INTRODUCING ABSTRACTION TO VULNERABILITY ANALYSIS

A Dissertation Presented
by
Vilas Keshav Sridharan
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of
Computer Engineering

Northeastern University
Boston, Massachusetts

March 2010
© Copyright 2010 by Vilas Keshav Sridharan
All Rights Reserved
Abstract

Tolerance to the effects of transient faults is now 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. One step in this analysis 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 does not express vulnerability in terms of the system stack (hardware, virtual machine, user program, etc.), limiting the amount of insight that can be generated. To remedy this, we propose the System Vulnerability Stack, a framework to calculate a vulnerability factor at each level of the system stack. These vulnerability factors can be used individually or combined to generate a system-level AVF measurement.

In this thesis, we first establish a rigorous theoretical and mathematical basis for the vulnerability stack, and introduce the simulation framework through which the individual vulnerability factors can be measured. We then present several methods by which the vulnerability stack can influence system design. We show that the Program Vulnerability Factor can be used during the software design cycle to increase the
robustness of a software program. We also show that the *Hardware Vulnerability Factor* can improve the hardware design cycle, improving the robustness of hardware as well as allowing better assessment of chip failure rates at design time. Finally, we demonstrate that the concepts behind stack can be applied at runtime to improve online monitoring of system vulnerability.
Acknowledgements

First of all, I would like to thank my advisor, Prof. David Kaeli, for his help, support, and advice throughout this process. I would also like to acknowledge Dr. Hossein Asadi and Prof. Mehdi Tahoori, our collaborators on my Master’s Thesis and the precursor to this work. I would also like to thank Arijit Biswas and Shubu Mukherjee at Intel Corporation, and Dean Liberty at AMD, Inc. My time at both those companies contributed significantly to my understanding and development of this thesis.

Finally, I would like to thank my wife Kirstin for her love, support, patience, and understanding while I have been in graduate school.
Contents

Abstract iii

Acknowledgements v

1 Introduction 1

1.1 A Brief History of Fault Tolerance ......................... 2
1.2 The Effect of Current Trends on Fault Tolerance ............ 5
1.3 The Importance of Vulnerability Analysis .................... 7
1.4 Reliability Techniques In Modern Computer Systems ........ 8
1.5 Scope and Contributions of This Thesis ..................... 10
1.6 Organization of This Thesis ............................... 11

2 Background and Related Work 13

2.1 Terminology .............................................. 13
2.2 Classes of Fault ........................................ 15
2.3 Fault Masking and Vulnerability Factors .................... 17
2.4 Techniques to Measure Vulnerability Factors ............... 18
   2.4.1 Fault Injection .................................... 18
   2.4.2 Probabilistic Modeling ............................ 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>Related Work</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Static Analysis</td>
<td>22</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Dynamic Techniques</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Architectural Vulnerability Factor</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Using AVF to Improve System Design</td>
<td>26</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Exploring Potential Limitations of AVF</td>
<td>26</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Estimating Vulnerability at Runtime</td>
<td>27</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Dependence on Application Behavior</td>
<td>29</td>
</tr>
<tr>
<td>2.8</td>
<td>Limitations and Opportunities</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>Benefits of Abstraction</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>The System Vulnerability Stack</td>
<td>35</td>
</tr>
<tr>
<td>3.1</td>
<td>Definitions and Concepts</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Individual Vulnerability Factors</td>
<td>39</td>
</tr>
<tr>
<td>3.2.1</td>
<td>The Program Vulnerability Factor</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Hardware Vulnerability Factor</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Computing System Vulnerability</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Multi-Exposure Bits</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Multiple Software Layers</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Computing System Soft Error Rates</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Summary</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>Simulation Methodology</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Background: ACE Analysis</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Adapting ACE Analysis to the Vulnerability Stack</td>
<td>55</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Changes to All Simulation Passes</td>
<td>55</td>
</tr>
</tbody>
</table>
6 Using the Stack for Hardware Design 102

6.1 Analysis of the Hardware Vulnerability Factor . . . . . . . . . . . . . 103
  6.1.1 Using HVF for Microarchitectural Exploration . . . . . . . . 105
  6.1.2 Using Occupancy to Approximate HVF . . . . . . . . . . . . 108
6.2 Bounding AVF During Hardware Design . . . . . . . . . . . . . . . . 110
  6.2.1 Capturing PVF Traces . . . . . . . . . . . . . . . . . . . . . . 113
  6.2.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

7 Using the Stack for Runtime Analysis 121

7.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
7.2 PVF Prediction via Software Profiling . . . . . . . . . . . . . . . . 123
7.3 HVF Monitor Unit . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
7.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

8 Summary and Conclusions 132

8.1 Summary of Research . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
  8.1.1 The System Vulnerability Stack . . . . . . . . . . . . . . . . . 133
  8.1.2 Vulnerability Stack Simulation Framework . . . . . . . . . . 133
  8.1.3 The Program Vulnerability Factor . . . . . . . . . . . . . . . . 135
  8.1.4 The Input Dependence of Program Vulnerability . . . . . . . . 136
  8.1.5 The Hardware Vulnerability Factor . . . . . . . . . . . . . . . 137
  8.1.6 PVF Traces . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
  8.1.7 The Program Vulnerability State and HVF Monitor Unit . . . 138
8.2 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
  8.2.1 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
List of Figures

1.1 Soft error rate over technology generations, from Baumann [1]. The soft error rate of a bit is predicted to remain roughly constant, but the soft error rate of a system is predicted to increase with Moore’s Law scaling. ................................................................. 6

1.2 AVF of several hardware structures in a modern microprocessor, from Sridharan [2]. The fraction of faults that affect correctness in a modern microprocessor is often below 15% on average. If a designer assumes that all faults affect correct operation, he/she will overestimate the effective error rate, and risks substantial overdesign of the processor. . 9

1.3 Results of a vulnerability analysis of the instruction, data, and L2 caches, from Asadi [3]. Vulnerability analysis helps identify the most important structures to protect. In many cases, the structure that sees the most errors is not obvious, since the error rate depends on the usage patterns of each structure. ................................. 9
2.1 Dependence of vulnerability on workload and microarchitecture. The vulnerability of a physical register file when running three programs on one microarchitecture and one program (bzip2) on three microarchitectures. This figure clearly shows that the vulnerability of the register file depends on both workload and hardware configuration. 30

3.1 The System Vulnerability Stack. The vulnerability stack calculates a vulnerability factor at every layer of the system. Each bit in a system is assigned a vulnerability at every layer to which it is visible. If a bit is vulnerable at every layer, it is vulnerable to the system (i.e., its AVF is 1). 36

3.2 Computing system vulnerability with a single software layer. Physical register P1 is mapped to architectural registers R1 and R2 over several 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. 42

3.3 Multi-exposure bits. A single microarchitecture-visible fault in a multi-exposure bit (e.g., 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 bit requires evaluating the impact of all the program-visible faults simultaneously. 44
3.4 Computing system vulnerability with multiple software layers. The behavior of architectural register R1 on a system with a virtual machine and user program. VM instructions are shown in black; user program instructions are shown in gray. R1 is mapped to program register V1 during instructions 1-5 and 14-15; to program register V2 during instructions 7-9; and not mapped during instructions 6 and 10-13. The vulnerability of R1 can be calculated using the VMVF and PVF during each instruction.

4.1 A comparison of Fault Injection to ACE Analysis. Fault injection requires many simulation passes to generate an AVF estimate. ACE Analysis requires only one pass, and thus requires many fewer simulation cycles than fault injection.

4.2 Simplified source code example from our M5 simulation infrastructure. Every structure is indexed with three indices. When an entry in the structure is read by the CPU, a read event is added to the corresponding object’s event queue. The event is also added to a structure-wide list of events waiting for analysis.

4.3 Simplified dynamic-dead analysis code from our Pin simulation infrastructure. Once an instruction commits, it is inserted onto the post-commit instQ queue. We then check its liveness and loop through its source and destination operands, increasing the consumer count on source operands and decreasing the active destination count on destination operands. We then call traverse() to determine if the producing instruction is dead.
5.1 *The PVF of the architectural register file.* Architectural Register File PVF for `bzip2_source`, `mgrid`, and `equake`.  

5.2 *The PVF of the architectural integer ALU.* Architectural Integer ALU PVF for `bzip2_source`, `mgrid`, and `equake`.  

5.3 *Comparison of AVF and PVF for the register file.* Physical Register File AVF (in gray) and Architectural Register File PVF (in black) for `bzip2_program`, `ammp`, and `sixtrack`.  

5.4 *Comparison of register AVF and PVF for benchmarks with low correlation.* Register File AVF (in gray) and PVF (in black) for `mesa`, `perlbmk_makerand`, and `vpr_route`. The low correlation for these benchmarks is a result of minor fluctuations around a constant value.  

5.5 *Comparison of AVF and PVF for the ALU.* Integer ALU AVF (in gray) and PVF (in black) for `bzip2_program`, `galgel`, and `sixtrack`.  

5.6 *Performance figures for several benchmarks.* The CPI for `bzip2_program`, `galgel`, and `sixtrack`. CPI variation is one contributor to low correlation between AVF and PVF in the ALU.  

5.7 *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.  

5.8 *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.
5.9 *The effect of loop optimizations on PVF, Part 1.* A loop from *bzip2* compiled by *gcc* with optimizations: (a) *-O3* and (b) *-O3 -funroll-loops -fno-spec-sched*. In this example, unrolling the loop results in a slight decrease in PVF (25.43% versus 26.56% for the original loop). However, the unrolled loop executes 15% faster, resulting in a 16% lower cumulative vulnerability.

5.10 *The effect of loop optimizations on PVF, Part 2.* A loop from *bzip2* compiled by *gcc* with optimizations: (a) *-O3 -funroll-loops -fno-spec-sched* and (b) *-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.

5.11 *Results when disabling speculative scheduling.* Percent change in PVF, runtime, and cumulative vulnerability when disabling speculative scheduling.

5.12 *Results when disabling instruction scheduling.* Percent change in PVF, runtime, and cumulative vulnerability when disabling instruction scheduling.

5.13 *Changes in PVF with different input data, plotted by time.* Integer Register File PVF for *400.perlbench* using input files *diffmail* and *splitmail* from the *train* data set.

5.14 *Changes in PVF with different input data, plotted by time.* Integer Register File PVF for *450.soplex* using input files *train* and *pds-20* from the *train* data set.
5.15 Changes in PVF with input data, plotted by code trace. Per-trace PVF for 400.perlbench and 450.soplex using multiple input files from their train data set. ......................................................... 95

5.16 Mean absolute error of our prediction algorithm. The absolute error in PVF of our predictor. Overall, we see a mean absolute error of under 0.06 for all benchmarks except 450.soplex. .................................................. 96

5.17 Real versus predictor PVF over time. Real versus predicted PVF for gcc_166, gcc_g23, and soplex_ref. .......................................................... 99

5.18 Comparison of input files for 450.soplex. The train.mps file covers more of the code traces touched by the ref.mps input file than does pds-20.mps; therefore, our predictor would perform better when trained with train.mps. ......................................................... 100

6.1 The AVF and HVF of the register file for equake and mgrid. In equake, a combination of pipeline stalls (high HVF) and dynamically-dead instructions (low PVF) cause its HVF to be a loose bound on AVF. This is not generally true; for example, the HVF of mgrid is a tight bound on AVF in certain places. It is impossible for a hardware designer to a priori predict this workload-specific behavior; a designer can only guarantee that AVF is less than HVF. Therefore, it is important to understand the causes of high HVF in a hardware structure. ......... 103

6.2 The HVF of the Reorder Buffer (ROB), Issue Queue (IQ), Load Buffer (LDB), and Physical Register File (PRF) for ammp and equake. A structure’s HVF varies substantially across benchmarks, increasing when the structure is full of correct-path instructions, and decreasing when the structure is empty or contains many wrong-path instructions. 106
6.2 *Continued from the previous page.* The HVF of the ROB, IQ, LDB, and PRF for *mcf*.

6.3 *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.

6.4 *The relationship between HVF and committed instruction occupancy across benchmarks for the ROB and IQ.* 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 0.98 across all benchmarks. This indicates that committed instruction occupancy is a good but conservative heuristic for HVF.

6.5 *Simulation speedup with PVF tracing.* Using PVF traces led to an approximately 2x reduction in simulation time. This is primarily due to the ability to perform separate PVF calculations using architecture-only simulation.

6.6 *The size of PVF traces.* PVF traces typically required between 0.2 and 4 bytes per instruction. This leads to trace sizes of between 20-400MB per 100M instructions. The trace size increases with the number of dead instructions in the benchmark.
6.7 *Snippets from a uncompressed PVF trace.* Each line represents one instruction and lists: the unique ID of the originating instruction; whether the event was a write (w) or a read (r); whether the event was to a register (R) or memory (M); the register index or memory address; and the liveness of the bytes in the event. Multiple dead events from one instruction are separated by a “:”.  

6.8 *The granularity of PVF analysis versus trace size.* PVF traces can be used to tighten the AVF bound during hardware design. Reducing the accuracy of the PVF analysis results in many dead events incorrectly being marked as live. This is accompanied by a sometimes significant reduction in trace size. We show results for PVF analysis at different granularity: word (2 byte), longword (4 byte), and quadword (8 byte), normalized to byte granularity.  

6.9 *Sources of unACEness in PVF traces.* There are many sources of unACE events that can be captured in PVF traces. On average, 57% of unACE events are due to dead instructions (static and dynamic), while 33% and 9% of unACE events are due to first-level and transitive logic masking, respectively. Because PVF traces are generated offline, we can add expensive features such as detection of transitive logic masking with no impact on simulation speed during hardware design.  

6.10 *The effect of incorporating PVF traces into HVF simulation.* Incorporating PVF traces tightens the AVF bound by 60% in the register file, 25% in the Issue Queue, 30% in the Store Buffer, 30% in the Load Buffer, and 24% in the Reorder Buffer.
7.1 **Results of our PVF prediction algorithm.** The x-axis plots functions that comprise over 90% of the execution of the first Simpoint of the ref input of the SPEC CFP2000 benchmarks. The y-axis plots the fraction of dead instructions using both the train and ref input sets. For a given function, the difference in PVF between input sets is generally less than 0.04, with the exception of several short functions in mesa and one in wupwise.

7.2 **Results of our AVF predictor.** A visual depiction of the results from Table 7.1 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).

7.3 **Predicted and measured AVFs of the Baseline, IQ-3, and IQ-All predictors.** Most benchmarks resemble applu and see a steady improvement of AVF estimates. For a few benchmarks such as gcc_166, swim, and mcf, IQ-All gives a slightly worse estimate than IQ-3, primarily due to either PVF or HVF underprediction as discussed in Section 7.4. However, the results are still within acceptable tolerances.
List of Tables

4.1 Simulated Machine Parameters. Baseline machine parameters used for experiments in this chapter. ................................................. 61

5.1 Pearson correlation coefficients between register PVF and AVF. Correlation between Architectural Register File PVF and Physical Integer Register File AVF. The correlation is high for most benchmarks. . . . 73

5.2 Register file AVF and PVF correlation across microarchitectures. The PVF-AVF correlation remains strong across these changes. Results for other benchmarks are similar. ...................................................... 76

5.3 Pearson correlation coefficients for ALU PVF and AVF (Column 2) and PVF and AVF*CPI (Column 3). The PVF of the Integer ALU only partially explains its AVF variation, but the PVF and CPI together explain all the AVF variation. ................................. 80

5.4 Mean absolute difference in PVF for traces common to both input sets. The PVF difference between code traces seen by both inputs is small. 97

5.5 Training input used for benchmarks with multiple inputs in the train data set. For benchmarks with multiple inputs in the train data set, we chose one input on which to train our predictor. ................. 98
7.1 Mean Absolute Errors (MAE) of the predicted AVF relative to the actual (measured) AVF. Our predictor often has a high MAE because the HVF Monitor Unit treats each entry as a single field, resulting in HVF overestimates. This is similar to the behavior of other techniques (e.g., [4]), and points to the need for a more accurate HVF assessment in complex structures such as the IQ. . . . . . . . . . . . . . . . . . . . . . . . . . . 128

7.2 Mean Absolute Errors of improved predictors that treat the IQ as multiple structures. Baseline is the IQ predictor from Table 7.1. IQ-3 treats the two most vulnerable fields in the IQ as separate structures, and IQ-All treats all 8 fields in the IQ as separate structures. Overall, IQ-3 requires less hardware than IQ-All but, as expected, delivers substantially better AVF estimates than the baseline predictor. . . . 129
Chapter 1

Introduction

Reliability is a first-class design constraint for systems from high-end mainframes to commodity PCs due to the effects of a variety of fault classes (e.g., transients, intermittents) [5] [6]. To meet reliability goals, microprocessor vendors typically set a failure rate target for each design and perform significant pre-silicon analysis to ensure a design adheres to this target [7]. The ability to predict a system’s failure rate from these faults is crucial to understanding the system’s performance relative to its reliability goals; therefore accurate fault modeling is a necessity for modern computer systems. One important aspect of fault modeling is measuring the effect of fault masking; masked faults do not affect correct system operation and do not impact a system’s failure rate. Although there are many classes of fault in modern computer systems, the class of faults termed transients occur quite often in the field; thus, prior techniques to quantify fault masking have typically used a transient fault as the baseline fault model [7] [8] [3].

In this thesis, we present a new method to quantify the level of fault masking in a system called the System Vulnerability Stack. While the Vulnerability Stack builds on
prior work in this field, it introduces significant enhancements to currently-existing techniques. The vulnerability stack achieves this by focusing on fault effects at architected interfaces between system layers, such as the Instruction Set Architecture (ISA) boundary between hardware and software. The vulnerability stack computes fault masking separately within each system layer by determining whether a fault to propagate to a layer’s interface. This allows for many benefits and opportunities that are not available when using previous techniques.

In this chapter, we present the reader with an introduction to some basic fault tolerance concepts. This includes a very brief overview of the scope of reliability research and its impact on modern systems, and an introduction to the classes of fault that are common in modern computer systems. We discuss in detail the concept of fault masking, as this concept is directly relevant to our thesis. Finally, we discuss the scope and contributions of our work, and give an overview of the remainder of this thesis.

### 1.1 A Brief History of Fault Tolerance

*Fault tolerance* is broadly defined as the ability of a system to operate in the presence of a failure in one or more of its components. Fault tolerance has a long history in modern computer systems; for example, researchers at IBM have been addressing faults in mainframe systems for almost half a century [9]. As a result, mainframe systems have led in the development and implementation of many reliability solutions [10].

There are many different types of faults that occur in computer systems. Historically, permanent (hard) faults have been the most commonly-occurring failure
events in terrestrial applications. Typically, these events have a failure rate on the order of 1 - 500 FIT per chip [11]. (1 FIT is one failure in one billion device hours.) Although transient faults due to high-energy particle strikes were a documented phenomenon [12], for many years the error rate at sea level due to transient faults was substantially lower than that of permanent faults. Thus, transient faults were of less concern than permanent faults in terrestrial applications.

In airborne and space applications, on the other hand, the rate of high-energy particle strikes is many orders of magnitude higher than at ground level [11]; therefore, transient faults have been a focus in these applications for decades [13]. These markets, however, are less performance and cost-sensitive than typical consumer markets. Therefore, solutions in this space often involved radiation-hardening of device cells [13] or Triple-Modular Redundancy (TMR) [14]. These techniques guarantee tolerance to virtually any fault. However, these techniques are also extremely expensive to implement and thus are only attractive in applications with high error rate (e.g., space-borne systems), or where the cost of an error is extremely high (e.g., nuclear simulations).

For many years, outside of mainframe and space-borne systems, tolerance to hardware faults was not a priority. This changed in the late 1990s, however, as a combination of Moore’s Law scaling and decreased transistor dimensions resulted in a significant increase in the rate of transient faults at ground level [5] [15] [16]. Mainstream server and consumer system vendors experienced several well-documented hardware failures in the field that were directly attributed to transient faults [5] [15]. Therefore, processor and system vendors could no longer afford to ignore the threat posed by these faults.

The result of this change was two-fold. First, the thrust of fault tolerance research
in terrestrial applications expanded to include the effects of transient faults. This is important because transient faults have different implications for system design than permanent faults. For instance, the presence of a permanent fault typically indicates an unstable or failed hardware component. However, the presence of a transient fault does not indicate a hardware component failure; once the fault is cleared, the hardware will continue to function normally. This affects the handling of fault detection and correction as well as the potential design of system recovery techniques.

The second major change is that the scope of fault tolerance research expanded to include systems such as desktop personal computers. These systems typically have significantly lower reliability requirements than mainframe, airborne, or spaceborne systems. Furthermore, these systems are extremely cost- and performance-sensitive; the cost of an error is much lower than in mainframe or spaceborne applications; and the rate of errors from other sources (e.g., software) is much higher. Hardware reliability features always come with a cost (in area, performance, power, or a combination of all three). Therefore, system vendors are willing to add features that improve hardware reliability only when absolutely necessary in order to meet a failure rate target.

The combination of these factors has led to the increasing importance of accurate assessment of a system’s true failure rate from transient faults. A system’s performance relative to its failure rate target will dictate the quantity and scope of fault tolerance features that a vendor must add to a system in order to achieve an acceptable product. A primary component of assessing a system’s failure rate is vulnerability analysis: the process of accurately determining a system’s error rate based on the incident transient fault rate [7] [17].
Having provided some historical context for our work, we examine in the next section whether fault tolerance, and especially tolerance to transient faults, will remain an important design constraint in future systems.

1.2 The Effect of Current Trends on Fault Tolerance

In the previous section, we presented a brief history of fault tolerance, and described the historical development of vulnerability analysis. A reasonable question for the reader to ask is whether this task will continue to be of import. In particular, will transient faults continue to be a problem in future computer systems? We address that question in this section.

Current trends in system design indicate that transient faults will be an important consideration in mainstream devices for years to come. Although the transient fault rate per device (e.g., transistor, memory cell) is predicted to remain roughly constant over several technology generations, increasing transistor counts (via Moore’s law scaling) mean that the per-system fault rate will continue to increase despite this trend. This trend is captured in Figure 1.1 [1], which shows that the per-system error rate approximately doubles with every technology generation. This doubling is due directly to the increased transistor count made possible by the decrease in feature size of each new technology generation. Although this chart is now several years old, the same trends are predicted to continue for the foreseeable future.

Another factor in the increasing importance of fault analysis is the increased underlying complexity of future microprocessors (in the form of higher core counts, heterogeneous cores, and greater integration of system components). This increased
complexity means that the task of fault tolerance will encompass more underlying structures and behaviors, and thus many different effects. In the face of this trend, it is imperative to have a robust fault framework that is capable of analyzing the complex behaviors and interactions that result.

Finally, the proliferation of semiconductor-based devices in everyday life means that the number of transistors per user (and therefore, the number of transient faults per user) will also continue to increase for the foreseeable future. This implies that the effective error rate per device must decrease in order to maintain a constant per-user error rate. Again, robust analysis tools and methodologies are required in order to meet this increasingly difficult challenge.

We have now established that tolerance to transient faults will be an increasingly important design constraint for future microprocessors. In the next section, we examine the role that vulnerability analysis plays in improving a system’s tolerance to transient faults.
1.3 The Importance of Vulnerability Analysis

As discussed in Section 1.1, vulnerability analysis determines the component(s) in a system that contribute most to the transient fault rate. This allows designers to focus their reliability efforts on those structures. Furthermore, accurate vulnerability analysis can prevent designers from over- or under-designing reliability features into systems, saving time and effort while ensuring that the system meets reliability targets.

Two examples serve to illustrate the importance of vulnerability analysis. Figure 1.2 shows the results of vulnerability analysis performed on several hardware structures in a high-performance microprocessor model [2]. On average, fewer than 20% of faults actually resulted in incorrect system operation across all structures. In the Store Buffer (SB), only 4% of faults affected the program’s output. This result shows the power of vulnerability analysis. The vulnerability analysis allows us to reduce our error rate estimate for these structures by a factor of 5-10x. If performed early enough in the design cycle, this can have a significant influence on the design of reliability features for each of these hardware structures.

A second example of the importance of vulnerability analysis is given in Figure 1.3 [3]. This figure shows the relative error rate of three hardware structures: an L2 cache tag array, an L1 instruction cache, and an L1 data cache. The L1 data cache is approximately 4x larger than the L2 tag array (64kB versus approximately 15kB for the L2 tag). Transient faults are uniformly distributed in space; therefore, the L1 data cache will incur 4x more faults than the L2 tag array. A naive analysis would infer, therefore, adding redundancy to the L1 data cache will provide a greater reduction in error rate than adding redundancy to the L2 tag array. The figure shows, however, that the L2 tag array actually results in 40% more errors than the L1 data
cache. Therefore, redundancy in the L2 tag array will provide a greater reduction in error rate than redundancy in the L1 data cache. Moreover, since the cost of redundancy features typically scales with structure size, redundancy in the 15kB L2 tag array will yield significantly more "bang for the buck" than redundancy in the 64kB L1 data cache.

The preceding examples demonstrate the importance of vulnerability analysis in making proper design choices for a system. As systems grow in complexity, improvements in the accuracy and simplicity of vulnerability analysis will be crucial to meeting the increased fault tolerance demands placed on designers. In the next section, we present a brief overview of the scope and extent of fault tolerance techniques in modern computer systems.

1.4 Reliability Techniques In Modern Computer Systems

Designing a modern computer system requires hundreds, or thousands, of engineers. This includes hardware architects and designers, operating system designers, software designers, layout and device engineers, and many more. All of these engineers work together to contribute a small portion of the overall system design. This level of coordination and collaboration would be impossible without abstractions such as the Instruction Set Architecture (ISA), which separates the architecture (i.e., the specified behavior) from the implementation of that behavior. The ISA serves as an "contract" between hardware and software designers, allowing different components to be designed independently. Prior to the development of the ISA, entire systems had to be developed in unison; and programs for one system could not be used on
Figure 1.2: AVF of several hardware structures in a modern microprocessor, from Sridharan [2]. The fraction of faults that affect correctness in a modern microprocessor is often below 15% on average. If a designer assumes that all faults affect correct operation, he/she will overestimate the effective error rate, and risks substantial overdesign of the processor.

Figure 1.3: Results of a vulnerability analysis of the instruction, data, and L2 caches, from Asadi [3]. Vulnerability analysis helps identify the most important structures to protect. In many cases, the structure that sees the most errors is not obvious, since the error rate depends on the usage patterns of each structure.
CHAPTER 1. INTRODUCTION

another system.

Many of these designers are also responsible for adding fault tolerance features to the system design. These features can be added at every layer of the system stack. For example, circuit designers can use fault-tolerant latches and flip-flops (e.g., BISER [18] or RAZOR [19]) or can add radiation-hardened (rad-hard) cells to critical hardware components [20] [13]. Hardware designers and microarchitects can incorporate Error Correcting Codes (ECC) such as Hamming codes [21], cache scrubbing [22], or structure flushing [7] into their designs. Operating system designers can implement policies to handle uncorrectable errors; software designers can add exception handling code or fault tolerance techniques (e.g., SWIFT [23]) to their programs. Each of these techniques works independently at every layer of the system stack due to the abstractions, such as the ISA, that are present in modern computers. Together, they all contribute towards the overall goal of a fault-tolerant system.

1.5 Scope and Contributions of This Thesis

As discussed in the previous section, reliability is independently implementable at every system level due to existing abstractions (ISA, etc). However, current vulnerability analysis techniques exploit very little of the abstraction present at higher levels of the system stack. This prevents vulnerability analysis from being more broadly adopted in the design of complex systems. The goal of this thesis is to expand the reach of vulnerability analysis to all levels of the system stack.

The key contribution of this thesis is the introduction of the System Vulnerability Stack. The vulnerability stack allows architects to exploit the abstractions present in a modern computer system to enhance the task of vulnerability
analysis. This allows architects a much more robust view of system behavior in the presence of faults. The benefits of this technique are numerous and will be detailed throughout this thesis.

Other contributions of this thesis include:

- the development and implementation of a simulator framework to evaluate the Vulnerability Stack;

- evaluation of the use of the stack to enhance the software design process, including an analysis of the Program Vulnerability Factor (PVF) and an examination of its dependence on input data;

- evaluation of the use of the stack to enhance the hardware design process, including an analysis of the Hardware Vulnerability Factor (HVF), and an examination of PVF traces as a method to improve soft error rate estimation during hardware design; and

- analysis and evaluation of use of the stack at runtime, including a proposal to estimate system vulnerability via two structures we call the Program Vulnerability State and an HVF Monitor Unit.

1.6 Organization of This Thesis

The remainder of this thesis is organized as follows. Chapter 2 provides the reader with necessary background information and a survey of existing methods to measure system vulnerability. These techniques are essential to understand as the Vulnerability Stack builds on this work and addresses several of the limitations. Then, Chapter 3 introduces the theory and mathematics behind the System Vulnerability
Stack. We develop in detail both terminology to guide system designers when using the stack, as well as methods to calculate vulnerability. Chapter 4 gives some background on simulation techniques to assess vulnerability, and then describes in detail our simulation methodology and framework, upon which all further results are based. Chapter 5 evaluates the use of the stack during the software design process, and performs a detailed analysis of the *Program Vulnerability Factor*. Chapter 6 describes the use of the stack during the hardware design process, including an analysis of the *Hardware Vulnerability Factor* as well as introducing the concept of *PVF traces*. Chapter 7 presents our proposal to harness the stack at system runtime, using the *Program Vulnerability State* and *HVF Monitor Unit* to estimate system vulnerability. Finally, Chapter 8 provides a summary of our work and contributions and discusses some limitations and future potential directions for our work.
Chapter 2

Background and Related Work

In this chapter, we describe current and proposed techniques to quantify system-level fault masking. We start by presenting the reader with terminology to which we will adhere for the remainder of this proposal. We then describe the two basic paradigms of fault modeling, \textit{fault injection} and \textit{probabilistic modeling}. Each class of technique has several variations which we describe in detail. The System Vulnerability Stack is a probabilistic fault model that most closely resembles the \textit{Architectural Vulnerability Factor}; therefore, we also present a survey of related work on AVF. At the conclusion of this chapter, the reader should possess a broad understanding of fault modeling in modern computer systems, and a detailed understanding of how to assess fault masking using Architectural Vulnerability Factors.

2.1 Terminology

We begin by presenting a set of definitions to which we adhere throughout the remainder of this thesis. Comprehending the distinction in these terms is crucial to understanding the remainder of this work.
We make a distinction throughout this work between a fault, an error, and a failure. We define a fault as the result of a raw event such as a single-event upset or an intermittent failure of marginal hardware. An error is one possible result of a fault, and is an event that causes a decrease in a system’s fault tolerance. Finally, a system failure is an event that causes the system to incorrectly process a task, or to stop responding to requests, and is one possible result of an error. As we will see, not all faults lead to an error; and not all errors lead to failures. For example, a fault might lead to an error in a system’s register file, regardless of whether it actually causes a system failure. If the register file was part of a lockstepped core-pair, for example, the system would detect and recover from this error; however, the system might not be able to tolerate a fault in the second register file, resulting in decreased fault tolerance during that time period.

Errors are often classified as detected or undetected. An undetected error might result in a Silent Data Corruption (SDC) failure. An SDC is a corruption of system state that is unreported to either the system or the program. This is generally regarded as the most severe failure that can result from an error. A detected error can be further classified as a Corrected Error (CE) or a Detected Uncorrected Error (DUE) [24]. Corrected Errors are errors from which recovery to normal system operation is possible, either by hardware or software. Detected Uncorrected Errors are errors that are discovered and reported, but from which recovery is not possible. These errors typically cause a program or system to crash.

The raw fault rate of a system to a particular class of fault is the number of faults of that type per unit time. This is typically expressed in units of Failures-In-Time (FIT); one FIT is equal to one failure in a billion hours. The error rate of a system is defined as the number of errors per unit time, also expressed in FIT. Since not all
faults result in system errors, we define system vulnerability as the fraction of faults in a system that become errors. Therefore, the error rate of a system from a particular class of fault can be expressed as the product of the raw fault rate and the system vulnerability:

\[
\text{Error Rate} = \text{Raw Fault Rate} \times \text{System Vulnerability}
\]  

(2.1)

Finally, we draw a distinction between two metrics to measure overall fault tolerance: reliability and availability. A system’s reliability can be defined as the fraction of initiated jobs that complete correctly. A system’s availability is the fraction of (wall-clock) time that a system is able to initiate jobs. Both metrics are a function of the system’s error rate and its error handling infrastructure. The relative importance of these metrics differs based on the usage model of the system; for some systems (e.g., servers with many small tasks), maintaining high availability can be more important than correctly completing a particular individual job.

## 2.2 Classes of Fault

Fault tolerance is a wide field that encompasses many different disciplines and areas within computer architecture. It includes such varied tasks as: determining the device-level details of a transistor that impact its susceptibility to high-energy particles such as neutrons; architecting software that can reduce the severity of an error; and designing a user interface to minimize the chance of mis-configuration by an operator. Broadly speaking, fault tolerance must be considered at every level of system design, from the device-level through the design of every piece of hardware and software within a system.
The focus of this thesis is on quantifying the fraction of hardware faults (from effects in devices, transistors, and microarchitectures) that result in errors. Therefore, this section presents a brief description of several classes of hardware fault found in modern computer systems.

A hard fault results in a permanent failure in the devices in question. As a result, this device generally becomes unusable for future system operation. For example, a memory cell that becomes stuck at low logic level (a stuck-at-0 fault) will always return a zero value regardless of the value stored in that location. Hard faults are often caused by device lifetime failures such as wearout; these happen as a device reaches the end of its useful life. The MTTF of devices to wearout in modern technology processes typically exceeds the useful life of the system; therefore, these failures are generally rare in practice [1].

In contrast, a transient fault does not result in permanent device failure, but rather in the corruption of data currently stored in the device; the device will still be usable to store future data. These faults most often arise from environmental sources such as an impact from a high-energy neutron (an effect of the interaction of cosmic rays with the atmosphere). For example, a particle that strikes a sensitive region of an SRAM cell can accumulates enough charge to flip the value stored in the cell; this will result in a transient fault in the SRAM cell. A key characteristic of transient faults is that, due to the random nature of the underlying events (e.g., particle strikes), each fault event is independent; that is, information about one transient fault does not provide any information about future transient fault events.

Finally, an intermittent fault is a fault that does not cause permanent damage, and typically results from internal conditions such as manufacturing remnants or voltage droop. As such, unlike transient errors, intermittent faults are usually correlated: an
intermittent fault in a bit indicates that the same bit (or nearby bits, depending on the details of the fault) is likely to experience another fault.

2.3 Fault Masking and Vulnerability Factors

Fault masking occurs when a fault in the system does not lead to a user-visible error [25]. For instance, a fault might occur in invalid data; it might occur in valid data that is never consumed; or it might be corrected by a redundancy technique such as Triple-Modular Redundancy (TMR). In any of these cases, the underlying fault will not affect correct operation of the system or program. These faults are said to be masked.

In the literature, the level of fault masking in a system is referred to as either its derating factor [26] or its vulnerability factor [7]. In this thesis, we use the term vulnerability factor.

A vulnerability factor quantifies the amount of fault masking in a system. For instance, if 10% of faults, on average, will affect correct operation of a system, that system’s vulnerability factor is 10%. A higher vulnerability factor implies a more vulnerable (less reliable) system. Vulnerability factors vary with a number of factors, including the workload executing on the system, the operating system, and the microarchitecture. A vulnerability factor will also vary over time as system conditions change. Finally, vulnerability factors can be defined for subsets of a system. For example, it is common to define and measure a vulnerability factor for every hardware structure in a system.
2.4 Techniques to Measure Vulnerability Factors

Techniques to measure vulnerability factors fall into two broad classes, static and dynamic techniques. Static analysis does not require simulation of actual usage of the system. Instead, it relies on statically-determined parameters (e.g., gate fanout, wire length, and input value probabilities) to arrive at a vulnerability factor estimate for a given system under test. Dynamic techniques, in contrast, use information gathered during simulation or operation of the system (e.g., observing the propagation of a fault during system operation) in order to estimate vulnerability factors. This thesis uses dynamic techniques to estimate vulnerability factors, so in this section we describe these techniques in detail. We describe related work on static analysis in Section 2.5.

Traditionally, dynamic techniques to measure vulnerability factors have used a technique known as fault injection. More recently, researchers have introduced probabilistic modeling to address some of the drawbacks of fault injection. In order to give the reader a broad understanding of the process of measuring vulnerability, we discuss both major paradigms in detail in this section.

2.4.1 Fault Injection

The most traditional method to assess system reliability is fault injection [27]. To use fault injection as the basis for determining vulnerability, a workload is executed on the device under test and, at a random point during execution, a randomly-chosen bit is flipped in the structure under observation. The program output is then examined to determine whether the fault caused a visible failure (that is, whether the fault caused a change in the workload’s outputs or execution state). If a failure occurs, the injected bit is vulnerable during the cycle of injection. To achieve statistical
significance, this process must be repeated many times per structure. The structure’s *vulnerability factor* can be calculated by dividing the number of vulnerable injections by the total number of injections. This is an estimate of the likelihood that a fault within the structure under test will lead to an error.

Fault injection can be performed either in simulation (software) or on the actual device silicon (hardware). Both methods have unique attributes, but also share some benefits and drawbacks.

**Hardware Fault Injection**

In hardware fault injection (HFI), faults are inserted into the actual device silicon, either by means of dedicated testing hardware [28] [29] or by a source of charged particles such as an electron beam [30]. Thus, HFI is the only vulnerability assessment method that does not depend on having access to the internals of the processor microarchitecture. Since HFI operates on a real device, it takes into account all real-world effects such as operating system interactions with the test workload and IO device latencies that are often approximated in simulation. Thus, HFI has the potential to yield the most accurate assessment of reliability. In addition, since the workloads are executing on real silicon, each run takes a relatively short amount of time; therefore, most workloads can be run to completion. This simplifies the task of determining whether a particular injection leads to an error: the system’s output can simply be compared to its fault-free output.

By definition, however, HFI must be performed after first silicon is manufactured. For most products, this is too late in the design cycle to have any impact on design. Furthermore, designing dedicated testing hardware for a chip or technology process can be prohibitive in die area or design time, and subjecting a chip to an electron
beam is time-consuming and expensive [30]. Thus, the results of HFI typically cannot be used to influence the design under test. They can, however, be used to impact future designs in a particular technology process and to validate the results of earlier vulnerability assessment. Therefore, most semiconductor companies perform significant HFI testing on their completed designs [31] [26].

Software Fault Injection

In software fault injection (SFI), in contrast to HFI, faults are injected in a simulation framework, either a low-level hardware description language (HDL) model or a high-level performance simulator [17]. These models are available before a design is complete; therefore, results of SFI can impact the design of the chip under test.

Unfortunately, however, simulation is slow relative to native execution. This means that most workloads cannot be run to completion; in practice, each SFI injection typically runs for several thousand cycles in an HDL model or several million or billion cycles in a performance simulator. It is possible that, at the end of the run, a fault will have been neither activated nor masked; its result is unknown. Furthermore, since the workload is not complete, it is impossible to compare its output to the output of a fault-free run to determine vulnerability. Therefore, designers typically use other criteria to determine whether a fault is activated. For example, a fault is often considered activated if it propagates to memory [4]. This is possible since (unlike in HFI) the simulator framework allows visibility into processor structures. These approximations can, however, introduce inaccuracy into the vulnerability calculation. In addition, similarly to HFI, multiple injections must be performed per structure in order to achieve statistical significance. This makes SFI extremely time-consuming to do properly. In practice, SFI is often performed without proper statistical significance,
leading to potentially inaccurate vulnerability estimates [7].

2.4.2 Probabilistic Modeling

To address the issues with fault injection presented above, researchers developed probabilistic modeling [7]. Probabilistic modeling determines a structure’s vulnerability factor not by injecting faults but by determining on a cycle-by-cycle basis whether a fault in a bit would result in an error. Thus, a probabilistic fault model can generate a statistically significant vulnerability estimate in a single, fault-free, execution of a program. This represents a significant reduction in runtime over fault-injection techniques. Probabilistic modeling requires access to the underlying microarchitecture; it typically works by recording events to the hardware structure under test and updating vulnerability information accordingly. Therefore, it can only be applied in simulation, and vulnerability results must still be validated using HFI on actual silicon. However, probabilistic modeling is well-suited to environments such as performance simulation, since it allows a user to quickly generate a statistically-significant vulnerability measurement. Furthermore, performance models are typically available early enough in the design stage to allow these vulnerability measurements to have a significant impact on the processor design.

2.5 Related Work

In recent years, researchers have used the concepts discussed in the previous section to develop many techniques to measure system vulnerability. We discuss several of these techniques in this section.
2.5.1 Static Analysis

Several static analysis techniques have been proposed at both the circuit / device level of the system stack, and at the architectural and software levels. Note that because a single device or circuit might run tens of thousands of software programs (and vice versa), static analysis techniques generally assess vulnerability at a single level of the system stack (e.g., device or program level).

Zhang et al. combine accurate cell library characterization (typically SPICE models using underlying device parameters) with an efficient representation of fault pulses as Binary Decision Diagrams (BDD) in order to achieve a fast, scalable, circuit-level vulnerability analysis [32]. This method captures both logic masking through the combinational circuit as well as timing vulnerability factors in one representation.

Asadi and Tahoori derive a method to determine a vulnerability factor for combinational circuits (the authors refer to this as a logic derating for combinational circuits) [33] [34]. The authors derive these factors based on statically-derived propagation probabilities through combinational logic circuits. The authors subsequently extended this work to include statically-derived timing vulnerability factors for sequential elements in a combinational circuit [35].

At a software level, Pattabiram et al. use backward slices of critical variables and choose to selectively protect these variables [36]. The authors introduce a static, compiler-based technique for identifying and ultimately protecting, these critical variables. This allows a compiler to reduce the overhead associated with data protection with only a small reduction in the program’s fault tolerance.
2.5.2 Dynamic Techniques

In 1997, Somani and Trivedi proposed the *cache error propagation model* that defined an error as a fault that propagated out of the cache [27]. Their work used software fault injection to determine the vulnerability factor of a cache. Subsequently, Kim and Somani used this model to measure the reliability of data cache accesses [37]. A specific contribution of this latter work was the authors’ modelling of tag array failures. They defined three categories of tag failure: pseudo-hit, pseudo-miss, and multi-hit, based on the effects of an error on a subsequent read.

Subsequent work by Asadi et al. extended these categories with a fourth category of failure, replacement error [3]; this work used a probabilistic model to measure the vulnerability of cache tag, data, and status arrays. This work was extended and used in several studies presenting results on and methods of mitigating cache vulnerability to transient faults [38] [39] [40].

Seifert et al. demonstrated that it is possible to analytically determine a *Timing Vulnerability Factor* for sequential circuit elements such as latches and SRAMs [41]. By analyzing the timing characteristics of each device, and the delay characteristics of the input network to the device, the authors found that a particle strike would not affect device operation during a substantial fraction of each cycle. For many devices, the vulnerable timing window is only 25% of the full cycle time. Until that point, typical simulations assumed that a latch, for example, was vulnerable for 50% of each cycle; as a result, these simulations would significantly overestimate the actual fault rate of a device.

Wang et al. performed fault injection on a Verilog HDL model of a superscalar processor [42]. They perform thousands of injections on their processor pipeline and determine that less than 15% of the injected faults result in user-visible errors.
The authors analyzed the most vulnerable structures in a processor; protecting these structures with low-overhead redundancy techniques (e.g., ECC), reduced the error rate by a further 75%.

*SoftArch*, a methodology and tool proposed in [8], extends the idea of error propagation to a probabilistic model. The methodology tracks the probability that a bit will be affected by a transient fault, either because the fault is *generated* in that bit, or because the fault is *propagated* to that bit from a previously-affected bit. This state is tracked for values that could affect program outcome; this enables the tool to generate an estimate of a program’s *Mean-Time-To-Failure* during the simulation run.

The IBM Power6 processor has extensive built-in error handling and recovery mechanisms. To test and assess the functioning of these mechanisms, Sanda et al. developed a fault modeling infrastructure [26]. Their methodology relies on statistical fault injection into micro-architectural and architectural simulation. The two major contributions of this work are: (1) a correlation between a statistical fault injection study into RTL and an accelerated beam (hardware fault injection) study using both protons and neutrons; and (2) the distinction between *machine derating* and *application derating* to separate fault masking in hardware and software. The authors report a good correlation between their fault injection into RTL and the accelerated beam experiments; this gives confidence into the ability of software-based models to provide accurate, statistically significant fault masking values. A major contribution of this work is the distinction between machine derating and application derating; it is a leading attempt to separately quantify fault masking in different system layers.
2.6 Architectural Vulnerability Factor

The most commonly-used vulnerability metric in current literature is the Architectural Vulnerability Factor (AVF) proposed by Mukherjee et al [7]. AVF was first developed to cover pipeline structures such as an instruction queue. The AVF of a structure is defined as the percentage of bits in a structure that are necessary for correct program execution over the simulation lifetime.

The AVF of a hardware structure can be calculated as the fraction of 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 [7]:

$$AVF_H = \frac{\sum_{n=0}^{N} \text{(ACE m-bits in } H \text{ at cycle } n)}{B_H \times N}$$  \hspace{1cm} (2.2)

To measure the AVF of a structure, the authors also introduce a technique known as ACE Analysis. A bit that is necessary for correct program execution is deemed necessary for Architecturally Correct Execution, and is termed an ACE bit. All other bits are unACE bits. ACE Analysis assumes that all bits are ACE unless they can be conclusively proven unACE. The method attempts, at every clock cycle, to conclusively prove whether a bit is unACE. Examples of unACE bits are bits solely present for performance enhancement, the operand bits of a NOP instruction, and opcode bits in a killed instruction.

This work was subsequently extended to cover caches and other address-based structures [43]. The work extends AVF measurement and calculation to data and tag arrays in caches. This work determines the vulnerability factor of a cache based on the ACE lifetime of cache words. It also defines a set of tag behaviors that closely resemble the ones defined by Kim and Asadi. The authors also introduce the concept of cooldown, a technique to compensate for the edge effects that occur at the
end of simulation. In prior work, researchers have treated any bits whose ACE-ness is unknown at the end of simulation as ACE, since their unACEness could not be proven. This is a conservative estimate that guarantees the reported ACE value will be an upper bound of the true ACE value. In structures with long data lifetimes (e.g., caches), this can lead to a potentially significant overestimate of the AVF, as many live entries can be present at the end of simulation. Cooldown compensates for this by running the simulation for a fixed number of cycles past the measurement endpoint to determine if any unknown bits can be subsequently marked unACE.

2.7 Using AVF to Improve System Design

Most recent research in the area of transient fault vulnerability has used AVF as the metric of interest. While these studies have generated insight into AVF behavior and have proposed novel applications for AVF, they have not proposed any substantive changes to the underlying metric. Recent studies fall broadly into three categories: exploring the potential limitations of AVF; observing differences in reliability behavior of applications; and estimating AVF at runtime to enable systems to adapt to a changing vulnerability environment.

2.7.1 Exploring Potential Limitations of AVF

In [44], the authors explore the validity of using AVF estimates to generate Mean-Time-To-Failure (MTTF) estimates for a processor. They observe that, because AVF values are not independent, using average AVF values to calculate MTTF can result in incorrect MTTF values. Specifically, the authors observe that the AVF value of bit $b$ at time $t$ is correlated with the AVF value of $b$ at time $t + k$; therefore, if $k$ is large
relative to the raw fault rate, inaccuracy is introduced into the AVF calculation. Furthermore, the authors show that, if the AVF values of structures $S$ and $R$ are correlated, then adding the structures' FIT rates to get a system FIT can also yield inaccurate results. The authors demonstrate that the magnitude of this error is small for typical programs and systems; they observe errors greater than 1% and up to 100% in systems with with long running workloads (e.g., 1 day to 1 year) that have extremely large components or high raw fault rates (e.g., greater than 10 faults per year per component). However, this imprecision only applies to MTTF calculation; it does not diminish the ability of AVF to yield insight into the reliability behavior of a structure.

In [45], the authors explore discrepancies between AVF values calculated using ACE Analysis and AVF values calculated using Software Fault Injection. The authors observe that ACE Analysis consistently yielded much looser bounds on the AVF of a processor structure than SFI; they attributed this to the difficulty of constructing a detailed ACE Analysis model and the inability of a single-pass simulation to detect phenomenon such as $Y$-bits [46]. Biswas et al. offer a rebuttal to this work in [47]; the authors demonstrated that using slightly more state in simulation significantly improved the ACE Analysis estimates, and suggested a post-commit analysis window to observe the effect of $Y$-bits.

### 2.7.2 Estimating Vulnerability at Runtime

In [48], the authors demonstrate that the AVF of a processor structure closely correlates to several easily-measurable architectural and microarchitectural markers (e.g., structure occupancy). Using a set of representative benchmarks, the authors construct an equation to relate these variables to AVF, and propose implementing this
equation in hardware to estimate the AVF of a structure at runtime. The authors demonstrate that this can yield accurate runtime AVF estimates, and allow the system to respond to high vulnerability conditions by dynamically enabling redundancy features.

Similarly, Duan et al. use Boosted Regression Trees to correlate AVF with processor statistics and extend this to predict correlations across microarchitectural changes [49]. The authors then propose using the Patient Rule Induction Method to quickly identify regions of high AVF at runtime based on the behavior of relatively few processor statistics.

Soundararajan et al. propose a technique to bound AVF from above by tracking the maximum number of critical bits present in a structure [50]. This can be easily determined at runtime by (for example) the Issue Queue by tracking dispatches, issues, and squashes. The system can bound the maximum allowable number of critical bits, which can determine how many new instructions to dispatch during each cycle. The authors then propose mechanisms to vary these rates (e.g. throttling).

Li et al. proposed the use of error bits to determine AVFs at runtime [4]. The hardware can inject “errors” into these error bits, and determine which of these errors are masked, and which would affect program execution. This essentially mimics the behavior of fault injection at runtime, allowing the system to dynamically determine an AVF value for each structure. The authors show that this is possible for many structures such as the Issue Queue, Reorder Buffer, and Register File; however, it cannot estimate AVFs for caches or memory as the lifetime of data values is too long.
2.7.3 Dependence on Application Behavior

In [51], the authors demonstrate that AVF of a system can vary substantially when the same source code is compiled with different compiler optimizations. These differences suggest that system vulnerability depends substantially on the workload, and even on the specific source code executed. In [52] and [53], the authors note that various applications treat data differently; specifically, many applications do not have a requirement that every computation be correct. This allows a different class of redundancy technique that can take advantage of the varying reliability requirements of these computations. Furthermore, this work motivates us to re-examine the definition of vulnerability. Specifically, how can we quantify whether a particular calculation is important, and how is this information conveyed to the AVF measurements?

One attempt to answer this question for a specific class of processors can be found in [54]. The authors note that graphics computations are, in general, more tolerant to faults than general-purpose computations. Many transient faults in graphics hardware will result only in a corrupted pixel in one frame of display; this is typically imperceptible to the human eye and therefore not critical. The authors therefore introduce the Visual Vulnerability Spectrum to quantify this tradeoff.

2.8 Limitations and Opportunities

Vulnerability analysis is a powerful concept, but a key limitation of previously proposed metrics, including AVF, are that they do not exploit the abstraction present at higher levels of the system stack. As a result, it is difficult to differentiate fault masking in hardware from fault masking in software. As shown in Figure 2.1, however, vulnerability depends on both the workload as well as the microarchitecture, and fault
masking can happen in the microarchitecture or in the program itself. When using AVF as a metric, it is difficult (if not impossible) to separate these various effects.

Two examples can help illustrate this point. The first example is a study by Jones et al. found that compiler optimizations had an impact on AVF [51]. However, based on their results, it is impossible to know whether the observed trends will apply across different microarchitectures. If the primary impact of a compiler optimization is to change the vulnerability of the program, then we expect that the result will be similar on another microarchitecture. However, if an optimization instead changes program behavior in a way that causes the microarchitecture to respond with different vulnerability, then the changes might be microarchitecture-specific. (An example of the latter transformation is a compiler optimization that increases the number of conflict misses in a cache, creating more memory stalls. On a different microarchitecture with a different cache organization, this change might have much less of an effect.)
The second example is that of a software vendor wishing to assess the fault tolerance of a particular program. In particular, the software vendor may want to identify regions of the code that are highly vulnerable to transient faults and add fault tolerance features to just those sections. In order to do this evaluation, the software vendor must measure the vulnerability of the program while running on every possible target system configuration, including every combination of microarchitecture, operating system, virtual machine, library functions, etc. The number of possible configurations can easily number in the hundreds or thousands. Then, these results must be combined to get a sense of which functions are the most vulnerable across all systems. This is clearly an extraordinarily time-consuming task, and it is complicated by the fact that the software vendor may not have knowledge or access to every possible target system.

Furthermore, most current vulnerability analysis techniques require access to a microarchitecture model of the target system. These models are typically only available to the processor manufacturers, and are not available to software vendors or the general public.

This task would be greatly simplified if the software vendor could instead measure the vulnerability of the program independent of the rest of the system. This would allow the program to be profiled just once, on any target system, and the functions ranked in terms of their vulnerability. Furthermore, since the evaluation is microarchitecture-independent, this implies that a microarchitecture model is not necessary; an architecture model would suffice. Many robust architectural simulators are publicly-available (e.g., Simics [55]).

The same argument applies to all components in a system: microarchitecture, operating systems, virtual machines, and user programs. The ability to assess the
vulnerability of one component independently of all others would greatly expand the utility of vulnerability analysis.

To accomplish these goals, we must develop a vulnerability analysis framework that exploits the abstractions present in modern computer systems. In particular, we must split vulnerability analysis across boundaries such as the ISA. This will enable us to differentiate vulnerability on one side of the boundary (e.g., in hardware) from vulnerability on the other side (e.g., in software). That is the goal of this thesis.

2.9 Benefits of Abstraction

The primary benefit of adding abstraction to vulnerability analysis is that it allows a designer to assess and improve the fault tolerance solely of his/her particular component (e.g., a user program). This enables a much broader segment of the computer architecture and software engineering communities to participate in the vulnerability assessment and remediation process; currently, these activities are typically performed by architects equipped with a microarchitectural model. For example, abstraction would allow the designer of a software application to assess the hardware-independent vulnerability of their code, identify highly-vulnerable regions, and focus their effort on reducing vulnerability in these areas (either via algorithmic changes or by adding redundancy). We demonstrate this type of analysis in the section dealing with the Program Vulnerability Factor.

Another benefit of abstraction is a substantial reduction in the overall effort required for vulnerability assessment. While there may be hundreds of implementations of each system component, the number of possible combinations of components (i.e., systems) is in the millions, even considering only systems where reliability is
a primary concern. By measuring the vulnerability of each component and reusing (rather than recomputing) these results to compute system vulnerability, abstraction can minimizes the number of times a given computation needs to be performed. An example of this is *dynamic-dead analysis*, a common technique to measure fault masking within a program by assessing the liveness of dynamic instructions [7] [43] [45]. Dynamic-dead analysis as described by Mukherjee et al. [7] requires thousands of lines of code and consumes a significant fraction of a simulator’s execution time. This is due to the analysis itself, but also to the state required to defer AVF computation until the analysis is complete. It would be much preferable to perform dynamic-dead analysis on a program once (offline), store the results in a trace, and use this as a simulator input when a benchmark is executed. This trace can then be accessed during microarchitectural simulation. This requires significantly fewer resources during simulation and can substantially reduce simulation time, which can in turn reduce time-to-market for new products. Furthermore, the up-front cost of the offline PVF analysis is amortized over the lifetime of the benchmark, allowing the analysis to take additional time to achieve higher accuracy (e.g., by performing fault injection into the architectural state [17]), and ultimately yielding a faster, more accurate, system vulnerability measurement. As part of this thesis, we develop and implement a framework that quantifies this benefit.

Finally, abstraction can allow architects to determine the source of fault masking, information that is crucial when attempting to understand and remediate the effects of faults. For instance, if a change in workload leads to a reduction in system vulnerability, this could be due to a greater level of fault masking within the new workload; or to a greater level of fault masking within the hardware because of a characteristic of the new workload. The difference is important: if the latter case is true, the fault
masking is external to the workload; a different system might respond very differently to the same workload. However, if the fault masking is inherent in the workload, we know that a different system will contain those same characteristics when presented with the same workload.
Chapter 3

The System Vulnerability Stack

In this chapter, we present a unified, full-system transient fault vulnerability framework. We call this framework the *System Vulnerability Stack* [56]. The vulnerability stack computes full-system vulnerability from individual vulnerability factors measured at each level of the system stack. Each vulnerability factor measures the fraction of faults visible to one system layer that propagate to another system layer. For example, the vulnerability of a virtual machine is the fraction of faults visible at the ISA boundary provided by the hardware that propagate to program-visible state (i.e., to the ISA boundary visible to the program being executed). The individual vulnerability factors can be combined to generate a system vulnerability factor (e.g., AVF). In this thesis, we use the terms *system vulnerability* and AVF interchangeably. However, we note here that system vulnerability also includes device-level fault masking that can be quantified using *Timing Vulnerability Factors*.

This chapter is organized as follows. First, we present the definitions and concepts that serve as the basic underpinning of the vulnerability stack. Next, we take these concepts and use them to develop the math to calculate an individual vulnerability
3.1 Definitions and Concepts

We define a fault as a raw failure event such as a single-bit flip. 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 3.1). 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 a range of 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 a user program. For a fault to cause an error within a given system layer, it must occur within a visible bit. 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., a user program) is a subset of the bits visible to a lower level of the stack (e.g., the microarchitecture).

There are two ways that a visible fault can cause an error in 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. The ACEness of this fault (to the system) depends on the behavior of the user program. Therefore, we call a bit that contains an exposed fault a *pACE bit* (potentially-ACE bit).

A visible fault can also 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 never exposed to the user program, but still causes an error in the microarchitecture. We refer to this fault as *activated* within the microarchitecture. The ACEness of this fault to the system depends only on the behavior of the microarchitecture. Therefore, we call a bit that contains an activated fault an *iACE bit* (internally-ACE bit).

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*, and we call a bit that contains a masked fault a *nonACE bit*. (We use the term *nonACE* within a layer to distinguish this from a bit that is *unACE* to the system as defined by Mukherjee et al. [7]).

A bit is either pACE, iACE, or nonACE within every system layer to which it is visible. At a given layer, only pACE bits are made visible (exposed) to higher system layers. For instance, nonACE and iACE bits within the hardware are not visible to the operating system or user program. It is possible for a bit to be both iACE and pACE within a layer. However, we assume without loss of generality that an iACE bit is not also pACE; that is, an iACE bit is not visible to any higher system layers. A bit’s ACEness to the system will be determined by the highest layer to which it is visible. If the bit is iACE within that layer, it is ACE to the system. If the bit is nonACE 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 nonACE to the microarchitecture and not 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 [57]. A microarchitecture-visible fault in this bit will be exposed to the next layer of the system, typically the operating system. Therefore, the bit is pACE within the hardware. Assume the operating system does not examine the address, but allows the load to proceed normally. Then the bit is also pACE within the OS. Finally, since the load is dynamically-dead, the bit is nonACE 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 bit will be pACE within the hardware, iACE within the operating system, and not visible to the user program. Therefore, in this case the bit will be ACE to the system.

3.2 Individual Vulnerability Factors

The vulnerability stack is composed of vulnerability factors from all system layers, and it is often interesting to examine individual vulnerability factors on their own. To compute the vulnerability of one system layer, we assign iACE and pACE bits a vulnerability factor (VF) of 1, and assign nonACE bits a VF of 0.

Since typical systems spend most of their time executing user program code (and
CHAPTER 3. THE SYSTEM VULNERABILITY STACK

not, for example, operating system code), the two most dominant vulnerability factors will tend to be PVF (for the user program) and HVF (for the hardware). In this section, we introduce and describe the measurement of both PVF and HVF.

3.2.1 The Program Vulnerability Factor

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. For example, the Alpha ISA defines the size of the Integer Register File as thirty-two 64-bit registers; this architectural resource contains 2048 bits. (A physical register file implementation of this resource may contain a substantially greater number of 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 [58].

The PVF of an bit is the fraction of time (in instructions) that the bit is either pACE or iACE, and the PVF of an architectural resource $R$ of size $B_R$ over $I$ instructions is the fraction of pACE and iACE bits in the resource:

$$PVF_R = \frac{\sum_{i=0}^{I} (\text{pACE and iACE bits in } R \text{ at instruction } i)}{B_R \times I} \quad (3.1)$$
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.

### 3.2.2 The Hardware Vulnerability Factor

The *Hardware Vulnerability Factor*, or HVF, is the vulnerability factor that quantifies fault masking within the microarchitecture. HVF measures whether a value is necessary for proper hardware behavior (the value is iACE) or whether the value becomes visible to the software (the value is pACE). Therefore, the HVF of a hardware structure is not affected by a program’s use of a value. For instance, the HVF of a load buffer entry does not depend on whether the result of the load is logically masked by a later operation in the program. Logical masking is a function of the workload, and will be captured in the program’s PVF.

The HVF of a hardware structure is the fraction of bits in the structure that are either iACE (activated by the hardware) or pACE (exposed to the software). The HVF of hardware structure $H$ with size $B_H$ over $N$ cycles can be represented as follows:

$$HVF_H = \frac{\sum_{n=0}^{N} (iACE \text{ and } pACE \text{ bits in } H \text{ at cycle } n)}{B_H \times N}$$ (3.2)

Both iACE and pACE bits can be identified using an RTL model or a performance simulator. Identifying pACE bits requires tracking values that become visible to architectural state. A bit is pACE if a fault in that bit will corrupt committed state. In a register file, for example, bits within physical registers that have valid architectural mappings are pACE. In instruction-based structures (e.g., Issue Queue,
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.

Figure 3.2 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, all bits of $P1$ are nonACE within the microarchitecture, and their HVF 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. Therefore, all bits of $P1$ are pACE within the microarchitecture, and their HVF is 1.

If the resulting program-visible fault is masked, the bit is nonACE within the program and it is assigned a PVF of 0. For instance, a fault during cycle 14 will be masked if it does not affect the compare operation in cycle 15. PVF uses dynamic instructions to mark time. Therefore, the PVF assigned to bit $b$ 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:

\[
AVF_{b,1-15} = \frac{1}{15} \times (PVF_{R1b,i1} + PVF_{R1b,i2} + PVF_{R1b,i3} + PVF_{R2b,i7} + PVF_{R2b,i8} + PVF_{R2b,i9} + PVF_{R2b,i10} + PVF_{R2b,i11} + PVF_{R2b,i13} + PVF_{R1b,i14} + PVF_{R1b,i15})
\]

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:
Figure 3.3: *Multi-exposure bits.* A single microarchitecture-visible fault in a *multi-exposure bit* (e.g., 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 bit requires evaluating the impact of all the program-visible faults simultaneously.

\[
AVF_{b,1-15} = \frac{1}{15} \sum_{n=1}^{15} HVF_{b,n} \times PVF_{a_{b,i_n}}
\]

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

\[
AVF_{H,N} = \frac{1}{N \times B_H} \sum_{b=1}^{B_H} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{a_{b,i_n}}
\] (3.3)

### 3.4 Multi-Exposure Bits

Some bits in a hardware structure will cause multiple program-visible faults when corrupted; we call these *multi-exposure bits.* Figure 3.3 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 multi-exposure bit. Instead, we must calculate AVF of a multi-exposure bit $b$ as follows:

$$AVF_{b,N} = \frac{1}{N} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{A_{b,n}}$$  \hspace{1cm} (3.4)

In this equation, $A_{b,n}$ is the set of all program-visible faults resulting from the microarchitecture-visible fault in bit $b$ at cycle $n$. $PVF_{A_{b,n}}$ is 1 if and only if the set of faults $A_{b,n}$ causes the program to produce incorrect output. Calculating a precise value for AVF using Equation 3.4 requires modeling multiple simultaneous faults during PVF analysis. This is tractable for a small set of faults, but is not feasible to evaluate in general.

In practice, however, there are many cases where we can conclusively determine that a multi-exposure bit will have a PVF of 0. For instance, a bit in the Physical Source Register Index field of the IQ will be unACE if the destination register is dead (i.e., its contents are never read). There are many phenomena (e.g., dead values, logical masking, variable length operands) that allow us to determine that a multi-exposure bit has a PVF of 0. Therefore, in this work we consider a multi-exposure bit to be unACE if we can prove that the resulting fault combination has a PVF of 0. Otherwise, we conservatively treat these bits as iACE.

### 3.5 Multiple Software Layers

The vulnerability stack can also quantify fault masking from software layers such as an operating system or virtual machine by focusing on the interfaces that these layers implement: a virtual machine implements an ISA, and an operating system
Figure 3.4: Computing system vulnerability with multiple software layers. The behavior of architectural register R1 on a system with a virtual machine and user program. VM instructions are shown in black; user program instructions are shown in gray. R1 is mapped to program register V1 during instructions 1-5 and 14-15; to program register V2 during instructions 7-9; and not mapped during instructions 6 and 10-13. The vulnerability of R1 can be calculated using the VMVF and PVF during each instruction.

implements an Application Binary Interface (ABI). Figure 3.4 depicts this process for register R1 on a system with a virtual machine and a (guest) user program. Instructions 1-5 and 15 are program instructions; instructions 6-14 are virtual machine instructions. From instructions 1-5, R1 is mapped to program register V1; a VM-visible fault in these cycles will be exposed to the user program. At instructions 6 and 7, the VM performs a context switch, saving V1 to memory and V2 to R1; R1 is mapped to guest register V2 from instructions 7-9. From instructions 10-13, the VM uses R1 to update its internal state; a fault in R1 will be masked during instructions 6, 10, and 13, and activated during instructions 11-12. Finally, in cycle 14, the VM restores the program context and the program continues execution in cycle 15.

To calculate the vulnerability of R1 over this period, we proceed similar to Equation 3.3: from instructions 1-5 and 14-15, a VM-visible fault is exposed to the user program, so \( V_{F_{R1}} = PV_{F_{V1}} \). Similarly, from instructions 7-9, \( V_{F_{R1}} = PV_{F_{V2}} \). For instruction 6 and instructions 10-13, the register is not mapped to program state; the vulnerability can be given by a Virtual Machine Vulnerability Factor (VMVF), which
is 1 if a fault would be activated and 0 if it would be masked. Therefore, the overall vulnerability of \( R_1 \) can be calculated as follows:

\[
VF_{R_1} = \frac{1}{I} \times \sum_{b=1}^{B_{R_1}} \sum_{i=1}^{I} VMVF_{b,i} \times PVF_{a,b,j_i}
\] (3.5)

Here, \( PVF_{a,b,j_i} \) denotes the PVF of the architectural state \( a \) mapped to bit \( b \) at time (instruction) \( j_i \), the program instruction that consumes the value stored in \( b \) during VM instruction \( i \). \( VF_{R_1} \) is the vulnerability of architectural register \( R_1 \) in the system under test; this value can be used in Equation 3.3 to calculate the AVF of the physical registers to which \( R_1 \) is mapped.

### 3.6 Computing System Soft Error Rates

For a given bit \( b \) in the system, its overall vulnerability during cycle \( n \) can be calculated as the product of the vulnerability factors of every component \( c \) through which it passes:

\[
VF_{b,n} = \prod_C cVF_{b,n}
\] (3.6)

This can be used to calculate the SER of the bit over \( N \) cycles:

\[
SER_b = \frac{1}{N} \times \sum_N \left( FIT_{b,n} \times \prod_C cVF_{b,n} \right)
\] (3.7)

Computing this across all bits \( B \) in a system yields an expression for the average SER of the system:
This equation represents an exact calculation of the average SER using instantaneous VF values; we can only use average VF values if the VF values are independent [44]. In general, TVF is assumed to be independent of the other VFs [41]; however, we currently make no assumptions of independence about the rest of the vulnerability factors.

3.7 Summary

In this chapter, we introduced the central idea of this thesis: the System Vulnerability Stack. The vulnerability stack allows computation of individual vulnerability factors for each layer of a system, such as the hardware, operating system, or program. This allows an individual system component (e.g., a user program) to compute its own vulnerability independent of the other components in the system. This can be a powerful benefit to both hardware and software designers, and can enable a substantial increase in the scope of vulnerability assessment.

We first presented the concepts and definitions that create the foundation for the vulnerability stack. We then demonstrated how to take these ideas and derive equations to compute system vulnerability. We discussed multi-exposure bits, which present a problem in calculation of the vulnerability factors, as well as the stack’s capability of handling multiple software layers. We also briefly described how to
calculate system-wide soft error rates using multiple vulnerability factors. Finally, we discussed two of the most prominent individual vulnerability factors, the Program Vulnerability Factor and the Hardware Vulnerability Factor.

In this chapter, we presented the reader with a thorough description of the theory behind the vulnerability stack. In the next chapter, we discuss our method for turning this theory into practice using the simulation infrastructure that we developed as part of this thesis.
Chapter 4

Simulation Methodology

In this chapter, we discuss the modeling infrastructure that we developed for the vulnerability stack. This will help orient the reader to the practical means by which one measures vulnerability factors in simulation, and will provide a basis for evaluating the results in the sections to follow.

As part of this thesis, we developed two separate simulation infrastructures. The first is a framework built on the open-source M5 microarchitectural simulator from the University of Michigan, and is capable of evaluating in detail vulnerability factors for all levels of the system stack. The second is a proof-of-concept, PVF-only, framework built on the Pin dynamic binary instrumentation tool from Intel. The latter infrastructure served to demonstrate that PVF can be measured using architecture-only simulation, and highlighted the simulation time benefits of this approach.

In order to orient the reader to simulation techniques, we present background information on ACE Analysis, a method to assess the AVF of a hardware structure, upon which technique our simulation infrastructure is based. We then discuss how to adapt ACE Analysis for use in measuring multiple vulnerability factors. Finally, we
discuss our specific simulator frameworks based on the M5 simulator and Pin dynamic binary instrumentation framework.

4.1 Background: ACE Analysis

ACE Analysis was introduced by Mukherjee et al. as a method to bound the AVF of a hardware structure relatively early during processor design [7]. ACE Analysis uses a performance simulator to generate an upper-bound estimate of a hardware structure’s AVF. Prior to the development of ACE Analysis, most work in estimating AVFs used statistical fault injection. ACE Analysis improves on fault injection in many aspects. First, fault injection requires many simulation passes (injections) in order to achieve a statistically significant AVF estimate. ACE Analysis requires just one simulation pass. As a consequence, ACE Analysis can generally track values for more cycles and identify fault masking that fault injection would miss. ACE Analysis can also give insight into system behavior by identifying reasons that bits become unACE. Finally, fault injection work is typically performed in an RTL model, which is not available until late in the design process. ACE Analysis was specifically designed to be used in a performance simulator model, which is available much earlier in the design process.

ACE Analysis functions as follows. First, we assume that all bits are ACE unless we can conclusively prove otherwise. We then simulate the system and identify events that cause bits to be unACE. For example, if a value is overwritten without ever being read, a fault in that value will not cause an error. Therefore, this event causes a bit to be unACE. Some values become unACE before the associated state or instruction become visible to architectural state (termed micro-architectural unACE-ness by Mukherjee et al.). Other values become unACE after becoming visible to
architectural state (architectural unACEness). Identifying architectural unACEness generally requires tracking processor state past the point of commit. Mukherjee et al. proposed the use of a post-commit instruction window to identify these sources of unACEness.

Conceptually, a bit is ACE only if its value propagates to an I/O operation on the processor. However, due to the long latencies involved in tracking I/O operations, few ACE Analysis infrastructures track bit values this far. Instead, researchers typically use a fixed window size within which they identify as many sources of unACEness as possible. For example, Mukherjee et al used a post-commit window size of 40,000 instructions to identify sources of unACEness [7]. If an value could not be conclusively identified as unACE within this window, the authors assumed it to be ACE.

In a modern processor, there are many sources of unACEness, including: bits overwritten before they are read; idle or invalid state; mis-speculated state; state used solely for prediction or performance enhancement; state that is beyond its last use (called ex-ACE state); NOP and prefetch instructions; dynamically-dead instructions; and logical masking. (This is not a complete list; ACE Analysis does not require identifying all possible sources of unACEness. When using ACE Analysis, a more comprehensive list will lead to a tighter upper bound on AVF.)

To make this discussion concrete, Figure 4.1 presents a simple example calculation of AVF for a simple system using both fault injection and ACE Analysis. The system in this example is a 4-bit register. The block diagram at the top shows the behavior of the system over 10 cycles. The system starts (at cycle 0) in state 0000. At cycle 8, three bits are written to change the state to 1011. This is the end state of the system. We assume for this example that we only care about the end state of the system.

The box on the left shows the process one would use to calculate the AVF using
fault injection. We first inject a fault into bit 0 at cycle 0. We then simulate the system for 10 cycles and check the output. If the fault appears at the output, we declare it an error. (In this case, the fault is overwritten at cycle 8, and does not appear at the output.) We then repeat the process for bit 1 at cycle 0, and so on for all bits and cycles. At the end of simulation, we calculate our AVF estimate by dividing the number of errors (16) by the total number of injections (40). In total, fault injection requires 220 simulation cycles and calculates an AVF estimate of 40%.

The box on the right shows the same calculation using ACE Analysis. We first assume that all bits are ACE unless we can conclusively prove otherwise. We then simulate the system and look for events that cause unACEness. We discover writes to bits 0, 2, and 3 at cycle 8. Each of these writes contribute 8 unACE cycles (each bit is unACE from cycle 0 to cycle 8). At the end of simulation, we divide the total number of unACE bits (24) by the total number of bits (4) times the total number of cycles (10), and subtract the result from 1. This results in an AVF estimate of 40% and requires only 10 simulation cycles.

In general, fault injection requires $O\left(\frac{1}{2}N^2B\right)$ cycles to estimate AVF, where $N$ is the number of simulation cycles in the measurement window, and $B$ is the number of bits in the structure. In order to alleviate this large overhead, most fault injection campaigns use statistical sampling. However, even using a statistical sample, in this case fault injection would require nearly 100 cycles to generate an accurate estimate. In contrast, ACE Analysis requires $O(N)$ cycles for a complete analysis. ACE Analysis requires the simulator to track more state during each cycle (i.e., the number of unACE bits). However, this extra overhead is more than offset by the reduction in simulation cycles.
CHAPTER 4. SIMULATION METHODOLOGY

Figure 4.1: *A comparison of Fault Injection to ACE Analysis*. Fault injection requires many simulation passes to generate an AVF estimate. ACE Analysis requires only one pass, and thus requires many fewer simulation cycles than fault injection.
4.2 Adapting ACE Analysis to the Vulnerability Stack

Using ACE Analysis as a model for the vulnerability stack requires adapting the technique in several ways. First and foremost, we must compute an individual vulnerability factor for each system layer. Second, we must modify the ACE Analysis algorithm to classify bits as either iACE, pACE, or nonACE. (Standard ACE Analysis classifies bits as either ACE or unACE.) Finally, in order to compute AVF, we must add a final step which computes AVF from the individual vulnerability factors.

Fundamentally, we do one ACE Analysis pass per system layer. Therefore, if we have two system layers (e.g., hardware and a user program), we do two simulation passes: an HVF pass and a PVF pass. These simulation passes can be concurrent or consecutive; we use both techniques in this thesis. If the simulation passes are consecutive, the results of the first pass (typically the PVF pass) must be saved in a trace for use in the second pass (typically the HVF pass). We discuss our format for PVF traces in Chapter 6.

In this section, we discuss the changes required to all ACE Analysis passes, followed by a discussion of specific changes necessary specifically for hardware (e.g., HVF) and software (e.g., PVF) simulation passes.

4.2.1 Changes to All Simulation Passes

During every simulation pass, we must determine whether each bit is iACE (a fault in this bit will be activated within this layer), pACE (a fault will be exposed to another system layer), or nonACE (a fault will be masked). Classifying a bit as nonACE is very similar to standard ACE Analysis. For example, if a bit is overwritten before it
is ever read, then a fault will be masked and the bit is nonACE within this layer.

Classifying a bit as pACE requires tracking the propagation of the value contained within the bit and determining if the value (or an output of a computation using this value) ever becomes exposed to the interface to the next system layer. For some values, this is straightforward. For example, a physical register in the hardware that is mapped to a valid architectural register is immediately exposed to the user program. For other values, this requires more effort, and a simulator infrastructure must be able to cope with relatively long propagation latencies. For instance, a value within a control structure in the hardware might only affect program execution after several billion cycles. Similarly, a value within an architected register might remain live for billions of instructions before eventually propagating to program output. Finally, a value within a hardware structure near the front of the instruction pipeline (e.g., the fetch unit), might be read and used by multiple structures before it propagates to program-visible state. This must be tracked by the simulator, and the results must be communicated back to the front-end structure in question.

To classify bits as iACE, pACE, or nonACE, we typically use a fixed-size post-commit window for each simulation pass. During this window, we attempt to classify a bit as either nonACE or pACE. Otherwise, the bit will default to iACE. This methodology is used in order to provide a conservative upper bound estimate of each vulnerability factor.

4.2.2 Changes for Hardware Simulation Passes

A hardware simulation pass typically requires a much smaller post-commit window than the 40,000 instructions used by Mukherjee et al. This is because values within most large hardware structures have latencies of only a few hundred cycles before
they are either overwritten or committed to program-visible state. Thus, a conclusive
determination on their ACEness can be made within a few hundred cycles. For the
occasional hardware structure that has a longer average latency, there are two choices.
We can either use a small post-commit window, which will lead to a looser upper
bound on HVF (since more bits will be marked as iACE), or use a longer window and
generate a tighter upper bound on HVF. The longer window will increase simulation
time; therefore, this choice is typically dictated by the size of the structure (and thus,
its importance to the overall accuracy of AVF calculation).

4.2.3 Changes for Software Simulation Passes

A software simulation pass (e.g., PVF analysis) must measure time using dynamic
instructions. Typical ACE Analysis uses clock cycle to measure time; thus, we must
modify the ACE Analysis algorithm to use a dynamic instruction count to demar-
cate the time at which an event occurred. In particular, the simulator must use the
in-order committed instruction stream in order to measure time. When performing
architecture-only simulation, this is a relatively straightforward process since the only
instructions present in the system are from the committed instruction stream, and
these are executed in order. If a full microarchitecture model is used, however, care
must be taken to properly account for squashed instructions. Since a squashed in-
struction never becomes program-visible, it does not contribute to the program-visible
time count. Furthermore, a high-performance microarchitecture will often execute in-
structions out-of-order for performance purposes. This must also be carefully tracked
and accounted for in the time count.

Note that, in the latter case, the microarchitecture model is used simply as a
baseline simulation framework to calculate PVF; the specifics of the microarchitecture
will *not* affect the PVF calculation.

## 4.3 M5 Simulation Infrastructure

The M5 simulator is a modern, C++-based, simulation framework from the University of Michigan [59]. The simulator code features pervasive object-orientation, making it very modular, as well as extensive use of templates that allow one CPU model to handle multiple ISAs. Multiple CPU models are distributed with the simulator, but the framework is flexible enough to allow users to add their own CPU models. In this thesis, we used two of the provided CPU Models: the complex Out-Of-Order (O3) CPU, which is loosely based on the Alpha 21264 CPU; and the simple in-order Atomic CPU, which is a simple, 1 cycle-per-instruction functional model. We used the O3 CPU as our baseline microarchitecture for HVF simulation. We used the Atomic CPU as the baseline CPU for PVF analysis of a workload.

As part of our thesis, we added a vulnerability infrastructure to this simulator. This is a stand-alone component of the simulation framework that can be harnessed by any CPU model to measure the vulnerability (AVF) of any hardware structure. Components of the infrastructure include: a module to assess the HVF of each hardware structure; a module to assess the PVF of the architectural resources within a program; and a module to combine the results to generate an AVF estimate for a given structure.

Figure 4.2 shows a simplified code snippet from the vulnerability infrastructure we added to the M5 simulator. Each hardware structure is indexed using three values (*index*1, *index*2, and *index*3 in the figure). These indices uniquely identify a bit within a hardware structure. For instance, a bit in the cache is uniquely identified
by its set, line, and bit offset. Every time a CPU generates an event (e.g., a read) to that bit within the cache, the readObj() routine is called with the appropriate indices. This routine adds a read event to the appropriate object’s event queue.

The readObj() routine also adds the event to a structure-wide list of events waiting for analysis. Recall that ACE Analysis works by analyzing an event to determine if it causes a bit to be unACE. For example, if a bit is accessed only by a dynamically-dead read, that bit will be unACE. Objects on the structure’s event queue are separately analyzed for ACEness, and the results are returned to the structure. For example, if a read event is generated by a squashed instruction, it will be nonACE with respect to the hardware. A dynamically-dead event, on the other hand, will be pACE with respect to the hardware but nonACE with respect to the user program. Every object computes its own ACE time based on the ACEness of the events on its event queue. The structure computes its overall ACE time by summing every object’s ACE time.

We note here a simulation performance advantage that the vulnerability stack possesses over standard AVF analysis. The length of time that an event resides on an event queue is directly proportional to the amount of time it takes to analyze and resolve that event’s ACEness. When performing AVF analysis, this analysis can require tens of thousands of simulator cycles (e.g., when using a 40,000-instruction post-commit window to identify dynamic deadness). When performing HVF analysis, on the other hand, we are only determining whether the event will affect architected state. This analysis usually requires just a few hundred simulator cycles, typically until the instruction associated with the event has committed. This results in a much smaller event queue, leading to shorter lookup times, better cache behavior, and a significant decrease in simulation time.
class Structure {
    // Create an object in the structure
    void insertObj(Tick t, uint index1, uint index2, uint index3);

    // Delete an object from the structure
    void deleteObj(Tick t, uint index1, uint index2, uint index3);

    // Create a write event on an existing object
    void writeObj(Tick t, uint index1, uint index2, uint index3);

    // Create a read event on an existing object
    void readObj(Tick t, uint index1, uint index2, uint index3);
};

void Structure::readObj(Tick t, uint i1, uint i2, uint i3) {
    // Get the object associated with these indices
    Object obj = this->lookupObj(i1, i2, i3);

    // Add the read event to the object's event queue
    obj.eventQ.addRead(Tick t);

    // Add the event to a global event list for later analysis
    this->eventQ.addRead(Tick t);
}

Figure 4.2: Simplified source code example from our M5 simulation infrastructure. Every structure is indexed with three indices. When an entry in the structure is read by the CPU, a read event is added to the corresponding object’s event queue. The event is also added to a structure-wide list of events waiting for analysis.
CHAPTER 4. SIMULATION METHODOLOGY

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

Table 4.1: Simulated Machine Parameters. Baseline machine parameters used for experiments in this chapter.

4.3.1 Experimental Setup

Most experiments using our M5 simulator infrastructure were conducted using a baseline system configuration, shown in Table 4.1. 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 [43] of the remainder of the Simpoint plus an additional 100M instructions.

4.4 Pin Simulation Infrastructure

The Pin infrastructure is a dynamic binary instrumentation tool for the x86 architecture, written and maintained by Intel [60]. Pin implements an API that allows a user to add arbitrary code to any point of a program’s execution. This API allows
a user to add code before and after every instruction, basic block, and function in
the program. Pin also allows for selective instrumentation. For example, a user can
instrument only memory references, floating-point operations, and so on.

Pin is extremely useful for analyzing program characteristics. The analysis code
added by the user is referred to as a Pin tool. We created an Pin tool to measure
two items: the PVF of the architected register file, and a dynamic-dead analysis to
measure the fraction of instructions whose results are never read, read only by dead
instructions, or are logically masked. We use this to develop insights into program
vulnerability characteristics.

Our Pin tool is only capable of measuring PVF; it cannot measure vulnerability
factors at any other system layer. In fact, the functionality in our Pin tool is also repli-
cated in the M5 simulation infrastructure. We use the Pin infrastructure as a proof-
of-concept vehicle to establish that architecture-only simulation can yield meaningful
insight into program vulnerability characteristics. The Pin tool also has a substantial
performance advantage over the M5 microarchitectural simulation environment. Our
M5 infrastructure runs, on average, at under 100 kSIPS (100,000 simulated instruc-
tions per second). Our Pin tool, on the other hand, runs at between 4-5 MSIPS (4 to
5 million simulated instructions per second). This order-of-magnitude speedup allows
us to simulate realistic workloads to completion in a reasonable amount of time.

Figure 4.3 shows a code example that demonstrates the dependency tracking
needed in order to determine whether an instruction is dead or live. When an in-
struction commits, it is inserted onto the committed instruction queue \textit{instQ}. Certain
instructions (e.g., IO operations, branches) are determined to be live at the point of
commit; we first check for these conditions. We then loop through the instruction’s
source and destination operands. For each source operand, we find the producer instruction on the \textit{archRegs} array. We then increase the number of consumers for this producer instruction. For each destination operand, we look up the previous producer instruction on the \textit{archRegs} array and decrease that instruction’s live destination register count. The \textit{traverse()} function then checks two conditions to determine if the producer is dead. First, the producer must have no live destination registers; and second, any destination register must either have zero consumers, or only dead consumers. The \textit{traverse} function is then called recursively from the producer as well.

This code demonstrates our algorithm for marking instructions as dead. We note here that this is an example for illustrative purposes, and has been simplified from our actual algorithm. The code example presented only tracks dependencies through registers. Our algorithm tracks instructions through both registers and memory. Furthermore, we also account for many other features such as logic masking, variable-sized operands, statically dead instructions, and so on.

4.5 Summary

In this chapter, we introduced the simulation infrastructure we developed to evaluate the various vulnerability factors of the System Vulnerability Stack. We first provided background on ACE Analysis, the technique on which our implementation is based. We then described the changes necessary to use ACE Analysis to measure vulnerability factors at different system layers. We then described the two simulation infrastructures that we developed as part of this thesis. The first, based on the M5 simulator from the University of Michigan, is a full-system microarchitectural simulator capable of evaluating vulnerability factors at any level of the stack. The second,
Figure 4.3: Simplified dynamic-dead analysis code from our Pin simulation infrastructure. Once an instruction commits, it is inserted onto the post-commit \texttt{instQ} queue. We then check its liveness and loop through its source and destination operands, increasing the consumer count on source operands and decreasing the active destination count on destination operands. We then call \texttt{traverse()} to determine if the producing instruction is dead.
based on the Pin dynamic binary instrumentation tool from Intel, is capable of evaluating the PVF of any architectural resource within a program. We developed the Pin infrastructure to demonstrate the possibilities provided for by architecture-only PVF simulation of a workload. We also provided some concrete source code examples to illustrate the workings of our algorithms.

The remainder of this thesis consists of results obtained using these simulation frameworks. In the following chapters, we demonstrate the use of the vulnerability stack during software design, during hardware design, and at system runtime. We use our simulation infrastructure to highlight the possibilities and capabilities of the vulnerability stack in all three of these applications. As previously discussed, many of the benefits of the vulnerability stack stem from the ability to measure individual vulnerability factors for each layer of the system. Therefore, much of the work in the following sections is dedicated to measuring a vulnerability factor independently and establishing its behavior.
Chapter 5

Using the Stack for Software Design

In this chapter, we focus on the use of the vulnerability stack during the design of a software program. Therefore, our primary focus is on analysis of the Program Vulnerability Factor; its behavior and what this means for system vulnerability; and how to integrate and effectively use PVF during the software design phase.

To those ends, we begin the chapter with a discussion on measuring the PVF profiles of various architectural resources. We profile many programs to observe the behavior of PVF, and show how specific features in the source code affect the vulnerability of the resulting binary. Following this, we discuss how changes in the PVF of an architectural resource will ultimately translate into changes in the AVF of a hardware structure. We then investigate two means by which software developers can control their program’s PVF: algorithm implementation and compiler optimizations. Finally, we discuss the impact of input data on program vulnerability.
5.1 Analysis of the Program Vulnerability Factor

In this section, we examine the PVF profiles of several applications to generate insight into program reliability behavior [61]. 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 [62]. Walcott et al. examined the phase behavior of AVF and experimentally derived equations to predict AVF from microarchitectural state [48]; however, their models target a specific hardware implementation.

Hardware-independent program vulnerability analysis allows insight based purely on architectural state. This can, for example, allow software-based redundancy techniques to protect only highly vulnerable regions of code, and leave low-vulnerability regions unprotected for improved performance. Many software fault detection techniques (e.g., SWIFT [23]) more than double the dynamic instruction count while software recovery techniques can more than triple the instruction count [63]; therefore, we believe program vulnerability estimates can be a valuable tool in reducing the overhead of redundancy features.

5.1.1 Methodology

For these experiments, we use the M5 simulation infrastructure described in Chapter 4. (We forgo the simulation speed benefits of architecture-only simulation in order to fairly compare AVF and PVF values.) We examine the PVF 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 5.3 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 5.1 at the single early simulation points given by Simpoint analysis [64]; 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 [59] using the baseline system configuration described in Chapter 4.

5.1.2 Architectural Register File

Figure 5.1 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 generateMTF-Values 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
5.1.3 Architectural Integer ALU

Figure 5.2 shows the PVF of the Architectural Integer ALU for the same three benchmarks. The program phases that are evident in Figure 5.1 are also visible in Figure 5.2, 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.

5.2 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
Figure 5.1: The PVF of the architectural register file. Architectural Register File PVF for bzip2-source, mgrid, and equake.
Figure 5.2: The PVF of the architectural integer ALU. Architectural Integer ALU PVF for bzip2_source, mgrid, and equake.
behavior of an architectural resource impact the AVF behavior of the underlying hardware structure? For example, how much of the behavioral difference in Figure 2.1a can be explained by differences in PVF? In structures with no microarchitecture-level fault masking (e.g., structures where the HVF is always 1), all AVF variation will be caused by PVF variation. Conversely, in structures with a variable HVF, 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 PVF and HVF.

5.2.1 Structures with Architecture-Level Fault Masking

PVF explains most of the AVF variation in structures where architectural masking dominates microarchitectural masking (i.e., changes in PVF are of much greater magnitude than changes in HVF). This can occur when a structure’s HVF is constant or changes very slowly. 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 (cacheless) memory subsystems.

We ran correlation experiments on the Integer Register File to test this hypothesis; Figure 5.3 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 5.1 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
Table 5.1: Pearson correlation coefficients between register PVF and AVF. Correlation between Architectural Register File PVF and Physical Integer Register File AVF. The correlation is high for most benchmarks.

PVF variation. For benchmarks with low correlation (e.g., mesa, perlkmk_makerand, and vpr_route), a visual examination of the data (Figure 5.4) 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 5.2. Although the range of AVF values (not shown) differs between microarchitectural configurations, the correlation between PVF and AVF remains strong across microarchitectural variation, showing less than a 4 percentage point difference across all configurations.
Figure 5.3: Comparison of AVF and PVF for the register file. Physical Register File AVF (in gray) and Architectural Register File PVF (in black) for bzip2_program, ammp, and sixtrack.
Figure 5.4: Comparison of register AVF and PVF for benchmarks with low correlation. Register File AVF (in gray) and PVF (in black) for mesa, perlbmk_makerand, and vpr_route. The low correlation for these benchmarks is a result of minor fluctuations around a constant value.
Table 5.2: Register file AVF and PVF correlation across microarchitectures. The PVF-AVF correlation remains strong across these changes. Results for other benchmarks are similar.

5.2.2 Structures with Both Architecture- and Microarchitecture-Level Fault Masking

If a structure has a non-constant HVF, it will exhibit microarchitectural fault masking effects. Therefore, PVF behavior will only partially explain changes in AVF: the AVF of an execution unit, for example, 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 or latency. 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 5.3 shows the AVF-PVF correlation over all the benchmarks; Figure 5.5 depicts the results for
bzip2, 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 5.6 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 5.3 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.)

5.2.3 Structures with Microarchitecture-Level Fault Masking

Many modern microprocessors contain structures that do not directly correspond to an architectural resource. For example, the Reorder Buffer (ROB), a common hardware structure in current microprocessors, has no architectural equivalent in most instruction sets. Intuitively, we expect that the AVF of these structures will show only a very slight correlation to the PVF of any architectural resource. Most
Figure 5.5: Comparison of AVF and PVF for the ALU. Integer ALU AVF (in gray) and PVF (in black) for bzip2_program, galgel, and sixtrack.
Figure 5.6: Performance figures for several benchmarks. The CPI for bzip2, program, galgel, and sixtrack. CPI variation is one contributor to low correlation between AVF and PVF in the ALU.
Table 5.3: Pearson correlation coefficients for ALU PVF and AVF (Column 2) and PVF and AVF*CPI (Column 3). The PVF of the Integer ALU only partially explains its AVF variation, but the PVF and CPI together explain all the AVF variation.
changes in AVF will be driven by changes in the HVF of the structure. We examine the impact of HVF on AVF in Chapter 6.

The vulnerability stack allows us to quantify this correlation, but we can also take this intuition a step further: since ROB AVF is dominated by HVF, 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.

5.3 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. [51], 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.

5.3.1 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 –O3. 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 5.7a and 5.7b. 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 5.8, 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.

5.3.2 Compiler Optimizations: Scheduling

Compiler optimizations can also have a significant effect on PVF. Figure 5.9 and Figure 5.10 contain three gcc-generated versions of a loop from bzip2. Figure 5.9a shows the loop compiled using –O3. Figure 5.9b and Figure 5.10a used loop unrolling but without speculative scheduling (–O3 –funroll-loops –fno-sched-spec) (the figure is repeated for clarity). Finally, Figure 5.10b 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 5.9a, register t1 is both source and destination and is vulnerable for the entire
Figure 5.7: 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 5.8: 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.
loop body. In Figure 5.9b, register $t_3$ is the destination; there are sections of code where this register is not vulnerable (e.g., between instructions $b1$ and $a2$). In Figure 5.10b, 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 ($t_3$ and $t_5$) 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. On the other hand, the speculatively-scheduled loop (Figure 5.10b) executes only 0.1% faster than the unrolled loop (Figure 5.10a) 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 5.1 using compiler setting –O3 –funroll–loops as our baseline. The results are shown in Figures 5.11 and 5.12. 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 5.9 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
Figure 5.9: The effect of loop optimizations on PVF, Part 1. A loop from bzip2 compiled by gcc with optimizations: (a) -O3 and (b) -O3 -funroll-loops -fno-spec-sched. In this example, unrolling the loop results in a slight decrease in PVF (25.43% versus 26.56% for the original loop). However, the unrolled loop executes 15% faster, resulting in a 16% lower cumulative vulnerability.
Figure 5.10: The effect of loop optimizations on PVF, Part 2. A loop from bzip2 compiled by gcc with optimizations: (a) -O3 -funroll-loops -fno-spec-sched and (b) -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.
CHAPTER 5. USING THE STACK FOR SOFTWARE DESIGN

speculative scheduling in these benchmarks. Figure 5.12 shows that disabling in-
struction scheduling altogether 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 optimizations that we examined (–fsched–spec–load,
–fsched–interblock, –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. Un-
surprisingly, these transformations have inconsistent effects on vulnerability. This
points toward potential future work: compiler-based algorithms to increase reliability
at minimal performance cost.

5.4 The Effect of Input Data on Program Vulner-
ability

The behavior of a system depends upon the inputs it is given. Therefore, the accuracy
of vulnerability measurements depends on the ability to either: calculate vulnerability
using real input data; or generate representative vulnerability behavior using synthetic
inputs. In this section, we explore whether vulnerability measurements taken with
one set of inputs can predict the vulnerability behavior of a program with another
set of inputs [65].

This is an important question to answer, because it significantly impacts the
impact that PVF can have on software design. The previous sections, for example,
assumed that reducing the PVF of an architectural resource at compile time (for
instance, via a profiling step using synthetic inputs) would result in a lower PVF for
Figure 5.11: Results when disabling speculative scheduling. Percent change in PVF, runtime, and cumulative vulnerability when disabling speculative scheduling.
Figure 5.12: Results when disabling instruction scheduling. Percent change in PVF, runtime, and cumulative vulnerability when disabling instruction scheduling.
that resource at runtime. This assumption depends on input data, which is the only difference between profile- and run-time, having a consistent effect on PVF.

Input data affects program behavior in many ways. For example, input data may determine how many iterations of a loop are executed; or a program that accepts integer and floating-point data might call different functions based on the data type [66]. Many studies have examined how input data affects performance characteristics; however, most of the work has focused on the similarity of the execution profile of a binary with different inputs [67] [68] or on finding input-dependent branches [66] [69]. Typically, the goal of these studies is to determine whether a small input set can be used instead of a larger one. For fault tolerance, however, there are also other considerations: for example, the value of an input field which does not affect control flow can change the amount of logical masking present in the program. To our knowledge, there is no prior published study which examines the effect of input data on program reliability.

5.4.1 Methodology

Since the primary impact of input data is on program behavior, we examine the impact of input data on PVF. For our experiments, we use the Pin dynamic binary instrumentation framework described in Chapter 4 and implement a Pintool to measure the PVF of a program’s integer register file [60].

We perform our experiments on the SPEC CPU2006 benchmarks compiled for the x86_64 architecture. All benchmarks were simulated to completion. Our PVF instrumentation routines imposed an average slowdown of 500x, resulting in an average simulation speed of between 1 and 5 MIPS. This is a substantial slowdown relative to native execution, but remains orders of magnitude faster than microarchitectural
simulation which can run at 1-10 KIPS in industrial-grade cycle-accurate simulators.

5.4.2 Dependence on Input Data

Figure 5.13 shows that the PVF of 400.perlbench varies substantially when run with different input data; this is one of the most noticeably input-dependent programs that we tested. Similarly, Figure 5.14 shows that the PVF of 450.soplex varies when the input data is changed. However, our data show that these differences are predictable. Figure 5.15 presents the same data sorted by code trace (a single-entry, multiple-exit region of code). Each point on the x-axis represents a unique code trace as identified by Pin; the y-axis is the average PVF of that trace over the simulation. The center region of each figure contains traces that are touched by both inputs, while traces unique to one input are plotted on the left and right. The figure shows that most traces common to both inputs have approximately the same PVF regardless of the input; therefore, the PVF differences observed in Figures 5.13 and 5.14 primarily arise from differences in execution profile (i.e., traces unique to an input). Table 5.4 confirms this numerically for benchmarks with multiple inputs in the train dataset; there are only slight PVF differences between inputs for traces common to multiple inputs. Therefore, we hypothesize that representative inputs (e.g., inputs that exercise all code paths) will be able to accurately characterize program PVF and allow prediction of a program’s PVF with unknown inputs.

To test this hypothesis, we implement an algorithm to predict the PVF of a program that is given unknown input data. Our goal is to predict average PVF behavior rather than to calculate an exact PVF value. This is useful, for example, in systems that estimate vulnerability at runtime. Our current goal, however, is to demonstrate the feasibility of PVF prediction rather than to develop a low-overhead
CHAPTER 5. USING THE STACK FOR SOFTWARE DESIGN

Figure 5.13: Changes in PVF with different input data, plotted by time. Integer Register File PVF for 400.perlbench using input files diffmail and splitmail from the train data set.

Figure 5.14: Changes in PVF with different input data, plotted by time. Integer Register File PVF for 450.soplex using input files train and pds-20 from the train data set.
Figure 5.15: Changes in PVF with input data, plotted by code trace. Per-trace PVF for 400.perlbench and 450.soplex using multiple input files from their train data set. Therefore, we focus on accuracy at the expense of runtime considerations (e.g., performance, memory overhead). Our predictor uses a training run to generate a table of PVF predictions for each trace in a program. The table includes the start address of each trace and the average PVF of the trace over the training run. When the program is run with real inputs, we access the prediction table at the start of each new trace. If the table contains an entry for the trace, we use that value as the predicted PVF for the trace. If we do not find an entry, we retain the existing PVF prediction.

5.4.3 Results

We use the train data set to train our predictor for each SPEC benchmark; Table 5.5 shows the input we use for benchmarks with multiple inputs (we always choose the input that touches the most traces). We then test our predictor on the benchmarks’
Figure 5.16: *Mean absolute error of our prediction algorithm*. The absolute error in PVF of our predictor. Overall, we see a mean absolute error of under 0.06 for all benchmarks except 450.*soplex*. 
Table 5.4: Mean absolute difference in PVF for traces common to both input sets. The PVF difference between code traces seen by both inputs is small.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input 1</th>
<th>Input 2</th>
<th>PVF Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>diffmail</td>
<td>splitmail</td>
<td>0.013</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>program</td>
<td>combined</td>
<td>0.006</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>nicklas2</td>
<td>blunder</td>
<td>0.016</td>
</tr>
<tr>
<td>450.soplex</td>
<td>pds-20</td>
<td>train</td>
<td>0.010</td>
</tr>
<tr>
<td>473.astar</td>
<td>BigLakes1024</td>
<td>rivers1</td>
<td>0.011</td>
</tr>
</tbody>
</table>

ref data sets. Figure 5.16 shows the mean absolute error of our predictions for each benchmark. Overall, the predictor has a mean error of under 0.06 for all benchmarks except 450.soplex. For example, the predictor generates predictions for over 99% of traces executed by gcc.166, shown in Figure 5.17a, and accurately captures the PVF behavior of the application. Gcc.g23 (Figure 5.17b), on the other hand, encounters more unique traces; the predictor only generates predictions for 90% of the traces, resulting in stretches of inaccuracy (e.g., the phase between instructions 14.2B and 17.5B). However, the policy of retaining the previous prediction works well, so the predictor has a mean absolute error of only 0.03 on average across the benchmark.

The predictor has an extremely high error on 450.soplex (mean error = 0.24). This is because 67% of the traces encountered by the ref input were not encountered during training; the pds-20.mps input is not representative of the ref input. Figure 5.17c shows a portion of the execution of 450.soplex. The predictor does well until approximately 27B instructions into execution, when 450.soplex begins to encounter a high percentage of new traces. To recognize this condition, a runtime predictor can count the fraction of predicted traces. When this count dips below a pre-defined threshold, such as in this case or in the case of gcc.g23, the predictor can indicate a low confidence in its PVF estimates.
Table 5.5: Training input used for benchmarks with multiple inputs in the train data set. For benchmarks with multiple inputs in the train data set, we chose one input on which to train our predictor.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Training Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>splitmail</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>combined</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>nicklas4</td>
</tr>
<tr>
<td>450.soplex</td>
<td>pds-20</td>
</tr>
<tr>
<td>473.astar</td>
<td>BigLakes1024</td>
</tr>
</tbody>
</table>

A post-hoc analysis of 450.soplex (Figure 5.18) shows that the predictor’s accuracy would have been significantly improved had it trained on train.mps rather than pds-20.mps. This again demonstrates the need for representative input data; but shows that given this input data, the PVF of an application can be accurately assessed.

5.4.4 Conclusions on Input Data Dependence

In this section, we examined the dependence of program vulnerability on input data. Our data show that the primary cause of PVF difference between inputs is changes to a program’s execution profile (i.e., which code is executed). Changes in the PVF of a single trace are only a small contributor to changes in PVF. As a result, an input file that causes a program to touch all code regions can be used to accurately characterize the PVF of a program. This can be used to predict the PVF of the program when run with arbitrary inputs. Our predictor generates highly accurate predictions when trained on representative inputs. In addition, this implies that software designers can use PVF at compile time to accurately assess (and potentially improve) the vulnerability of their application when executed on a real system. This result is significant, as compile-time vulnerability assessment is one of the primary
Figure 5.17: *Real versus predictor PVF over time.* Real versus predicted PVF for `gcc_166`, `gcc_g23`, and `soplex_ref`. 
Figure 5.18: Comparison of input files for 450.soplex. The train.mps file covers more of the code traces touched by the ref.mps input file than does pds-20.mps; therefore, our predictor would perform better when trained with train.mps.

5.5 Summary

In this chapter, we examined the ability of the Program Vulnerability Factor to influence software design. In particular, we first examined the range of PVF variation for multiple software resources, including identifying source code regions and algorithms that contribute to higher PVF. We also examined how changes in an architectural resource’s PVF correlate with changes in a hardware structure’s AVF. This is important information for software designers, in order to understand the impact that PVF reduction will have. We also examined several practical methods to reduce PVF in software, including algorithmic changes and compiler optimizations. We demonstrated that these changes can have a significant impact on PVF. Finally, we demonstrated that changes in input data affect a program’s PVF. We found that these changes are generally caused by changes in the instruction flow, and the PVF
of a basic block remains roughly constant across input data variation.

As a result of our experiments, we believe that PVF can be an important tool for software developers and compiler architects to use in their effort to improve system reliability. We believe that compiler techniques in particular are an extremely fruitful area for future research, and will pursue these techniques in the future.
Chapter 6

Using the Stack for Hardware Design

In this chapter, we focus on the use of the vulnerability stack during the design of a microprocessor. Therefore, while we first focus on an analysis of the Hardware Vulnerability Factor, we also discuss a method to generate AVF estimates at design time, an important component of the hardware design phase, using PVF traces.

To those ends, we begin the chapter with a discussion on measuring the HVF of various hardware structure. We discuss various uses for HVF and demonstrate how HVF correlates to other easily-measurable processor statistics such as occupancy. Following this, we discuss how to generate a tight bound on AVF at design time using PVF traces. We also examine some of the design tradeoffs necessary when using PVF traces. We demonstrate that the use of PVF traces can lead to a significant reduction in simulation time, as well as allowing for the potential to substantially increase accuracy of design-time AVF estimates.
Figure 6.1: The AVF and HVF of the register file for equake and mgrid. In equake, a combination of pipeline stalls (high HVF) and dynamically-dead instructions (low PVF) cause its HVF to be a loose bound on AVF. This is not generally true; for example, the HVF of mgrid is a tight bound on AVF in certain places. It is impossible for a hardware designer to a priori predict this workload-specific behavior; a designer can only guarantee that AVF is less than HVF. Therefore, it is important to understand the causes of high HVF in a hardware structure.

6.1 Analysis of the Hardware Vulnerability Factor

Currently, architects use AVF to guide decisions such as where or when to add redundancy. For runtime reliability techniques (e.g., IQ squashing [24] or dynamic RMT [48]), this is appropriate. However, using AVF to to evaluate the impact of design-time microarchitectural decisions (e.g., structure sizing) is suboptimal because hardware designers have no control over a program’s PVF.

For example, Figure 6.1 shows the AVF and HVF of equake and mgrid. Equake’s HVF is a loose bound on its AVF, but mgrid’s HVF is a tight bound on its AVF in some places. At design time, a hardware designer does not know every program that will run on the hardware. It is impossible for a hardware designer to determine the precise AVF-HVF relationship of an unknown workload; a designer can only guarantee that AVF will be less than or equal to HVF. Thus, it is important to understand behaviors that cause high HVF, regardless of the AVF of a particular
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.

In this section, we measure the HVF of several hardware structures: the Issue Queue (IQ), Reorder Buffer (ROB), Load Buffer (LDB), Store Buffer (STB), and Integer Register File (RF). In the entry-based structures, we split each entry into multiple sub-fields and track iACEness and pACEness independently for each subfield. For example, an IQ entry contains several fields, including a destination register address and control information for the functional units. We measure the HVF of each field separately and calculate the weighted average to compute IQ HVF.

Figure 6.2 plots the HVF of the Reorder Buffer (ROB), Issue Queue (IQ), Load Buffer (LDB), and Physical Register File (PRF), of three typical benchmarks from the SPEC2000 suite, using the single early simulation points given by Simpoint analysis [64] and the baseline configuration of our M5 simulator infrastructure.

It is obvious from the graphs that the HVF profile of each structure depends significantly on the workload that is executing. For example, the ROB’s HVF looks very different when it is executing ammp than when it is running equake. However, there are obvious correlations between HVF profiles across structures when
the same workload is running. For example, all 4 structures show a similar high-HVF spike when running earthquake, although the duration and magnitude of the spike differ by structure. These correlations are due to hardware effects that impact the entire pipeline. High HVF phases are often the result of stalls (e.g., cache misses) in the microarchitecture. These stalls result in a structure that is full of instructions that cannot execute (or commit). High HVF can also result when the machine is being utilized to its fullest. When this occurs, structures are generally full of correct-path instructions with few flushes or stalls.

Low HVF phases (e.g., in earthquake) will result when the structure is relatively empty (for instance, due to 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.

### 6.1.1 Using HVF for Microarchitectural Exploration

In this section, we present the results of a microarchitecture exploration study performed using HVF. For all experiments, we use our M5 simulator infrastructure in the baseline configuration described in Chapter 4. All experiments were run using the SPEC CPU2000 benchmarks at the single early simulation points given by Simpoint analysis [64]. We use a 100M-instruction warmup window and a 100M-cycle cooldown window for all simulations [43].

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 6.3 shows the results averaged over all benchmarks. We can see that a 32-entry Store Buffer increases
Figure 6.2: The HVF of the Reorder Buffer (ROB), Issue Queue (IQ), Load Buffer (LDB), and Physical Register File (PRF) for ammp and equake. A structure’s HVF varies substantially across benchmarks, increasing when the structure is full of correct-path instructions, and decreasing when the structure is empty or contains many wrong-path instructions.
Figure 6.2: Continued from the previous page. The HVF of the ROB, IQ, LDB, and PRF for mcf.
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.

### 6.1.2 Using Occupancy to Approximate HVF

It is tempting to think of a structure’s HVF as simply its occupancy, especially since previous studies have shown a relationship between occupancy and AVF for many structures [48] [50]. However, occupancy and HVF differ: for example, bits in an occupied entry can be unACE. Furthermore, bits in an unoccupied entry (e.g., valid bits) can be iACE in some structures.

However, it is possible to use a structure’s occupancy as a heuristic for its HVF. Figure 6.4 demonstrates that committed instruction occupancy closely tracks HVF for the ROB and IQ across all benchmarks. Furthermore, if we look at occupancy over time within each benchmark, we find a very high (> 0.97) correlation between occupancy and HVF. Our results provide insight into previous work that shows that a high fraction of AVF variation is explained by structure occupancy [48] [49]. Our
CHAPTER 6. USING THE STACK FOR HARDWARE DESIGN

Figure 6.3: 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.
results indicate that the relationship between occupancy and AVF is due to the relationship between occupancy and HVF. Significant deviations from this relationship are likely due to program-specific (PVF) effects.

Figure 6.4 shows that, by itself, occupancy is an extremely conservative predictor of HVF. For example, the IQ HVF is approximately 65% of the IQ occupancy on average; for certain benchmarks, the average HVF is only 50% of the average occupancy. This result has an important implication for AVF measurement in complex hardware structures. If a techniques assumes that all bits in a particular entry behave the same, it can significantly overestimate AVF because it is using occupancy to estimate HVF. (For example, many techniques bound AVF by assuming that all bits in the instruction queue entry of a live instruction are live (e.g., [50] [4] [8]).) While this upper-bound behavior is desirable in a design setting, our results indicate that significantly more detail is needed to generate accurate AVF estimates.

6.2 Bounding AVF During Hardware Design

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. Failure rates from transient faults are typically estimated using a combination of raw fault rate estimates (based on process technology parameters) and AVF simulation, verified by post-silicon radiation-beam experiments [7] [26]. In this section, we discuss how to use the vulnerability stack to improve the AVF simulation step of this process. Note that all HVF and PVF measurements are performed using full ACE Analysis (i.e., not calculated using heuristics).
Figure 6.4: The relationship between HVF and committed instruction occupancy across benchmarks for the ROB and IQ. 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 0.98 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) using an architecture-only simulator such as Pin [60]. 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.

This methodology has several advantages over standard AVF analysis. First, since benchmarks are typically used over multiple chip generations, PVF traces can be re-used over multiple processor designs. Furthermore, removing PVF analysis from the critical path of the processor’s design reduces the amount of time needed for AVF simulation. 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 [7].

Figure 6.5 shows that, in our experiments, the use of PVF traces resulted in an approximately 2x reduction in the time required for microarchitectural simulation relative to standard AVF 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 the simulator needs to track. Standard AVF analysis defers analysis of an event until after the corresponding instruction has been analyzed in a post-commit window. This post-commit window can be tens of thousand instructions in length [7]. Therefore, the simulator must maintain large event history buffers within each hardware structure. When using PVF traces, we defer analysis of an event only until after HVF
analysis, which is typically completed immediately after the corresponding instruction commits. This significantly reduces the size of the history buffers and yields a much smaller memory footprint for the simulator itself.

In Section 6.2.1, we describe the generation of PVF traces and examine some of the required tradeoffs. In Section 6.2.2, we examine the impact of incorporating these traces into HVF analysis.

### 6.2.1 Capturing PVF Traces

In principle, capturing a PVF trace of a program requires recording the PVF of all architecture-visible state. We reduce the amount of state significantly by exploiting the behavior of ACE Analysis. ACE Analysis determines a bit’s ACEness by tracking events to that bit: a bit is ACE if it is read by an operation 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.

For each benchmark, we run PVF analysis and record a trace of all architecturally dead events tracked through registers and memory. Architecturally dead events arise from effects such as static and dynamically dead instructions, transitively dead instructions, static and dynamic logical masking, and transitive logical masking, among others. During HVF simulation, the trace is read and the events are mapped by the simulator to the corresponding microarchitectural event(s).

We store PVF traces as gzipped ASCII text files. This both enables a human-readable trace format as well as minimizes the disk space required for the traces. Typical traces range from 1-100 megabytes of disk space per 100 million program instructions. (Trace size varies based on the number of dead events in the workload.) Figure 6.6 presents the trace sizes we recorded for the SPEC2000 benchmark.
programs. Benchmarks with more dead events (e.g., lower PVF values) have larger traces.

Figure 6.7 shows representative snippets from a PVF trace of ammp. We record a line in the trace for each instruction in the program that generates a dead event. For each dead event, we record: the instruction number that generated the event; whether this was a dead read (r) or a dead write (w); whether this event was to a register (R) or memory (M); the register index or memory address; and the liveness of each byte within the read or write. Finally, one instruction can generate multiple dead events; these are separated by a colon (:). For example, the entry:

316 w R 6 00001000: r M 4831380408 0000:

records that instruction 316 generated a write to register r6, of which only the 5th byte was live. The instruction also performed a 4-byte read to memory, and all bytes within the read were dead.

Increasing the accuracy of PVF analysis will lead to more dead events and a larger trace, but will also lead to a tighter bound on AVF. Figure 6.8 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.)

6.2.2 Results

Figure 6.9 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
Using PVF traces led to an approximately 2x reduction in simulation time. This is primarily due to the ability to perform separate PVF calculations using architecture-only simulation.

PVF traces typically required between 0.2 and 4 bytes per instruction. This leads to trace sizes of between 20-400MB per 100M instructions. The trace size increases with the number of dead instructions in the benchmark.
2 w R 2 00000000: r R 2 00000000:
5 r R 18 11110000:
6 r R 18 10000000:
14 w R 6 00000100: r M 4831380408 0000:
15 r R 11 11110000:
... 
316 w R 6 00001000: r M 4831380408 0000:
... 
2004 r R 68 00000000:
2007 w M 5392507088 00000000: r R 41 00000000: r R 12 00000000:
2019 w M 5392507088 00000000: r R 41 00000000: r R 12 00000000:

Figure 6.7: Snippets from a uncompressed PVF trace. Each line represents one instruction and lists: the unique ID of the originating instruction; whether the event was a write (w) or a read (r); whether the event was to a register (R) or memory (M); the register index or memory address; and the liveness of the bytes in the event. Multiple dead events from one instruction are separated by a ":".

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 [7].

Our results extend these findings to all structures that we tested and to both first-level and transitive logic masking. This behavior makes intuitive sense, since many bits in the IQ affect an entire instruction and are ACE if even a portion of the instruction’s result is ACE. However, we found this to also be true in the register file: dead instructions contribute substantially more unACEness than logically masked values.

Figure 6.10 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
Figure 6.8: The granularity of PVF analysis versus trace size. PVF traces can be used to tighten the AVF bound during hardware design. Reducing the accuracy of the PVF analysis results in many dead events incorrectly being marked as live. This is accompanied by a sometimes significant reduction in trace size. We show results for PVF analysis at different granularity: word (2 byte), longword (4 byte), and quadword (8 byte), normalized to byte granularity.
significant reductions in the AVF bound: 25% for the IQ, 27% and 30% for the Load and Store Buffers, and 22% for the ROB.

### 6.3 Summary

In this chapter, we examined the ability of the vulnerability stack, and in particular the Hardware Vulnerability Factor, to influence the process of hardware design. We first analyzed HVF across a variety of structures and benchmarks. We then demonstrated the use of HVF to avoid bias from non-representative workloads during microarchitectural exploration, and determined that a structure’s occupancy is a conservative predictor of its HVF.

We then demonstrated the use of the vulnerability stack to generate AVF estimates during hardware design via PVF traces. The use of PVF traces results in a significant speedup of AVF computation, and can reduce time-to-market for new products. PVF traces can also allow for increased accuracy, since the PVF computation is not in the critical path of processor design, and therefore can afford more simulation cycles to improve accuracy.

As a result of our experiments, we believe that HVF and the vulnerability stack can be important tools for hardware designers and chip architects in their efforts to improve processor designs, estimate vulnerability, and ultimately improve system reliability.
Figure 6.9: Sources of unACEness in PVF traces. There are many sources of unACE events that can be captured in PVF traces. On average, 57% of unACE events are due to dead instructions (static and dynamic), while 33% and 9% of unACE events are due to first-level and transitive logic masking, respectively. Because PVF traces are generated offline, we can add expensive features such as detection of transitive logic masking with no impact on simulation speed during hardware design.
Figure 6.10: The effect of incorporating PVF traces into HVF simulation. Incorporating PVF traces tightens the AVF bound by 60% in the register file, 25% in the Issue Queue, 30% in the Store Buffer, 30% in the Load Buffer, and 24% in the Reorder Buffer.
Chapter 7

Using the Stack for Runtime Analysis

Chapter 5 showed that PVF can be used during software design to statically reduce vulnerability at runtime. Chapter 6 demonstrated that components of the vulnerability stack, in particular HVF, can be used to improve the hardware design cycle. In this chapter, we demonstrate that the vulnerability stack can also be used during the software design cycle 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., [48]). AVF monitoring can be used in conjunction with redundancy techniques (e.g., [70], [71], [72]) to provide highly-effective redundancy with low overhead.

Previous runtime AVF estimation techniques fall into two general categories: training-based predictors [48] [49]; and runtime-only predictors [50] [4]. The two
training-based predictors develop a formulation between hardware statistics that can be measured at runtime and AVF, based on a set of training benchmarks. 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 [48]. Similarly, Duan et al. use Boosted Regression Trees to correlate AVF with processor statistics and extend this to predict correlations across microarchitectural changes [49].

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 [50]. The technique proposed by Li et al. uses simulated fault injection tracked via error bits attached to each hardware structure [4]. 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.

7.1 Methodology

The predictors discussed above each have their own strengths and weaknesses. However, our key observation is that none of the prior techniques include any mechanism for the programmer to influence the AVF computation; they 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 applications are memory-intensive yet extremely fault-tolerant [54]. As a result, every
predictor discussed so far will almost certainly overestimate the AVF in this situation.

Similarly, software-based redundancy techniques such as SRMT [73], SWIFT [23], or PLR [74] 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 can address 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 a PVS register for the integer register file and one for the floating-point register file, and a PVS register for instruction-based structures, which stores 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.

### 7.2 PVF Prediction via Software Profiling

Chapter 5 shows that in the SPEC CPU2006 suite, most changes in register PVF due to input data stem from changes in execution profile, and that the PVF of a basic
CHAPTER 7. USING THE STACK FOR RUNTIME ANALYSIS

Figure 7.1: *Results of our PVF prediction algorithm.* The x-axis plots functions that comprise over 90% of the execution of the first Simpoint of the *ref* input of the SPEC CFP2000 benchmarks. The y-axis plots the fraction of dead instructions using both the *train* and *ref* input sets. For a given function, the difference in PVF between input sets is generally less than 0.04, with the exception of several short functions in *mesa* and one in *wupwise.*

block is relatively insensitive to input data. 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 7.1 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 restoring 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 run-time overhead. First, we only update the PVS registers if the difference between the old and new predictions is more than 0.03. 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.

### 7.3 HVF Monitor Unit

To estimate HVF at runtime, we implement an HVF Monitor Unit (HMU) for each structure. At a fixed interval (e.g., 512 cycles), 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 the added ROB bits. If the bit for a given structure is set, an AVF counter is incremented. When the counter reaches 8, 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 is divided by the total number of samples (2048) to yield
an AVF estimate for the previous interval.

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. For instance, the trigger event in the Issue Queue is an instruction issue, while the trigger in the Store Buffer is a writeback. However, 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 can be shared across all structures). By choosing power-of-two interval sizes, we implement all necessary arithmetic operations as either bit-shifts or integer addition. This makes our overhead is similar to that of other proposed techniques (e.g., [4]).

We expect that using only one HMU per hardware structure will sometimes result in an inaccurate estimate of HVF. This is because the HMU will estimate will converge to committed instruction occupancy; as discussed in Chapter 6, this is a loose bound on HVF. The estimate can be improved by treating individual fields within a structure as separate structures, and implementing separate HMU units to measure each field. For example, the IQ’s Displacement field, which stores instruction-carried constants such as load/store offsets and arithmetic immediate values, can be evaluated separately from the rest of the IQ.

7.4 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 \textit{IQ-3} uses 3 HMUs in the IQ, one for the Displacement field, one for the Branch Control field (which is used only for control instructions), and one for the remainder of the fields. We identified these two fields as the most likely to benefit from separate measurement. Similarly, predictor \textit{IQ-All} splits the IQ into 8 sub-structures. Obviously, \textit{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 \textit{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 difference between our AVF estimates and the actual AVF.

Figure 7.2 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 \textit{applu}). Table 7.1 confirms this result; the mean absolute error (MAE) for many benchmarks is over 10, which we deem unacceptably high. 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 \textit{IQ-3} and \textit{IQ-All} predictors are shown in Figure 7.3 and Table 7.2. As expected, \textit{IQ-3} substantially improves its HVF estimates over the baseline predictor, which translates into better AVF predictions. \textit{IQ-3} has a 3.6 average MAE across all benchmarks and only one benchmark has an MAE higher than 8. \textit{IQ-All} further improves the estimate, reducing the average MAE to just 2.7, and only one benchmark has an MAE higher than 6.

An interesting result is that a few benchmarks have higher MAE with the \textit{IQ-All}
Table 7.1: Mean Absolute Errors (MAE) of the predicted AVF relative to the actual (measured) AVF. Our predictor often has a high MAE because the HVF Monitor Unit treats each entry as a single field, resulting in HVF overestimates. This is similar to the behavior of other techniques (e.g., [4]), and points to the need for a more accurate HVF assessment in complex structures such as the IQ.

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

predictor than with IQ-3. As discussed in Section 7.2, 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 acceptable limits. Based on these results, we suggest that predictor IQ-3 achieves a good balance between overhead and accuracy.
### Chapter 7. Using the Stack for Runtime Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>IQ-3</th>
<th>IQ-All</th>
<th>Benchmark</th>
<th>Baseline</th>
<th>IQ-3</th>
<th>IQ-All</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>18.6</td>
<td>8.5</td>
<td>2.5</td>
<td>lucas</td>
<td>9.2</td>
<td>4.2</td>
<td>4.4</td>
</tr>
<tr>
<td>applu</td>
<td>19.6</td>
<td>7.7</td>
<td>1.5</td>
<td>mcf</td>
<td>6.5</td>
<td>2.1</td>
<td>5.4</td>
</tr>
<tr>
<td>apsi</td>
<td>5.4</td>
<td>1.9</td>
<td>0.9</td>
<td>mesa</td>
<td>6.2</td>
<td>3.2</td>
<td>0.7</td>
</tr>
<tr>
<td>art110</td>
<td>7.1</td>
<td>6.1</td>
<td>6.6</td>
<td>mgrid</td>
<td>3.2</td>
<td>2.3</td>
<td>3.0</td>
</tr>
<tr>
<td>crafty</td>
<td>0.4</td>
<td>0.8</td>
<td>1.5</td>
<td>perlbgcolor</td>
<td>3.3</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td>eon rushmeier</td>
<td>6.3</td>
<td>3.4</td>
<td>0.5</td>
<td>sixtrack</td>
<td>24.1</td>
<td>6.8</td>
<td>4.5</td>
</tr>
<tr>
<td>equake</td>
<td>6.0</td>
<td>3.8</td>
<td>3.0</td>
<td>swim</td>
<td>0.9</td>
<td>3.5</td>
<td>5.2</td>
</tr>
<tr>
<td>facerec</td>
<td>10.0</td>
<td>4.9</td>
<td>4.0</td>
<td>twolf</td>
<td>3.8</td>
<td>1.4</td>
<td>0.7</td>
</tr>
<tr>
<td>galgel</td>
<td>5.3</td>
<td>3.0</td>
<td>1.6</td>
<td>vortex2</td>
<td>6.0</td>
<td>3.8</td>
<td>0.7</td>
</tr>
<tr>
<td>gcc,166</td>
<td>2.9</td>
<td>1.6</td>
<td>4.6</td>
<td>vpr_route</td>
<td>5.3</td>
<td>2.5</td>
<td>0.7</td>
</tr>
<tr>
<td>gzip,graphic</td>
<td>2.0</td>
<td>2.7</td>
<td>3.7</td>
<td>wupwise</td>
<td>4.1</td>
<td>2.6</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 7.2: Mean Absolute Errors of improved predictors that treat the IQ as multiple structures. Baseline is the IQ predictor from Table 7.1. IQ-3 treats the two most vulnerable fields in the IQ as separate structures, and IQ-All treats all 8 fields in the IQ as separate structures. Overall, IQ-3 requires less hardware than IQ-All but, as expected, delivers substantially better AVF estimates than the baseline predictor.
Figure 7.2: Results of our AVF predictor. A visual depiction of the results from Table 7.1 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).
CHAPTER 7. USING THE STACK FOR RUNTIME ANALYSIS

Figure 7.3: Predicted and measured AVFs of the Baseline, IQ-3, and IQ-All predictors. Most benchmarks resemble applu and see a steady improvement of AVF estimates. For a few benchmarks such as gcc, swim, and mcf, IQ-All gives a slightly worse estimate than IQ-3, primarily due to either PVF or HVF underprediction as discussed in Section 7.4. However, the results are still within acceptable tolerances.
Chapter 8

Summary and Conclusions

In this thesis, we have introduced, developed, and analyzed a novel method to assess the vulnerability of a high-performance microprocessor to transient faults. Our goal was to provide a method which could exploit the abstractions present in modern computer systems (e.g., the Instruction Set Architecture) to significantly expand the scope of transient-fault analysis. Currently, transient-fault analysis is primarily limited to chip architects and hardware designers. By exploiting higher-level abstractions, we can provide a framework for program architects, software designers, and operating system engineers (among others) to reason about and more fully participate in vulnerability analysis. The hope is that this increased engagement will ultimately yield substantial benefits for overall system reliability.

8.1 Summary of Research

In this section, we summarize the work accomplished through this thesis, with a focus on the novel concepts and ideas that we have introduced, developed, and analyzed.
8.1.1 The System Vulnerability Stack

The major novel contribution of this research was the introduction, development, and analysis of the System Vulnerability Stack. The Vulnerability Stack builds on existing work, primarily the Architectural Vulnerability Factor (AVF), but extends this method by calculating a vulnerability factor at every level of the system stack. Each vulnerability factor is independent of all others in a system. These vulnerability factors can be used on their own to measure the inherent vulnerability of a given system layer. For example, the Program Vulnerability Factor measures the microarchitecture-independent vulnerability of a software program. In addition, these vulnerability factors can be combined to yield an overall system vulnerability factor (e.g., AVF).

As part of this thesis, we developed the concepts and terminology that underlie the system vulnerability stack. For example, we defined the terms $pACE$, $iACE$, and $nonACE$, to describe the possible results of a fault in a given system layer. We then presented a precise mathematical formulation that captures these concepts and allows the calculation of a vulnerability factor for each layer. We also presented a formula to handle systems with multiple software layers. We described the concept of a multi-exposure bit: a bit in which a fault will create multiple visible faults in a higher system layer. We presented an exact formulation to calculate the vulnerability of a multi-exposure bit. Finally, we described how to use the vulnerability stack to compute system-level soft error rates.

8.1.2 Vulnerability Stack Simulation Framework

As part of this thesis, we developed and implemented a simulation framework to evaluate system vulnerability using the concepts of the vulnerability stack. Our framework
is built on top of the previously-developed M5 simulation environment. In particular, we performed our evaluations on the detailed out-of-order CPU model (O3 CPU) in the M5 simulator. This CPU model is loosely based on an Alpha 21264 microarchitecture, scaled up to reasonable parameters. We used this simulation environment for the majority of our evaluations in this thesis.

Our simulation framework allows separate evaluation of each component of the vulnerability stack such as HVF and PVF. We implemented hooks to evaluate the HVF of any microarchitectural structure. In this thesis, we measured the HVF of structures including: the Reorder Buffer, Issue Queue, Load Buffer, Store Buffer, and Physical Register File. We also implemented the ability to measure the PVF of any architectural resource. In this thesis, we measured the PVF of the Register File, as well as measuring the fraction of dynamically-dead instructions in an execution stream. Finally, our framework can combine PVF and HVF components to generate AVF measurements for any hardware structure in the system.

Our simulator is also capable of reading and recording PVF traces. This capability allows a benchmark’s PVF to be measured once, and the results re-used across multiple microarchitectural simulations. This results in a significant (2x) reduction in microarchitectural simulation time.

Finally, we have also implemented a proof-of-concept simulation framework using the Pin dynamic binary instrumentation environment on the x86 ISA. Dynamic binary instrumentation is essentially an architecture-only simulation environment. (There is no microarchitecture model in this environment.) Therefore, Pin has a significant speed advantage over microarchitecture simulators such as M5. Within Pin, we implemented a proof-of-concept PVF-only vulnerability framework to demonstrate: (a) that accurate PVF evaluation can be performed using architecture-only simulation;
and (b) the significant speedups achievable by using architecture-only simulation. For instance, our Pin simulation environment ran at 4-5 MIPS. By contrast, typical M5 speeds range from 10-100 KIPS.

We used the Pin simulation environment to evaluate the input dependence of PVF. By using Pin, we were able to execute the SPEC CPU2006 benchmarks to completion using the train input sets. This allowed us to get an accurate PVF profile of the entire execution of the application.

8.1.3 The Program Vulnerability Factor

As part of the development of the vulnerability stack, we defined and developed the Program Vulnerability Factor, a metric to measure the inherent (microarchitecture-independent) vulnerability of a program. The Program Vulnerability Factor is one of the most useful components of the vulnerability stack, primarily because it can be used by software architects and program designers to make reliability decisions during the software design process. It is possible to measure the PVF of any architecturally-visible (ISA-defined) resource in a program, such as an architected register or register file, instruction memory, or the various ISA-defined operation (addition, subtraction, etc). We demonstrated that the PVF of the register file generally correlates closely with the AVF of the underlying physical register file. The PVF of an operation such as addition shows lower correlation with the AVF of the underlying execution unit, but the PVF still captures the architecturally-defined portion of the execution unit’s vulnerability.

Finally, we demonstrated that the PVF of a program can be affected significantly by the choice of algorithm or compiler switches. In particular, we used three different quicksort implementations to demonstrate that the choice of algorithm had
a significant impact on the reliability of the overall system. We also demonstrated that compiler optimizations can have a significant effect on the PVF of a program. Current compiler optimizations, which are targeted at performance improvements, do not show a consistent effect on programs. However, this evaluation demonstrates the potential that future compiler optimizations can use PVF as a metric to target system reliability improvement.

8.1.4 The Input Dependence of Program Vulnerability

At compile time, only synthetic (training) input data is available. For PVF to be useful at compile time, we must be able to use the training data to predict a software’s PVF when using real input data. Thus, as part of our study, we examined the input dependence of program vulnerability. We used our Pin simulation environment to determine whether program PVF with synthetic inputs can be used to predict PVF with real inputs.

Our results indicated that some programs see significant changes in PVF when the input data is changed. However, most of these changes are due to differences in the execution profile of the program (i.e., changes in the order and frequency with which basic blocks are executed). When we examine the PVF of a single basic block or code trace (a single entrance / multiple exit region of code), we find that a trace’s PVF is very similar between input data sets. We used this knowledge to design a simple predictor that builds a PVF profile of a program by code trace. We built this profile using the train inputs, and used it to predict the PVF of the SPEC CPU2006 suite when executed with reference inputs. We showed that this simple predictor has high accuracy, predicting the PVF of most benchmarks to within a few percent accuracy.
8.1.5 The Hardware Vulnerability Factor

Another primary component of the vulnerability stack is the Hardware Vulnerability Factor, a metric to measure the inherent vulnerability of a hardware structure. The HVF of a hardware structure is the sum of the iACE and pACE bits within that structure, divided by the product of the total number of bits in the structure and the number of cycles in the measurement window.

We showed that the HVF of a hardware structure varies greatly based on how the workload uses the structure. However, it is important to note that HVF is not impacted by workload-specific ISA-visible effects. We demonstrated that the HVF of a hardware structure be used during microarchitecture exploration, for example, in order to avoid bias from workload-specific effects. Finally, we also showed that committed instruction occupancy is highly correlated with HVF in several structures, but is conservative as a measure of HVF.

8.1.6 PVF Traces

One of the important goals of our work was to improve processor design methodology. A key aspect of processor design is the estimation of AVFs for each hardware structure. These AVF values are used, in conjunction with particle flux measurements, to calculate the overall failure rate of the processor due to transient faults. Generation of these AVF values in simulation can be difficult and time-consuming; it requires a significant investment in infrastructure as well as hundreds of simulation runs to generate average AVFs.

The vulnerability stack can improve this aspect of processor design by allowing for significant reuse of simulation effort. We proposed the use of PVF traces to measure and record the vulnerability of a benchmark program. A PVF trace can be
used in conjunction with HVF simulation to generate AVF estimates for a hardware structure.

Benchmarks are typically used over multiple chip generations; therefore, PVF traces will likely be re-used over multiple processor designs. Furthermore, by removing PVF analysis from the critical path of the processor’s design, we can reduce the amount of time needed for AVF simulation. In our experiments, the use of PVF traces resulted in an approximately 2x reduction in simulation time relative to standard AVF analysis. This also allows us to increase the accuracy of PVF analysis without increasing simulation time during the processor design phase. For example, we can assess effects such as transitive logic masking which take significant simulation time and are often omitted from AVF simulation [7].

8.1.7 The Program Vulnerability State and HVF Monitor Unit

The final major contribution of this thesis is the development and analysis of the Program Vulnerability State, and the related HVF Monitor Unit.

The Program Vulnerability State is a set of registers that allows hardware and software to communicate vulnerability information at runtime. These registers are an architectural extension and can be virtualized, allowing multiple layers of software to participate in the vulnerability process. We proposed using one PVS register for each of: the integer register file, the floating-point register file, and for instruction-based structures. A program updates these registers with its estimated vulnerability for each of its architectural resources.

The HVF Monitor Unit is a microarchitectural structure that allows HVF estimation of a wide variety of hardware structures. The HMU works in conjunction with
the PVS to allow a system to calculate AVF estimates at runtime. In this thesis, we developed the microarchitecture for this structure and assessed its implementation feasibility and accuracy across several large hardware structures.

8.2 Discussion

In this section, we discuss a limitation of the work as presented so far as well as the potential future applications and uses that are enabled by the vulnerability stack. This is helpful in order to give the reader some perspective of the potential future utility of this work.

8.2.1 Limitations

As discussed in Chapter 3, a multi-exposure bit is a pACE bit which produces multiple visible faults at the next system layer. In this thesis, we presented an Equation 3.4 to reason about these bits. Solving this equation to calculate the vulnerability of a multi-exposure bit involves modeling multiple simultaneous faults in some system layer. In this thesis, we presented a method to produce an upper bound on the vulnerability of a multi-exposure bit, but not to solve it precisely. We believe the impact of these bits to be small, and that this upper bound will suffice in most circumstances. However, a method to precisely evaluate the vulnerability of a multi-exposure bit might be useful in some circumstances.

In general, modeling the behavior of multiple faults in a system is an intractable problem. However, we can generate additional information that may make this problem tractable for the specific case of multi-exposure bits. For example, we can run an HVF simulation and record a fault vector for each multi-exposure bit. This vector
contains which ISA bits are corrupted by the given multi-exposure bit. We can then run a PVF simulation using these fault vectors, and determine the PVF of each fault vector. This uniquely determines the PVF of the underlying multi-exposure bit in hardware. Although this simulation would be slow, we believe it would be feasible, and would enable us to precisely evaluate the vulnerability of a multi-exposure bit.

8.2.2 Potential Future Applications

To give the reader a general sense of the potential for this work, we discuss three possible uses of the vulnerability stack. These uses primarily result from the abstraction the vulnerability stack provides that is not present in current techniques to estimate vulnerability.

Compiler Algorithms to Minimize PVF

Perhaps the most compelling use of the vulnerability stack is its potential to improve software reliability, for example by enabling a compiler to assess and reduce PVF. This process can lead to the development of compiler transformations that are targeted at reducing PVF. For example, the loop unrolling example given in Chapter 5 can be automated in the compiler. The compiler can then measure the PVF of both the original and transformed code, and implement the code that has the lowest PVF.

We believe this to be a fertile area for future study, and one of the primary benefits of the vulnerability stack.

Evaluating the Impact of Other Hardware Faults

Another benefit of the vulnerability stack is that it separates the underlying fault model (e.g., a single bit flip in the hardware) from the program-level effects of the
fault (e.g., the PVF of the fault). Potentially, this can allow for an evaluation of other underlying fault types using the same basic methodology. For example, it may be possible to evaluate the effects of hard (permanent) faults using a fault-specific evaluation technique which categorizes the effects of the hard fault as iACE, pACE, or nonACE. The results of this analysis can then be used in a standard PVF analysis to determine the outcome of the program in the presence of a hard fault.

To date, there have been few system-level hard fault models that have gained wide acceptance in the architecture community. We believe the vulnerability stack has the potential to allow for a general-purpose hard fault model that can be of use to the computer architecture community.

Enabling System Recovery

Finally, we believe that another area in which the vulnerability stack can have influence is that of system recovery. As discussed above, the vulnerability allows for separation of a fault’s cause from its effects. In particular, the stack defines a program-visible fault by its effects on the ISA interface. This can allow software to focus on correction and recovery techniques based on these ISA effects, rather than requiring software to develop unique recovery algorithms for each individual fault class. This is especially useful if hardware can consolidate multiple fault classes into similar ISA-visible effects.

A concrete example of this can be found in our prior work that proposed a full taxonomy of the possible outcomes of an error in a system [75]. In this work, we classify errors by their severity. We distinguish errors that are fatal to the system (Globally Uncorrectable Errors) from errors that are only fatal to a process (Locally
Uncorrectable Errors) or that are correctable by a process (Locally Correctable Errors). The difference in these classifications is the level of fault containment attained by the hardware. We demonstrated that this type of reasoning can lead to concrete benefits for a system’s reliability and availability, by encouraging architects to adopt techniques (such as data poisoning) that reduce error severity even if they do not reduce system AVF. By encouraging hardware and software designers to focus on fault effects at the ISA boundary, we believe that the vulnerability stack can further expand this type of thinking, and ultimately improve systems’ recovery capability.
Bibliography


[20] C. Slayman, “Cache and memory error detection, correction, and reduction tech-
niques for terrestrial servers and workstations,” Device and Materials Reliability,


bting in microprocessors: Myth or necessity?” in PRDC ’04: Proceedings of
the 10th IEEE Pacific Rim International Symposium on Dependable Computing

Software implemented fault tolerance,” in International Symposium on Code
Generation and Optimization (CGO ’05), 2005.

reduce the soft error rate of a high-performance microprocessor,” in International

“Understanding the propagation of hard errors to software and implications for
resilient system design,” in International conference on architectural support for
programming languages and operating systems (ASPLOS-XIII), 2008.

POWER6 processor,” IBM Journal of Research and Development, vol. 52, no. 3,


