HVF.

We simulate AVF using a two-step process. First, we perform PVF simulation of a program and save the results in a *PVF trace*. This step happens offline (i.e., not on the critical path of a processor design) in architecture-only simulation (e.g., using Pin [10]). Second, during the hardware design phase, we perform HVF simulation of the microarchitecture. For a given instruction or data value, we determine its HVF and then retrieve the corresponding PVF value from the trace. The simulator then multiplies the two values to determine the AVF of the value in question.

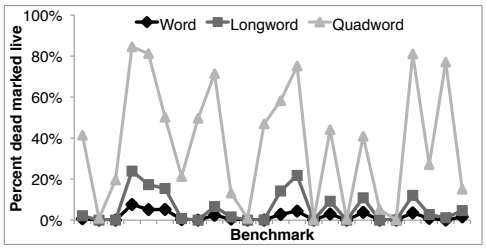
Since benchmarks are typically used over multiple chip generations, PVF traces can be re-used over multiple processor designs. This also allows us to increase the accuracy of PVF analysis without increasing simulation time during processor design. For instance, we assess effects such as transitive logic masking, which take significant simulation time, that are often omitted from AVF simulation [11].

Our methodology resulted in an approximately 2x reduction in the time required for AVF simulation relative to standard ACE analysis. The speedup is due to two factors. First, we no longer need to perform PVF calculations (e.g., dynamic-dead analysis) during microarchitectural simulation. Second, we drastically reduce the amount of state that the simulator needs to track. Standard ACE analysis defers analysis of an event until after the corresponding instruction has been analyzed in a post-commit window which can be tens of thousand instructions in length [11]. This requires the simulator to maintain large event history buffers within each hardware structure. Our methodology defers analysis of an event only until after HVF analysis, which typically completes soon after the corresponding instruction commits. This significantly reduces the average size of the history buffers and yields a much smaller memory footprint for the simulator itself. Our experiments show an approximately 3x reduction in memory usage for each simulation.

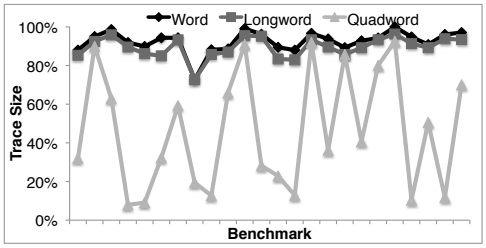
5.1 Capturing PVF Traces

The primary difficulty with offline PVF analysis is capturing PVF traces such that they can be easily mapped to simulation state during HVF simulation. To do this, we exploit the fact that ACE Analysis monitors events (e.g., reads and writes) in each structure. A bit within the structure is ACE if it is read by an event that affects the outcome of the program (a *live event*). A bit is unACE if it is not read; or if it is read only by *dead events*.

During PVF simulation, we assign a unique ID to each in-



(a)



(b)

Figure 9: PVF traces can be used to help estimate AVF during hardware design. There is a tradeoff between the accuracy of PVF analysis and the size of PVF traces. For example, changing the granularity at which we perform PVF analysis from 1 to 8 bytes decreases the number of dead events identified, but significantly reduces the trace size. We show results for PVF analysis at different granularities: word (2 byte), longword (4 byte), and quadword (8 byte), normalized to byte granularity.

struction. If an instruction generates a dead event, we record its ID in the trace along with identifying information such as the type of event, the register or memory target, and the deadness of each byte within the event. During HVF simulation, we recreate the ID of each instruction and check the trace for any dead events associated with that instruction. If an instruction generates a microarchitecturally-live event, we apply the deadness value from the trace to determine the event's liveness to the system. We use the resulting liveness value to make ACE/unACE determinations within each hardware structure.

Increasing the accuracy of PVF simulation will lead to more dead events and a larger trace, but will also lead to a better estimate of PVF. Figure 9 presents an example of one such tradeoff: the granularity at which events are recorded in the trace. Reducing the granularity of the analysis decreases trace size at the expense of accuracy. (In this work, we record traces at a byte granularity.)

5.2 Results

Dead events in a PVF trace arise from effects such as static and dynamically dead instructions, transitively dead instructions, static and dynamic logical masking, and transitive logical masking. Figure 10 shows that, on average, 57% of dead events in our PVF traces are the result of dead instructions, while 33% and 9% of dead events result from first-level and transitive logical masking, respectively. Interestingly, this breakdown is not reflected in the contribution to AVF of each of these sources. For example, in the IQ, dead instructions reduce the AVF bound by 24% on average, while first-level and transitive logic masking combined provide only a 1% further reduction. Mukherjee et al. found a similar result; they showed that first-level logic masking had a 1% impact on the AVF of the execution units [11]. Our results extend these findings to all structures that we tested and to both first-level and transitive logic masking.

Figure 11 shows the impact of using PVF traces with HVF simulation to bound AVF. In all cases, incorporating PVF information significantly reduces the AVF estimate achieved using HVF-only simulation. Accounting for PVF yields a 62% reduction in the AVF bound for the register file, primarily due to the effects of dead and ex-ACE register values. In the other structures, the use of PVF traces also results in significant reductions in the AVF bound: 25% for the IQ, 27% and 30% for the Load and Store Buffers, and 22% for the ROB.

6. ESTIMATING AVF AT RUNTIME

Sections 4 and 5 demonstrated that HVF can be used to improve the hardware design cycle. Previous work has shown that PVF can be used during software design to statically reduce program vulnerability [21]. In this section, we demonstrate that the vulnerability stack can also be used to *dynamically* impact system reliability at runtime. In particular, we demonstrate how to combine compile-time PVF estimates with runtime HVF estimates to monitor AVF at runtime. A runtime AVF monitor can allow a system to tune its redundancy capabilities, enabling protection when AVF is high and disabling it when AVF is low (e.g., [23]). AVF monitoring can be used in conjunction with redundancy techniques (e.g., [1], [7], [13]) to provide highly-effective redundancy with low overhead.

Previous runtime AVF estimation techniques fall into two basic categories: training-based predictors [4] [6] [23]; and runtime-only predictors [9] [19]. The training-based predictors use a set of training benchmarks to determine a relationship between AVF and hardware statistics that can be measured at runtime. For instance, Walcott et al. use linear regressions to generate a microarchitecture-specific predictor equation that allows AVF estimation based on statistics that are easily measurable at runtime [23]. Similarly, Duan et al. use Boosted Regression Trees to correlate AVF with processor statistics and extend this to predict correlations across microarchitectural changes [6].

Runtime-only predictors, on the other hand, attempt to measure (or bound) AVF directly at runtime. For instance, the predictor proposed by Soundararajan et al. uses the ROB's instruction occupancy to bound its AVF [19]. Another technique, proposed by Li et al., uses simulated fault injection tracked via *error bits* attached to each hardware structure [9]. If a fault propagates to a pre-defined failure point, the injection is said to result in an error. The system computes AVF as the number of errors divided by the number of injections.

6.1 Methodology

Our key observation is that none of the predictors discussed above include any mechanism for the programmer to influence the AVF computation; they all require hardware designers to predict software reliability behavior. To do this, hardware designers generally rely on the behavior of typical software. For instance, the predictor proposed by Li et al. assumes a fault is ACE if it propagates to memory. This assumption falls apart in many software applications that deviate from the norm. For example, gaming and multimedia applications are memory-intensive yet extremely fault-tolerant [17] [22]. As a result, every predictor discussed so far will likely overestimate the AVF of these applications.

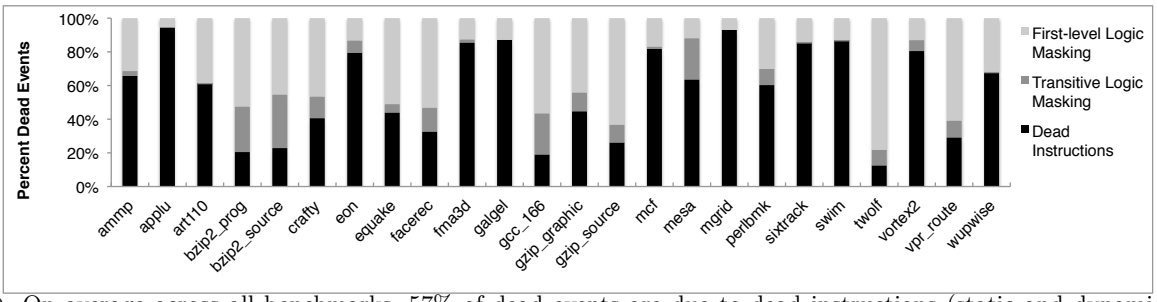
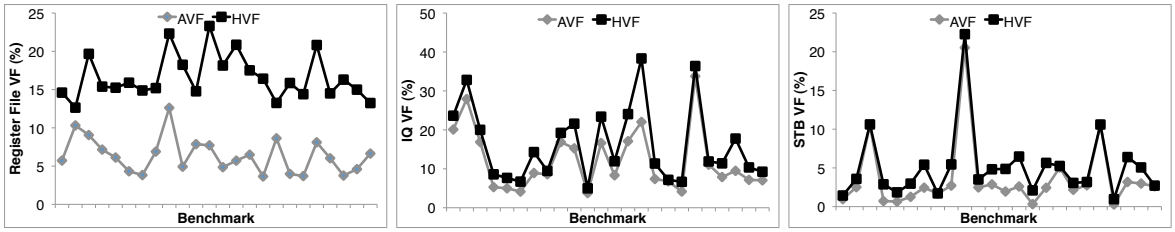


Figure 10: On average across all benchmarks, 57% of dead events are due to dead instructions (static and dynamic), while 33% and 9% of dead events are due to first-level and transitive logic masking, respectively. Since PVF analysis is performed offline, we can perform expensive analyses such as detection of transitive logic masking without increasing simulation time during hardware design.



(a) Register File (b) Issue Queue (c) Store Buffer

Figure 11: Incorporating PVF traces tightens the AVF bound by 60% in the register file, 25% in the Issue Queue, and 30% in the Store Buffer. Results for the ROB and Load Buffer are similar to those of the IQ and Store Buffer.

Similarly, software-based redundancy techniques such as SWIFT [14], PLR [18], or SRMT [24] will not be recognized by any of the hardware-based predictors. In fact, these techniques typically increase processor utilization. This will likely lead to higher AVF estimates than for the same application without redundancy. The problem is further complicated if a mix of redundant and non-redundant code is executing on the same system.

The vulnerability stack addresses this problem by separating the analysis of program behavior (PVF) from hardware behavior (HVF). We can calculate HVF estimates in hardware at runtime, but allow each program to supply its own PVF estimates. This allows each program to specify its level of fault tolerance to the system in a microarchitecture-independent format. Finally, the system can combine the HVF and PVF estimates to generate an overall AVF estimate.

To communicate PVF estimates to hardware, we propose a set of registers called the *Program Vulnerability State* (PVS). These registers are used by software to communicate PVF estimates of architectural resources to the system. We implement one PVS register for the integer register file and one for the floating-point register file, and one for instruction-based structures to store the fraction of dead instructions in the dynamic instruction stream. Since ECC protection is standard on most memory-based structures, we omit any memory-based registers.

6.2 PVF Prediction via Software Profiling

Our prior work has shown that in the SPEC CPU2006 suite, changes in a program's execution profile are the primary reason for PVF changes due to different input data, and that the PVF of a single basic block is relatively insensitive to input data [20]. This implies that computing PVF based on representative training data will generate accurate runtime PVF estimates. In this work, we adopt this methodology as it requires a programmer to generate PVF estimates just once at compile time.

To generate PVF predictions, we profile each benchmark in the SPEC 2000 suite using the *train* input data set. Based on this training run, we generate an average PVF value for each function in the program's call graph; we then instrument the program to record these values into the PVS registers at runtime.

Figure 12 shows the accuracy of the training runs for the SPEC CFP2000 benchmarks. The only benchmark where input data has a significant effect on a function's PVF is *mesa*, where the *ref* input shows a lower PVF in several short-lifetime functions. This is likely solvable by using a more extensive training data set, but we leave a detailed examination of this effect for future work. Results for the SPEC CINT2000 benchmarks are similar; the only benchmark that shows significant PVF differences is *gcc*, where the *propagate_block* function's PVF increases from 59% in the *train* data to 72% in the *ref* data. Again, we expect this to be solvable with more extensive training or by profiling at a granularity smaller than a function. We note that, despite these prediction inaccuracies, our runtime predictor yields acceptable AVF estimates for both *gcc* and *mesa*.

Loading and storing PVS registers at each function boundary requires two memory operations per function per PVS register. However, we can apply profile-time optimizations to reduce the runtime overhead. First, we only update the PVS registers if the absolute difference in PVF between old and new predictions is more than 3%. Second, we omit predictions for short-lifetime functions to avoid adding overhead to short function executions. Finally, in certain code regions that exhibit a round-robin calling pattern between three or more functions, we use one average prediction for all functions in the call chain. Using our optimizations, we reduce the overhead of PVS updates to an average of 6.2 extra memory operations per million instructions.

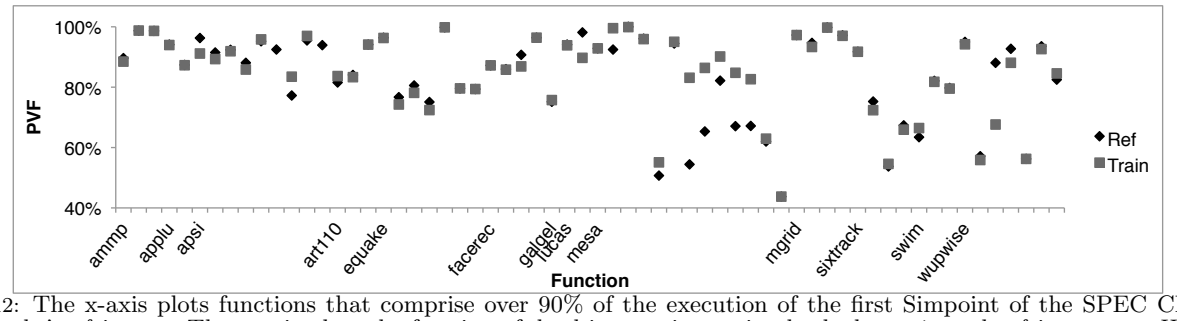


Figure 12: The x-axis plots functions that comprise over 90% of the execution of the first Simpoint of the SPEC CFP2000 benchmarks' *ref* inputs. The y-axis plots the fraction of dead instructions using both the *train* and *ref* input sets. Within a function, the difference in PVF between input sets is less than 4% except for short-lifetime functions in *mesa* and *wupwise*.

6.3 HVF Monitor Unit

To estimate HVF at runtime, we implement an *HVF Monitor Unit* (HMU) for each structure. At a fixed 512-cycle interval, the HMU chooses a random entry within the structure. If the specified entry generates an *HVF trigger* during the next 512-cycle interval, the structure sets a bit within the ROB entry for the corresponding instruction. At commit, a centralized *AVF Calculation Unit* (ACU) reads these added ROB bits. If the bit for a given structure is set, a 3-bit *HVF Counter* is incremented. When the counter overflows, it is reset and the value from the appropriate PVS register is fetched, right-shifted by 3 bits, and added to an *AVF Counter* for the structure. Every 2^{20} cycles, the *AVF Counter* value is divided by the total number of samples (2048) to yield an AVF estimate for the previous interval.

We choose our interval sizes to allow a new AVF prediction every 1 million cycles, similar to previous work [9]. By choosing power-of-two interval sizes, we implement all arithmetic operations as either bit-shifts or integer addition.

The core logic in the HMU and ACU is similar for all structures and can easily be replicated across structures; only the HVF trigger varies from structure to structure. The trigger event in the Issue Queue is an instruction issue, the trigger in the ROB is an instruction commit, and the trigger in the LDB is a read of the load address. Each structure typically has just one HVF trigger, limiting the complexity of monitoring these events.

In terms of hardware overhead, each structure requires the HMU logic and related state; a bit within each ROB entry; a set of basic counters in the ACU; and the logic required to calculate AVF (which is used infrequently and can be shared across all structures). The overhead of our technique is similar to that of other proposed techniques (e.g., [9]).

We expect that using only one HMU per hardware structure will sometimes result in a high estimate of HVF because the estimate will converge to committed instruction occupancy. As discussed in Section 4, this is a loose bound on HVF. The estimate can be improved by treating individual fields within a structure as independent structures with separate HMUs. For example, the IQ's displacement field stores instruction-carried constants such as load/store offsets and arithmetic immediate values, and can be evaluated separately from the rest of the IQ.

6.4 Results

We implement our baseline HVF predictor (one HMU per structure) in the Reorder Buffer, Load Buffer, and Issue Queue. In addition, we implement two additional predictors in the Issue Queue. Predictor *IQ-3* uses 3 HMUs in the IQ: one each for the Displacement field and Branch Control field (which is used only for control instructions), and a third for the remainder of the fields. We identified the displacement and branch control fields as the most likely to benefit from separate measurement. Similarly, predictor *IQ-All* splits the IQ into 8 sub-structures. Obviously, *IQ-All* has high hardware overhead, but we include these results to demonstrate the potential of the technique.

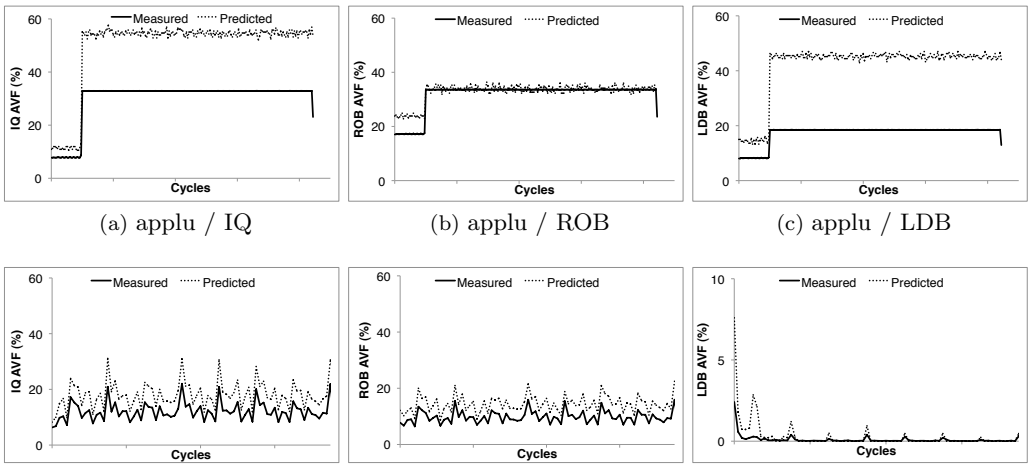
We run our experiments on the first Simpoint of the *ref* input data set for each benchmark. We evaluate our AVF predictor on two relevant metrics: the ability of the predictor to follow time-varying changes in AVF; and the mean absolute error of our AVF estimates to the actual AVF. Actual AVF is measured on a per-field basis as described in Section 4.1.

Figure 13 visually depicts a representative set of results of the baseline predictor. The predictor is highly responsive to time-varying changes in AVF. However, it is obvious that the predictor can have a high error (e.g., the IQ and LDB when running *applu*). Table 2 confirms this result; the mean absolute error (MAE) for many benchmarks is over 10%, which we deem unacceptably high[3]. Our data show that these large errors are due almost entirely to HVF overestimates, and can be corrected by using multiple HMUs per structure.

The results for the *IQ-3* and *IQ-All* predictors are shown in Figure 14 and Table 3. As expected, *IQ-3* substantially improves its HVF estimates over the baseline predictor, which translates into better AVF predictions. *IQ-3* has a 3.6% average MAE across all benchmarks and only one benchmark has an MAE higher than 8%. *IQ-All* further improves the estimate and reduces the average MAE to just 2.7%. Furthermore, only one benchmark has an MAE higher than 6%.

An interesting result is that a few benchmarks have a higher MAE with the *IQ-All* predictor than with *IQ-3*. As discussed in Section 6.2, *gcc*'s training input underestimates the *ref* input's PVF. This underestimate is offset by an HVF overestimate in the *Baseline* and *IQ-3* predictors, but not in *IQ-All*. In *mcf* and *swim*, predictor *IQ-3* slightly overestimates HVF, while predictor *IQ-All* underestimates HVF by a slightly higher magnitude, resulting in a slightly higher MAE. However, the magnitude of the MAE for these benchmarks is still within those reported by prior work [9]. Based on these results, we suggest that predictor *IQ-3* achieves a good balance between overhead and accuracy.

[3]Because AVF is reported as a percent, absolute errors carry a unit of percent. For example, an actual AVF of 10% and an estimated AVF of 15% implies a mean absolute error (MAE) of 5%.



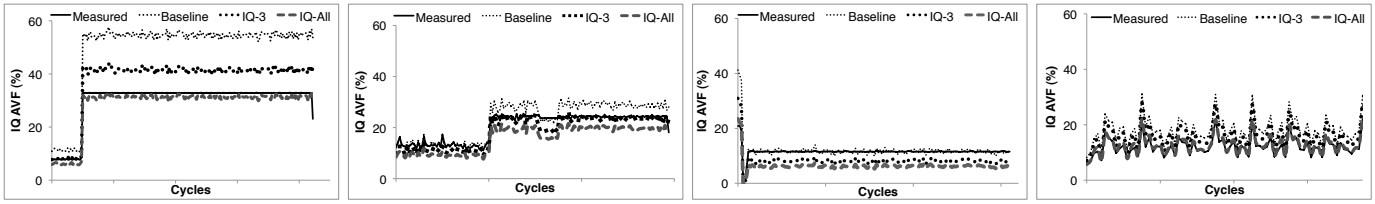
(a) applu / IQ (b) applu / ROB (c) applu / LDB

(d) vortex2 / IQ (e) vortex2 / ROB (f) vortex2 / LDB

Figure 13: A visual depiction of the results from Table 2 for the Issue Queue, Reorder Buffer, and Load Buffer, for the first 100M-instruction Simpoint of *applu* and *vortex2*. Our predictor closely follows the time-varying behavior of AVF in every benchmark. However, the predictor has a high MAE for certain benchmark / structure pairs (e.g., *applu* IQ and LDB).

Benchmark	IQ	ROB	LDB	Benchmark	IQ	ROB	LDB
ammp	18.6	5.7	7.1	lucas	9.2	3.0	0.1
applu	19.6	1.6	24.4	mcf	6.5	5.3	0.1
apsi	5.4	1.9	6.5	mesa	6.2	4.3	5.9
art110	7.1	8.6	6.7	mgrid	3.2	6.2	12.0
crafty	0.4	0.5	0.5	perlbmk_makerand	3.3	4.1	3.4
eon_rushmeier	6.3	3.0	1.0	sixtrack	24.1	1.8	10.4
equake	6.0	4.0	9.5	swim	0.9	2.3	23.9
facerec	10.0	3.3	1.6	twolf	3.8	3.7	2.3
galgel	5.3	6.5	12.9	vortex2	6.0	4.7	0.2
gcc_166	2.9	1.8	4.9	vpr_route	5.3	4.7	0.9
gzip_graphic	2.0	1.8	18.2	wupwise	4.1	3.1	1.7

Table 2: Mean Absolute Errors (MAE) (in %) of the predicted AVF relative to the actual (measured) AVF. Our predictor often has a high MAE because the HMU treats each entry as a single field (similarly to other techniques [9]), which results in HVF overestimates. This highlights the need for more accurate HVF assessment in complex structures such as the IQ.



(a) applu (b) gcc_166 (c) swim (d) vortex2

Figure 14: Predicted and measured AVFs of the *Baseline*, *IQ-3*, and *IQ-All* predictors. Most benchmarks (e.g.,*applu*, *vortex2*) and see a steady improvement of AVF estimates. For a few benchmarks (*gcc_166*, *swim*, and *mcf*), *IQ-3* performs slightly better than *IQ-All* due to PVF or HVF underprediction as discussed in Section 6.4. However, all *IQ-All* AVF predictions still have an absolute error of less than 6%, with an average absolute error of less than 3%.

Benchmark	Baseline	IQ-3	IQ-All	Benchmark	Baseline	IQ-3	IQ-All
ammp	18.6	8.5	2.5	lucas	9.2	4.2	4.4
applu	19.6	7.7	1.5	mcf	6.5	2.1	5.4
apsi	5.4	1.9	0.9	mesa	6.2	3.2	0.7
art110	7.1	6.1	6.6	mgrid	3.2	2.3	3.0
crafty	0.4	0.8	1.5	perlbmk_makerand	3.3	2.0	0.7
eon_rushmeier	6.3	3.4	0.5	sixtrack	24.1	6.8	4.5
equake	6.0	3.8	3.0	swim	0.9	3.5	5.2
facerec	10.0	4.9	4.0	twolf	3.8	1.4	0.7
galgel	5.3	3.0	1.6	vortex2	6.0	3.8	0.7
gcc_166	2.9	1.6	4.6	vpr_route	5.3	2.5	0.7
gzip_graphic	2.0	2.7	3.7	wupwise	4.1	2.6	2.2

Table 3: Mean Absolute Errors (in %) of improved predictors that treat each IQ entry as multiple structures. *Baseline* is the IQ predictor from Table 2, with 1 HMU. *IQ-3* uses 3 HMUs in the IQ, and *IQ-All* uses 8 HMUs, one for each field in the IQ. As expected, *IQ-3* requires less hardware than *IQ-All* but delivers much better AVF estimates than *Baseline*.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the Hardware Vulnerability Factor, one component of the System Vulnerability Stack. We demonstrated that HVF is useful during the hardware design process. HVF can be used to provide greater insight to microarchitectural design decisions and to accelerate AVF simulation, allowing hardware designers to more quickly and accurately assess a processor's vulnerability. Finally, we proposed a technique to enable the runtime monitoring of AVF using pre-calculated PVF and online HVF estimates. The key benefit of this technique is to enable software designers to influence runtime AVF calculations.

In general, the vulnerability stack presents many further opportunities for exploration and optimization. For example, we have not yet explored VM and OS-level vulnerability, although these are important components of many modern computer systems. Furthermore, techniques to dynamically predict PVF may improve the task of runtime AVF estimation. Finally, because the vulnerability stack isolates the system-level effects of a transient fault from its (device-level) causes, we believe that portions of the vulnerability stack may eventually be used to model the effects of other hardware faults.

Overall, we believe that HVF and the System Vulnerability Stack represent an important step forward in our understanding of system vulnerability. The stack can empower a much broader community (e.g., software and OS designers) to more fully participate in the design of fault-tolerant systems, and has the potential to significantly expand the scope of fault tolerance research and enable the design of more robust systems at lower cost.

Acknowledgments

The authors would like to thank Arijit Biswas and Michael D. Powell for their comments on the original manuscript and the anonymous reviewers for their feedback on the final version. This work was supported in part by a Fellowship provided by the Northeastern University Provost's Office.

8. REFERENCES

[1] N. Aggarwal, N. P. Jouppi, P. Ranganathan, J. Smith, and K. Saluja. Reducing overhead for soft error coverage in high availability systems. In *Workshop on System Effects of Logic Soft Errors (SELSE-4)*, Austin, TX, April 2008.

[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 *International Symposium on Computer Architecture (ISCA-32)*, 2005.

[4] A. Biswas, N. Soundararajan, S. S. Mukherjee, and S. Gurumurthi. Quantized AVF: A means of capturing vulnerability variations over small windows of time. In *Workshop on System Effects of Logic Soft Errors (SELSE-5)*, 2009.

[5] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-10)*, 2002.

[6] L. Duan, B. Li, and L. Peng. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *Int'l Symposium on High Performance Computer Architecture (HPCA-15)*, 2009.

[7] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *International Symposium on Computer Architecture (ISCA-32)*, 2005.

[8] X. Li, S. Adve, P. Bose, and J. Rivers. Softarch: an architecture-level tool for modeling and analyzing soft errors. *International Conference on Dependable Systems and Networks (DSN '05)*, 2005.

[9] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *International Symposium on Computer Architecture (ISCA-35)*, 2008.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI '05)*, 2005.

[11] 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 *International Symposium on Microarchitecture (MICRO-36)*, 2003.

[12] 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.

[13] V. K. Reddy, E. Rotenberg, and S. Parthasarathy. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. In *Int'l Conf. on Arch. Support for Prog. Langs. and Op. Sys. (ASPLOS-12)*, 2006.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization (CGO '05)*, 2005.

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

[16] N. Seifert and N. Tam. Timing vulnerability factors of sequentials. *Device and Materials Reliability, IEEE Transactions on*, 4(3):516–522, Sept. 2004.

[17] J. W. Sheaffer, D. P. Luebke, and K. Skadron. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In *SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '06)*, 2006.

[18] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[19] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In *International Symposium on Computer Architecture (ISCA-34)*, 2007.

[20] V. Sridharan and D. R. Kaeli. The effect of input data on program vulnerability. In *Workshop on System Effects of Logic Soft Errors (SELSE-5)*, 2009.

[21] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *International Symposium on High Performance Computer Architecture (HPCA-15)*, 2009.

[22] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin. Efficient fault tolerance in multi-media applications through selective instruction replication. In *Workshop on Radiation Effects and Fault Tolerance (WREFT '08)*, 2008.

[23] 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.

[24] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO '07)*, 2007.

[25] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *International Symposium on Computer Architecture (ISCA-31)*, 2004.