GemV: A Validated Micro-architecture Vulnerability Estimation Tool

by

Srinivas Karthik Tanikella

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved December 2016 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Rida Bazzi
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

Several decades of transistor technology scaling has brought the threat of soft errors to modern embedded processors. Several techniques have been proposed to protect these systems from soft errors. However, their effectiveness in protecting the computation cannot be ascertained without accurate and quantitative estimation of system reliability. *Vulnerability* – a metric that defines the probability of system-failure (reliability) through analytical models – is the most effective mechanism for our current estimation and early design space exploration needs. Previous vulnerability estimation tools are based around the Sim-Alpha simulator which has been to shown to have several limitations. In this thesis, I present gemV: an accurate and comprehensive vulnerability estimation tool based on gem5. Gem5 is a popular cycle-accurate micro-architectural simulator that can model several different processor models in close to real hardware form. GemV can be used for fast and early design space exploration and also evaluate the protection afforded by commodity processors. gemV is comprehensive, since it models almost all sequential components of the processor. gemV is accurate because of fine-grain vulnerability tracking, accurate vulnerability modeling of squashed instructions, and accurate vulnerability modeling of shared data structures in gem5. gemV has been thoroughly validated against extensive fault injection experiments and achieves a 97% accuracy with 95% confidence. A micro-architect can use gemV to discover micro-architectural variants of a processor that minimize vulnerability for allowed performance penalty. A software developer can use gemV to explore the performance-vulnerability trade-off by choosing different algorithms and compiler optimizations, while the system designer can use gemV to explore the performance-vulnerability trade-offs of choosing different ISAs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A soft error is a transient bit flip caused by external radiation, alpha particles, neutrons, and cosmic rays Baumann (2005). With transistor technology scaling, soft errors are becoming an important design concern Dixit and Wood (2011). Soft error rate is increasing with decreasing feature size and supply voltage Martinez-Alvarez *et al.* (2012). In modern embedded systems, reliability is especially important due to aggressive dynamic voltage and frequency scaling Mahatme *et al.* (2013).

Many techniques have been proposed to protect embedded processors against soft errors Nicolaidis (2011); Lee *et al.* (2011). However, soft error protection is not cheap, and also not always effective. Traditionally, fault injection has been used to evaluate the effectiveness of protection schemes against soft errors Entrena *et al.* (2012). Since statistical fault injection is time consuming Nguyen and Yagil (2003), previous works have used targeted fault injection to estimate the failure rate Michel *et al.* (1991); Alkhalifa *et al.* (1999). However, estimating failure rate using targeted fault injection is very difficult to set up correctly and is often flawed Shrivastava *et al.* (2014); Cho *et al.* (2013).

An alternate method to estimate failure rate is to first estimate vulnerability factor, which is the probability that a fault in a hardware bit will result in program failure Mukherjee *et al.* (2005, 2003). For a given program, *vulnerability* is the sum of vulnerable bits in a processor over its execution period. Compared to fault injection, vulnerability analysis can be performed in a single simulation. This makes it useful for fast and early design space exploration Biswas *et al.* (2008); Jeyapaul and Shrivastava (2011). Vulnerability analysis must take into account the effects of all masking effects

to accurately estimate the failure rate or reliability.

Several works have been presented to estimate the system vulnerability based on cycle-accurate simulators Li *et al.* (2005); Fu *et al.* (2006). However, prior vulnerability estimations tools are limited by the underlying simulator platforms, due to which they are: i) not comprehensive, i.e., only model the vulnerability of only a small subset of the microarchitectural components of the processor, ii) are inaccurate iii) are inflexible and iv) are not validated.

Gem5 is an accurate micro-architecture simulator Butko *et al.* (2012) that addresses several problems with previous simulators. It models the processor quite close to hardware form and allows modeling various ISAs and also multicore processors. In this work, we present gemV: a tool for comprehensive and accurate vulnerability estimation based on gem5 Binkert *et al.* (2011). gemV comprehensively models the architectural vulnerability for all the sequential components of a processor. Architectural vulnerability takes into account the masking effects at micro-architectural level, but does not consider logical masking or software level masking. gemV achieves accurate architectural vulnerability estimation through i) fine-grained modeling of hardware components in a processor, ii) correctly modeling the vulnerability of squashed instructions, and iii) correctly modeling the vulnerability of inaccurately modeled hardware components. gemV is thoroughly validated by extensive fault injection experiments against benchmarks with minimal software masking. Extensive fault injection experiments validate gemV to 97% accuracy with 95% confidence.

These qualities allow gemV to be used as an early design space exploration tool for architectural vulnerability analysis. It enables us to answer questions like: how does altering the issue width of the processor affect vulnerability? Is a dual-issue processor more vulnerable than a single-issue processor? The answer is not obvious, as reducing the size of the hardware reduces the number of vulnerable bits at a given

time but it could also increase the runtime. Since bit-size and runtime affect vulnerability in opposing terms, the effect of varying hardware size can only be answered through quantitative experiments. In the same vein, how does the number of cores affect vulnerability? The algorithm of the program, the compiler used or the level of optimization used also affect the runtime. These questions of the trade-offs between runtime, hardware and software configuration, and vulnerability can now be answered rapidly and accurately. A hardware designer can use gemV to find alternate processor designs to minimize vulnerability. In my experiments, I observe that vulnerability decreases when increasing issue-width from 1 to 3. Beyond this, any increase in issue-width does not have a noticeable effect on vulnerability as any decrease in runtime is offset by the increased hardware size. A software designer can also use gemV to find the least vulnerable algorithm for a program. For example, I show that switching from a selection sort to a quick sort algorithm can affect the system vulnerability by 91%.

Chapter 2

BACKGROUND AND PREVIOUS WORK

2.1 Background

Mukherjee *et al.* (2003) propose the concept of vulnerability and present a systematic methodology to calculate vulnerability. A bit b in a microarchitectural component at a specific time t is vulnerable if a soft error in *(b,t)* results in system failure Mukherjee *et al.* (2003). Vulnerability is the sum of all vulnerable bits in a processor. The vulnerability of a processor can be estimated through micro-archtectural simulation by tracking the vulnerable bits in the processor. To accurately estimate vulnerability, the simulation must evaluate the effects of masking. Masking occurs when a soft error in a bit of the processor does not translate to system failure.

A transient error in a logical circuit might not be captured in a memory circuit because of masking Blome *et al.* (2005). This masking could be because of:

Logical masking, which occurs when the transient error is effectively gated from propagating futher due to other input values. For example, a transient error at the output of a circuit which is ANDed with 0 is logically masked.

Temporal masking, which occurs when the transient error does not arrive at a latch at the clock transition and is not latched.

Electrical masking, which occurs when the transient error is attenuated by subsequent logical gates due to the electrical properties of the gates.

Once a transient error is captured in a memory circuit, it is still possible for the error to be masked by

Architectural masking, which occurs when the soft error is masked by the architec-

tural state of the processor. For example, a soft error in a misspeculated instruction in an out-of-order processor does not result in system failure and thus is said to be masked.

Software masking, which occurs when the soft error is masked by the instructions being executed on the processor. For example, a soft error in a dynamically dead instruction does not result in system failure and is said to be masked.

A vulnerability model that can capture all masking effects will give an accurate estimation of reliability or failure rate. In this work, we only model architectural level masking effects.

2.2 Limitations of Previous Works

Several works have used cycle-accurate simulators to estimate vulnerability. SoftArch Li *et al.* (2005) modeled the error generation and propagation based on a probabilistic model in Turandot simulator Moudgill *et al.* (1999), a trace drive simulator. SoftArch Li *et al.* (2005) requires circuit-level details, such as latch and gate count, to estimate vulnerability. This is not always possible during design space exploration. Mukherjee *et al.* (2003) propose the concept of vulnerability and present a systematic methodology for calculating it. However, this tool is not publicly available. The closest work to this is Sim-SODA Fu *et al.* (2006), a microarchitectural simulator based vulnerability estimation tool which uses the Sim-Alpha simulator. Sim-SODA presents a unified simulator framework to estimate the vulnerability of various hardware structures within a processor using vulnerability computing methods introduced in Mukherjee *et al.* (2003). They estimate vulnerability by tracking the vulnerable bits in the processor pipeline for committed instructions and discarding squashed instructions. However, Sim-SODA has several limitations primarily due to the limitations of Sim-Alpha.

(i) Sim-SODA has limited usability due to Sim-Alpha. The Sim-Alpha simulator Desikan *et al.* (2001b) is a purely user-level functional simulator. The simulator has been shown to be up to 43% inaccurate in runtime estimations Desikan *et al.* (2001a). They show that in many cases, Sim-Alpha underestimates the runtime of macro benchmarks with a maximum negative error of -38.4%, while its performance is inaccurate by up to 43% for other benchmarks. Since vulnerability is directly proportional to runtime of the program Mukherjee *et al.* (2003), the inaccuracy is also reflected in the estimated vulnerability. Furthermore, Sim-SODA is limited to a single ISA (ALPHA) model. It can only simulate single core architectures and has limited microarchitectural detail. Several pipeline buffers are not modeled in Sim-Alpha. It does not model a floating point pipeline and thus is limited to integer benchmarks.

(ii) Sim-SODA is inaccurate. They estimate vulnerability at a coarse level of granularity, leading to inaccurate estimation of vulnerability. For instance, several hardware structures in the instruction fetch and issue logic are modeled as a single hardware structure – "the instruction window", which does not model individual hardware structures such as the fetch queue, decode queue, and therefore cannot be evaluated for their vulnerability.

(iii) Sim-SODA is not comprehensive in its vulnerability modeling. Several hardware structures such as the pipeline registers, rename map, and history buffer are not modeled. Comprehensiveness is an important quality for a vulnerability estimation tool to study the breakdown of vulnerability of a specific hardware structure as a percentage of the total processor vulnerability. This is useful in studying the effectiveness of new protection mechanisms and also in designing new protection mechanisms to target the hardware structure contributing the highest percentage of the overall system vulnerability. Furthermore, Sim-SODA does not model realistic

hardware with protection mechanisms. Caches on many modern processors are built with parity protection techniques or ECC. Modeling these protection mechanisms within a vulnerability estimation tool allows for realistic estimation of vulnerability.

(iv) Sim-SODA tool is not validated. As described in Section 3.1.3 , gemV has been validated against fault injection experiments and its accuracy established.

Chapter 3

GEMV: FINE-GRAINED COMPREHENSIVE VULNERABILITY ESTIMATION

3.1 Modeling

This section describes my approach to gemV as a vulnerability estimation tool by addressing the problems observed in previous works.

3.1.1 GemV is Accurate

I achieve accurate vulnerability estimation with a four-pronged approach:

Leveraging the gem5 simulator framework: gemV is built on gem5, which is a pretty accurate cycle-accurate simulator, with 1-17% error in runtime estimation Butko *et al.* (2012). Furthermore, unlike previous simulation infrastructures (e.g., Simplescalar, Sim-Alpha), gem5 models the microarchitectural components of processor in a close to hardware form. This allows us to track the read and write to the bits inside the component accurately and therefore model vulnerability accurately. Further, gem5 has the ability to run in full system mode that can simulate the behaviour of an operating system. Any operating system activity during the execution of the program should also be analyzed for vulnerability. By adding the vulnerability due to operating system calls, the accuracy of total system vulnerability is improved.

Fine-grained vulnerability tracking: To understand why fine-grained modeling is important, it is first important to understand what fine-grained modeling is. This section describes fine-grained modeling and its importance using two examples - pipeline

registers and cache.

A naive or coarse grained vulnerability analysis model views a hardware structure as monolothic for the purposes of vulnerability analysis. This means that when an instruction is read in that hardware structure, all the bits in the hardware structure are considered vulnerable. In a fine-grained vulnerability model, the hardware structure is broken down into its indivudual fields. Some of these fields may not be used by some instructions. For example, pipeline registers in the *Rename/IEW* stage have fields to hold memory addresses for memory reference instructions. However, these fields are not used by other instructions. Thus treating them as vulnerable would lead to inaccurate vulnerabilty estimates. In an ARM-v7a pipeline, ALU instructions use 71 bits of the *Rename/IEW* pipeline stage, whereas memory-reference instructions use 132 bits Jeyapaul (2015). Fine-grained modeling is thus important because not all bits of a hardware structure are vulnerable for every instruction. During execution of an instruction, only the bits in a micro-architectural component that store data specific to that instruction are vulnerable. By identifying the subset of fields in a pipeline registers that are vulnerable for different instruction types, we can compare the difference between a fine-grained and a coarse-grained vulnerability model. The subset of fields in the pipeline register used by different instruction types can be identified using RTL analysis on the processor pipeline. As shown in Fig. 3.1, on an average 21% of the fields in the pipeline registers are actually vulnerable. Coarse-grained vulnerability models assume that all the fields are vulnerable, thus overestimating the vulnerability by 5X.

In the case of caches, coarse-grained vulnerability models treat entire cache blocks as vulnerable. However, reads and writes to the cache often occur at the word-level. Treating the entire cache block as vulnerable in this case would again lead to an overestimation of vulnerability. Fine-grained vulnerability analysis is thus also important

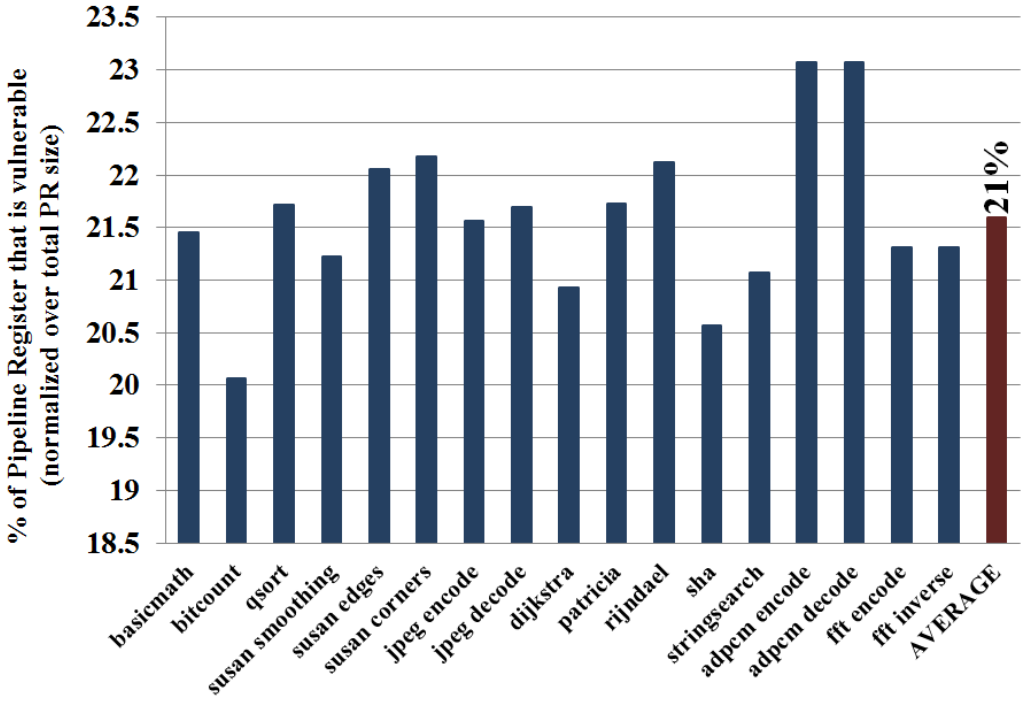

Figure 3.1: Percentage of Pipeline Register Fields That Are Actually Vulnerable (Normalized Over the Case When All the Bits Are Considered Vulnerable). A Naive Method of Vulnerability Analysis Assumes That All The Pipeline Register Bits Are Vulnerable. However, Fine-grained Analysis Shows That Actual Vulnerability Of the Pipeline Is Only 21% Of the Naive Method.

for caches for accurate vulnerability estimation. Fig. 3.2 shows the comparison between a coarse-grained vs a fine-grained cache vulnerability mode. The coarse-grained model treats entire cache blocks as vulnerable whereas the fine-grained model tracks vulnerability for individual words in the cache. We observe that the coarse-grained model consistently overestimates the vulnerability of the cache for various benchmarks and on an average it overestimates it by about 12%.

To achieve fine-grained vulnerability estimation in gemV, I instrument every hardware component modeled in the gem5 out-of-order processor with a *Vulnerability Tracker* – which tracks the read/write accesses on each component and thereby computes their respective vulnerable periods at the *field* level granularity. In this, with the knowledge of the type of instruction accessing the hardware, instruction specific

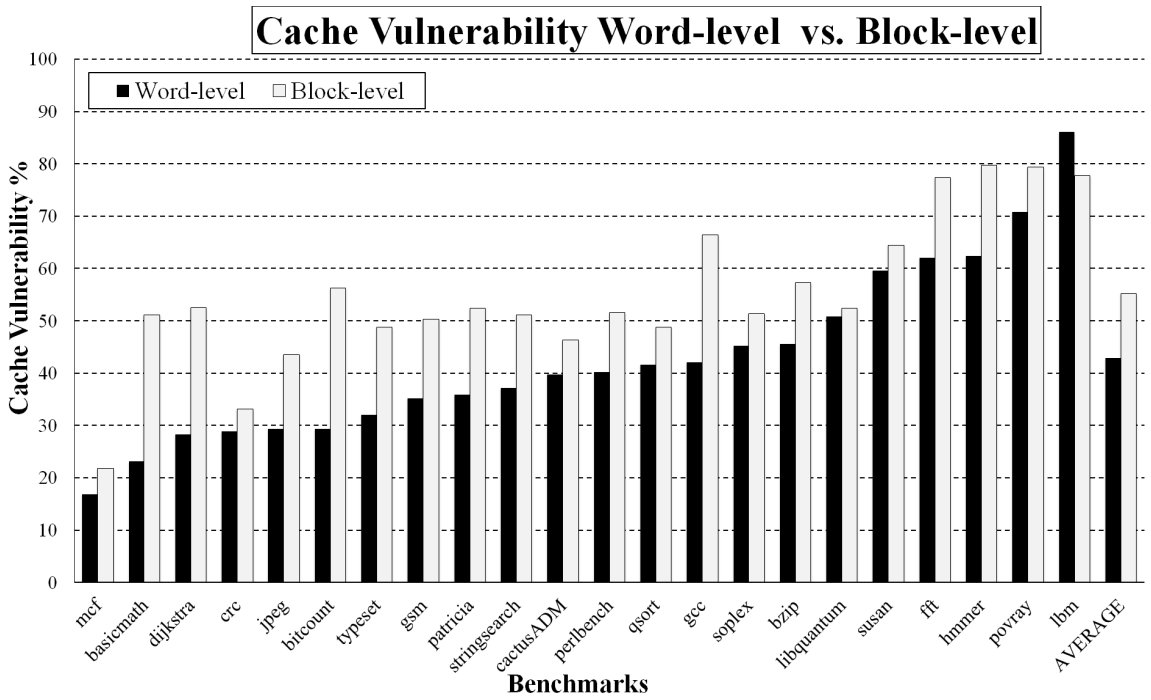

Figure 3.2: Cache Vulnerability Comparison Between a Coarse-Grained and Fine-Grained Model for Various Benchmarks. Coarse-grained Models Vulnerability at Block-level While Fine-grained Models Cache at Word-level. The Coarse-grained Model Overestimates Cache Vulnerability By an Average 12%

.

vulnerability modeling can be applied. For instance, if an ALU instruction is passing through the *Rename/IEW* pipeline stage, the vulnerability tracker only tracks the vulnerability of the bits that are vulnerable for an ALU instruction. For the cache, accesses to a word (in a cache-block) is monitored individually, and based on the configured working of the cache architecture (movement of blocks between cache levels and memory), the vulnerability periods are computed accurately.

Accurate modeling of the vulnerability of shared data structures in gem5: As shown in Fig. 3.3, gem5 uses the *dynamic instruction* structure to handle the intricacies of microarchitectural simulation in software. This structure is shared across multiple hardware components such as the pipeline registers. This is done for ease

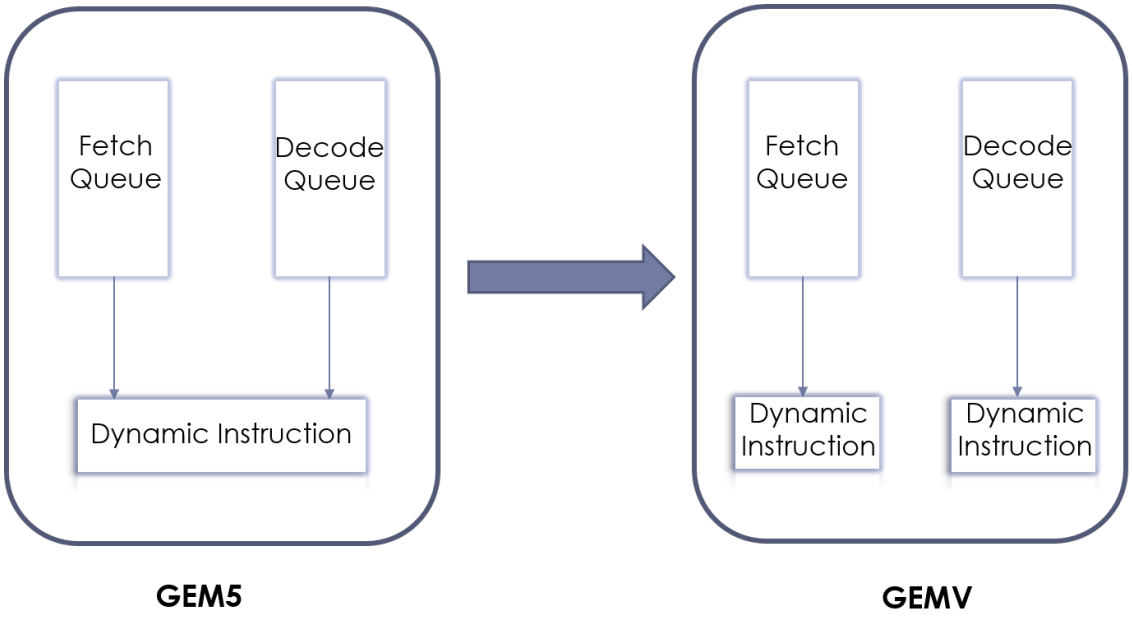

Figure 3.3: Shared Dynamic Instruction in Gem5 is Split Among Individual Hardware Structures in GemV for Realistic Hardware Modeling.

of programming and fast simulation. However, this is not how the hardware would work. Accurate vulnerability estimation requires that all hardware structures be independently analyzed. To achieve this, I modified the gem5 framework to separate all the hardware components using the dynamic instruction and track their states independently. This allows us to estimate the vulnerability and inject faults on each hardware component independently.

Accurate modeling of the vulnerability of squashed instructions: Correctly accounting for vulnerability when an instruction is squashed improves accuracy. An instruction is squashed due to mis-speculation in an out-of-order processor. Under these conditions, most of the bits used by the instruction are considered not vulnerable. However, certain bits are still vulnerable. As shown in Fig. 3.4, the rename map is used to maintain a mapping between architectural and physical registers. The rename map uses a history buffer to maintain the previous mapping of an architec-

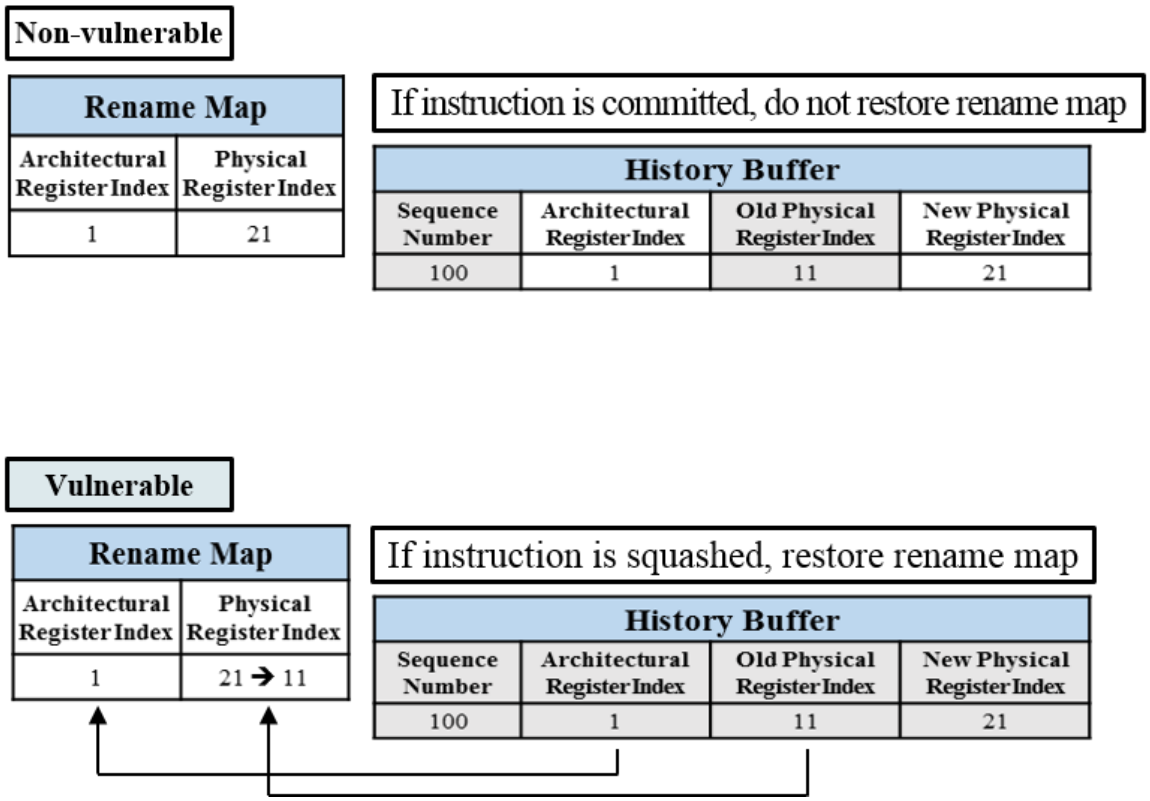

Figure 3.4: A Scenario Under Which the History Buffer is Vulnerable.

tural register. This is so that when an instruction is squashed, the processor state can be rolled back to the last committed instruction. Therefore, when an instruction is squashed, the history buffer is vulnerable as it is read and the old mapping written back to the rename map. Previous vulnerability estimation tools such as Sim-SODA Fu *et al.* (2006) considered all squashed instructions to be not vulnerable.

However, one of the limitations of gemV is that it does not consider the effects of software masking on vulnerability. Masking due to dynamically dead instructions could impact the system vulnerability, but is not modeled in gemV.

3.1.2 GemV is Comprehensive

gemV is comprehensive as it models the vulnerability of all major hardware structures in a processor - such as the fetch queue, decode queue, rename queue, issue

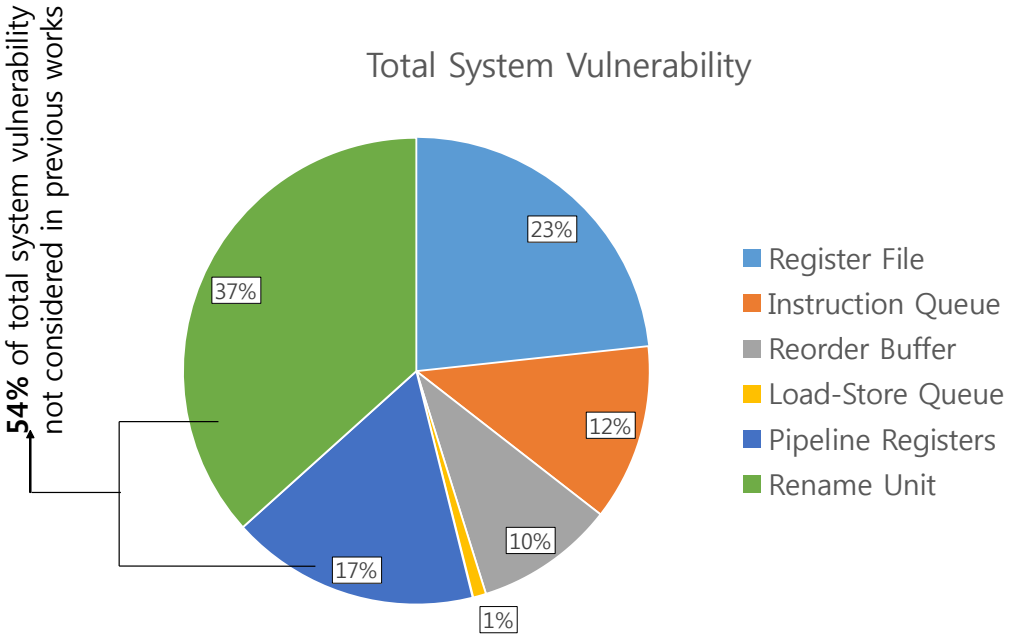

Figure 3.5: Breakup of total processor vulnerability of which 53% has not been modeled in previous work.

queue. I also model the complete rename map by tracking vulnerability of the rename map, history buffer and pipeline queue registers. Fig. 3.5 shows the breakup of processor vulnerability in the default configuration of gem5 ARM out-of-order processor running *stringsearch* benchmark. About 54% of the total system vulnerability that gemV models has not been modeled in previous works.

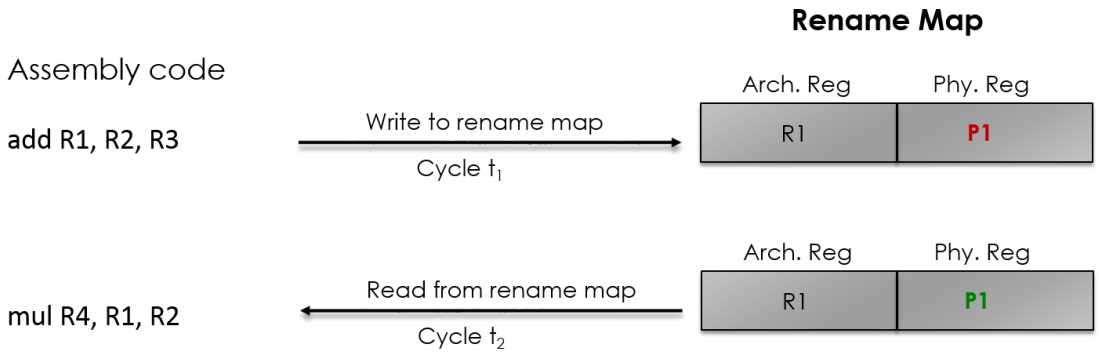

Figure 3.6: Vulnerability Tracking in GemV. The Tracker Tracks Read and Write Accesses to Calculate Total Vulnerability.

I classify the hardware structures in gem5 into two types for vulnerability mod-

eling. (i) Pipeline structures such as the fetch queue, decode queue, rename queue. The vulnerability of these structures is modeled by tracking an instruction moving through the pipeline. I track the cycles at which an instruction is written to and read from a pipeline structures. (ii) Storage structures such as the register file, rename table and history buffer. The vulnerability of these structures is modeled by inter-instruction dependency. Reads and writes from all the instructions to these storage structures is tracked to find vulnerable write to read intervals. As shown in Fig. 3.6, if an instruction writes to the rename map at cycle t_1 and another instruction reads from the rename map at cycle t_2. Then $t_2 - t_1$ is the vulnerable period of that entry of the rename map. A vulnerability tracker records all the reads and writes to each field in a structure along with the corresponding cycle number. When an instruction is committed, the tracker compiles the sequence of reads and writes into vulnerable t_v and non-vulnerable periods t_{nv}. The vulnerability of a hardware structure can be calculated as $\sum t_v * S$ where S is the size of the hardware structure.

3.1.3 GemV is Validated

In order to establish the accuracy of gemV, I perform extensive fault injection on every microarchitectural component modeled. For each fault injection run, a single-bit in a component is flipped at a random time (during program execution). In this validation campaign, I inject 300 faults per component for each of the ten benchmarks from MiBench Guthaus *et al.* (2001) and SPEC CPU2006 Henning (2006). 300 fault injections give us 95% confidence in the test results Leveugle *et al.* (2009).

Table 3.1 lists the results of the fault injection experiments. The results show that component vulnerability estimated using gemV is about 97% accurate. Benchmarks were chosen to minimize the effects of software masking on error propagation.

I implement the fault injection setup in gem5. For components that are modeled as

Table 3.1: GemV Validation Against Fault Injection. 300 Faults Injected Per Component for Each of the Following Benchmarks: *Matrix Multiplication*, *Hello World*, *Stringsearch*, *Perlbench*, *Gsm*, *Qsort*, *Jpeg*, *Bitcount*, *Fft*, and *Basicmath*

Component	**Faults Injected**	**Match**	**Mismatch**	**Accuracy**
Register file	3000	2899	101	96.63
Rename map	3000	2748	252	91.60
History buffer	3000	2781	219	92.70
Instruction queue	3000	2978	22	99.27
Reorder buffer	3000	2760	240	92.00
Load-store queue	3000	2979	21	99.30
Fetch queue	3000	2890	110	96.33
Decode queue	3000	2902	98	96.73
Rename queue	3000	2827	173	94.23
I2E queue	3000	2959	41	98.63
IEW queue	3000	2873	127	95.77
Overall Accuracy				96.78%

independent structures (lists, queues, etc.), I implement a wrapper to pick a random bit and flip it during a random cycle of execution. gem5 is designed such that there is sharing of dynamic instruction information across some of the components like ROB, LSQ and IQ. In such components, it is not possible to directly utilize the software design to implement bit level fault injections. To circumvent this problem, I instrument the simulator infrastructure in such a way that for each component the bit-fields and structures can be manipulated independently.

The result of a validation run is declared as a match if the result of the fault injection agrees with the prediction made by gemV. For example, if gemV predicts that a bit is vulnerable, then the corresponding fault injection run should result in an incorrect output or program failure. As shown in Table 3.1, there are 2899 matched results and 101 mismatched results for the Register File, giving us an accuracy of

96.63%.

3.1.4 GemV is Flexible

gemV is flexible in its support for multiple ISAs, multi-cores and system call simulation Binkert *et al.* (2011). Due to this, gemV offers several advantages in vulnerability estimation over previous works. Firstly, gemV can estimate vulnerability irrespective of the underlying ISA. This can be used in estimating vulnerability of the same program across different ISAs such as x86, ARM, SPARC, etc as demonstrated in Fig. 4.3. Secondly, gemV can estimate the vulnerability of a program running on out-or-order processors in both single core and multi-core configurations.

3.1.5 GemV Models COTS Processors

gemV is capable of estimating vulnerability for commodity off-the-shelf (COTS) processors. This is achieved by taking advantage of the gem5 platform as an accurate and complete simulator framework and further build on it by modeling protection techniques such as parity and ECC protected caches. Several modern and popular embedded processors such as the ARM1156T2S, ARM Cortex A8 and AM3359 Ko *et al.* (2015b) use parity protection for reads and writes in their caches. The vulnerability of programs running on such processors can be studied using gemV.

3.1.6 Limitations of GemV

While gemV offers several advantages over existing vulnerability estimation tools, it also has a few limitations. (i) gemV does not model several masking effects such as logical masking and dynamic dead code masking. (ii) gemV is limited by the accuracy of the processor model in gem5.

3.2 GemV: Implementation in Gem5

gemV is implemented in gem5 with a *Vulnerability Tracker*. The vulnerability tracker is a modular plugin to the gem5 code base that allows fast prototyping and rapid development of vulnerability tracking for new components. An an instruction passes through the pipeline, the vulnerability tracker tracks the reads and writes to each hardware structure simulated in gem5. When the instruction is committed, the vulnerability tracker computes its associated vulnerability.

The vulnerability tracker wraps around all reads, writes, commits and squashes that occur in the gem5 simulation framework. For example, when an instruction enters the rename stage, it is written in the rename table. The vulnerability tracker captures the tick of this write as shown in 3.1.

Listing 3.1: Capturing a read in the register file with the vulnerability tracker

```
FullO3CPU<Impl>::readArchIntReg(int reg_idx, ThreadID tid)
{
    intRegfileReads++;
    PhysRegIndex phys_reg = commitRenameMap[tid].lookupInt(reg_idx);
    //Vulnerability tracking. Capture the register number and the CPU cycle
    this->regVulTrack.vulOnRead(phys_reg, tick);
    return regFile.readIntReg(phys_reg);
}
```

Similarly, reads are also tracked in all the micro-architectural components. When the instruction is retired after a commit or squash, the vulnerable intervals are calculated from the reads and writes as shown in 3.2.

Listing 3.2: Calculating vulnerable intervals when an instruction is retired

```
RegisterVulnerabilityCalculator::instRetire()
{
    for(int idx = 0; idx < numRegs; ++idx) {
      // Iterate over the list of reads and writes to find vulnerable
          intervals
        if(!hist[idx].empty()) {
            std::list<History>::iterator hiter = hist[idx].begin();
            Cycle previous_cycle = hiter->cycle;
            Operation previous_op = hit->op;
            while(hit != hist[idx].end()) {
                // If READ after WRITE, then interval is vulnerable
                if(hit->op == READ && (previous_op == READ || previous_op ==
                    WRITE)) {
                    vul += REGISTER_WIDTH*(hiter->this_cycle -
                        previous_cycle)/TICKS_PER_CYCLE;
                }
                prev_cycle = hiter->cycle;
                prev_op = hiter->op;
                hiter++;
            }
        }
    }
}
```

Chapter 4

GEMV FOR DESIGN SPACE EXPLORATION

The value of gemV is in making possible fast and early DSE or Design Space Exploration. Radiation testing requires developers to build a fully working prototype before evaluating the reliability, and even RTL fault injection requires developers to bring down the design to synthesizable form before reliability can be quantified. As opposed to these, gemV allows designers to evaluate reliability at a very early high-level modeling stage. As opposed to fault injections in micro-architecture simulator, gemV is hundreds of times faster, since it can estimate reliability in just one simulation run.

4.1 GemV for Hardware Design

gemV can quantitatively answer difficult performance-vulnerability trade-off questions, e.g., how does changing the issue-width in a processor affect runtime and vulnerability? On one hand, a wider issue-width could reduce the runtime and therefore vulnerability. But on the other, a wider issue-width requires more sequential components in the processor, thus increasing the vulnerability. The overall effect on vulnerability is not obvious. With gemV, we can study the effects of such changes and quantitatively answer such difficult questions. For our benchmarks, we observe that vulnerability decreases when increasing issue-width from 1 to 3. Beyond this, any increase in issue-width does not have a noticeable effect on vulnerability as any decrease in runtime is offset by the increased hardware size.

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

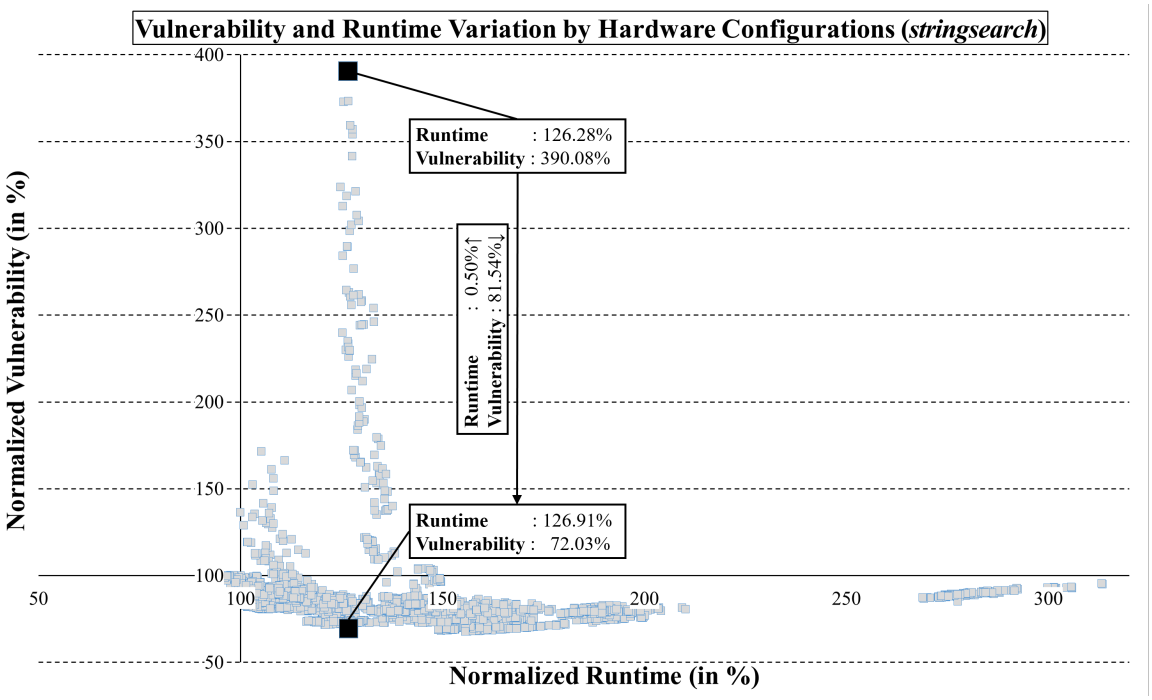

Figure 4.1: Different Hardware Configurations Generates Interesting Design Space in Terms of Runtime and Vulnerability. Vulnerability Can be Reduced by up to 82% With Less Than 1% Runtime Overhead by Varying Hardware Configurations.

some design parameters, e.g., cache sizes, issue width, ROB size, load store queue size etc., to minimize the vulnerability. This can be answered with gemV by plotting design points for runtime against vulnerability. In this experiment, we vary the total number of entries in the Re-order Buffer(ROB), Load-Store Queue(LSQ) and Instruction Queue(IQ) to plot a design space for *stringsearch* Guthaus *et al.* (2001) as shown in Fig. 4.1. We establish a baseline runtime and vulnerability with sizes of 192, 64, and 8 entries for ROB, LSQ, and IQ respectively. A hardware designer can use this design space to choose the required hardware configuration as dictated by runtime and vulnerability bounds. Given a certain runtime target, the hardware designer can now find several design points for vulnerability as shown by the grey band in Fig. 4.1. In this example, for a runtime overhead of $\pm 2\%$, it is possible to find a design point with 81% less vulnerability. Given any runtime or vulnerability

overhead it is now possible to find alternate design points with lower vulnerability or runtime with gemV. Note that the benchmarks were chosen to minimize the effects of software level masking effects.

Table 4.1: Runtime Overhead(%) For an Optimal Component Size to Minimize Vulnerability

	ROB	**IQ**	**LSQ**	**FW**	**DW**	**RW**	**I2EW**
Runtime Overhead %	1.18	49.55	85.36	41.74	41.43	43.89	0.65

ROB: Re-Order Buffer, IQ: Instruction Queue, LSQ: Load-Store Queue, FW: Fetch Width, DW: Decode Width, RW: Rename Width, I2EW: Issue Width

Table 4.2: Vulnerability Overhead(%) For an Optimal Component Size to Minimize Runtime

	ROB	**IQ**	**LSQ**	**FW**	**DW**	**RW**	**I2EW**
Vulnerability Overhead %	5.83	42.85	27.89	20.12	20.28	20.64	2.46

ROB: Re-Order Buffer, IQ: Instruction Queue, LSQ: Load-Store Queue, FW: Fetch Width, DW: Decode Width, RW: Rename Width, I2EW: Issue Width

I also find interesting trade-offs between runtime and vulnerability by varying the size of a component with all other components fixed. For example, Table 4.1 shows that choosing an LSQ size for minimum vulnerability will increase runtime up to 85% for *stringsearch*. Similarly, Table 4.2 shows that choosing an Instruction Queue size for minimum runtime can increase vulnerability up to 43%. I can also observe that the trend in vulnerability and runtime variation per component. For example, runtime is more sensitive to LSQ size than vulnerability since runtime can be reduced by 64% while vulnerability increases by just 28%. On the other hand, both runtime and vulnerability decrease as the issue width increases. Interestingly, the issue width affects vulnerability more adversely than the runtime as opposed to the LSQ. Runtime and vulnerability can be reduced by up to 21% and 59%, respectively, with increasing issue width.

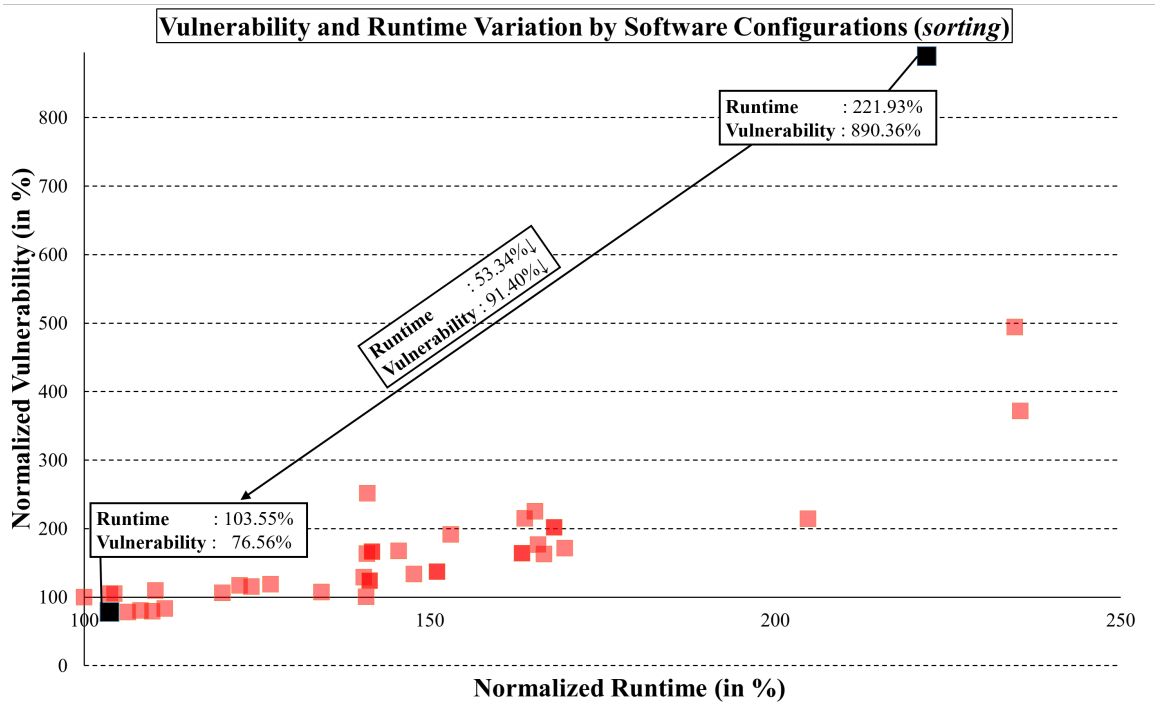

Figure 4.2: Different Software Configurations can Generate Interesting Design Space in Terms of Vulnerability on the Same Hardware. Vulnerability can be Reduced by 91% Without Runtime Overhead With Software Changes.

gemV can also be used by the software engineer to find alternate design points with lower vulnerability or runtime. Alternate design points can be realized with software changes in either the algorithm, the compiler used or the level of optimization. For example, given the choice of two sorting algorithms - such as quick sort and insertion sort - which would be the optimal choice for the best trade-off between runtime and vulnerability? gemV can be used to study the design space for runtime and vulnerability due to changes in software. To study such changes, we perform an experiment by establishing a baseline runtime and vulnerabiity for an insertion sort algorithm compiled with *gcc* at the highest(O3) level of optimization. Fig. 4.2 presents the normalized runtime and vulnerability for various combinations of algorithms, compilers and optimization levels. We consider an array sorting application with

five sorting algorithms (bubble, quick, insertion, selection, and heap sorting), two compilers (GCC and LLVM Lattner and Adve (2004)), and four optimization levels (no optimization, O1, O2, and O3). We note that vulnerability can be reduced by up to 91% without additional runtime overhead with software changes. The software engineer can use this design space to choose optimal design points to meet runtime and vulnerability requirements. In this example, switching from a selection sort algorithm at O1 level of optimization to quick sort at O3 level of optimization reduces runtime by 53.34% and vulnerability by 91.4%.

Table 4.3: Effects of Software Configuration(Algorithm, Optimization Level And Compiler) On Run-time and Vulnerability (*Sorting*)

		Mean (%)	Max (%)	Min (%)
Algorithm	Runtime	32.73	113.95	11.23
	Vulnerability	140.77	1005.44	23.44
Optimization	Runtime	33.03	101.19	9.69
	Vulnerability	120.39	739.46	6.06
Compiler	Runtime	26.39	52.33	0.35
	Vulnerability	48.46	314.08	5.16

As summarized in Table 4.3, software changes can result in a large design space for runtime and vulnerability. In general, vulnerability is much more sensitive to the software configurations than runtime. The maximum increase in runtime can be up to 114% by changing the sorting algorithm. The choice of compiler can also affect vulnerability up to a 314%. This vulnerability-aware design space exploration in software can allow the software designer to meet specific requirements in runtime or vulnerability or both.

4.3 GemV for System Design

A system designer can also use gemV to make design choices in several interesting ways. In this experiment, we will demonstrate two such examples. (i) Given a choice

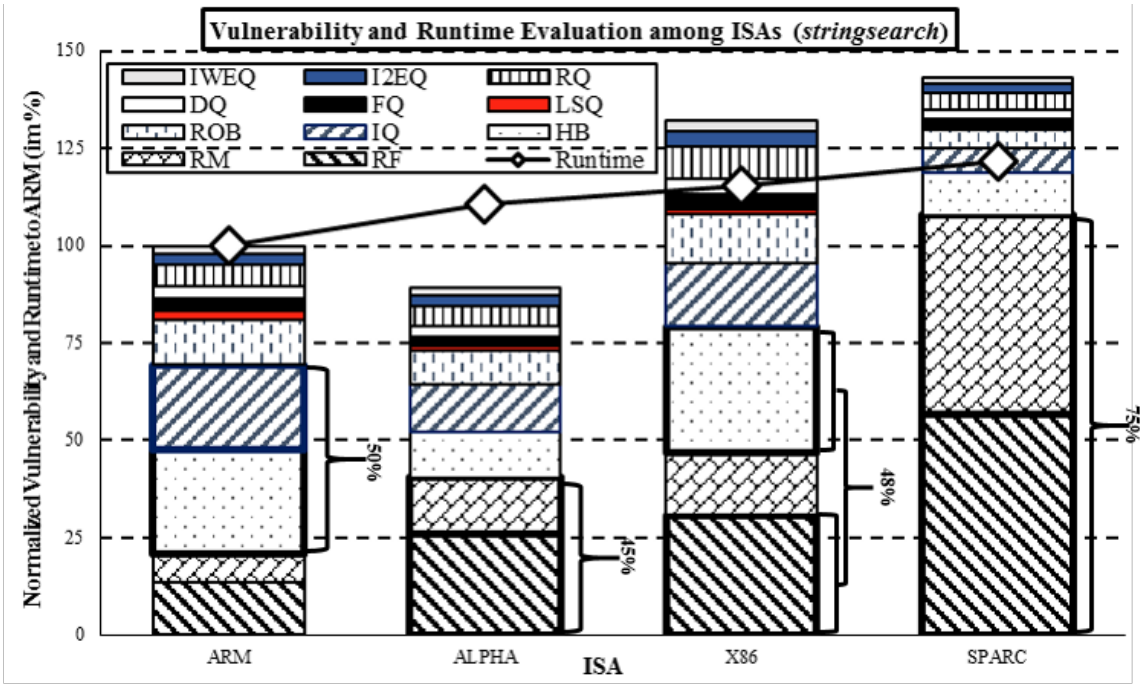

Figure 4.3: Variation in Runtime And Vulnerability For *Stringsearch* Under Different ISAs

of processors running different ISAs, which one offers the best trade-off in runtime or vulnerability? We ran this experiment by changing the ISA within gemV while keeping all hardware sizes constant. Fig. 4.3 shows vulnerability and runtime under different ISAs such as ARM, SPARC, x86, and ALPHA for the *stringsearch* benchmark, with no change in hardware and software configurations. Baseline vulnerability and runtime are established on the ARM ISA. *Stringsearch* running on an ALPHA is 38% less vulnerable than an equivalent SPARC. The system designer can choose the ARM ISA for minimum runtime or the ALPHA for minimum vulnerability.

(ii) The system designer can also study the breakdown of vulnerability to individual hardware components. This can be used to design protection techniques targeting specific components.Fig. 4.3 shows the detailed breakdown of each component such as HB (history buffer), RM (rename map), LSQ, IQ, IEWQ (IEW queue), I2EQ, RQ, DQ, FQ, RF (register file), and ROB. History buffer and IQ take up the highest

fraction (50%) of the vulnerability in an ARM processor while the Rename Map and Register File contribute the most in case of SPARC and ALPHA respectively. In this example, a protection mechanism such as ECC can be applied to the register file on the SPARC processor. However, the same protection is not very useful on the ARM processor as the RF contributes only 21% to the system vulnerability.

Chapter 5

SUMMARY

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

REFERENCES

Alkhalifa, Z., V. S. Nair, N. Krishnamurthy and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection", Parallel and Distributed Systems, IEEE Transactions on **10**, 6, 627–641 (1999).

Baumann, R., "Soft errors in advanced computer systems", Design Test of Computers, IEEE **22**, 3 (2005).

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator", ACM SIGARCH Computer Architecture News **39**, 2 (2011).

Biswas, A., P. Racunas, J. Emer and S. S. Mukherjee, "Computing accurate AVFs using ACE analysis on performance models: A rebuttal", Computer Architecture Letters **7**, 1, 21–24 (2008).

Blome, J., S. Mahlke, D. Bradley and K. Flautner, "A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor", in "In Proceedings of the First Workshop on Architecture Reliability", (2005).

Butko, A., R. Garibotti, L. Ost and G. Sassatelli, "Accuracy evaluation of gem5 simulator system", in "Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on", pp. 1–7 (IEEE, 2012).

Cho, H., S. Mirkhani, C.-Y. Cher, J. A. Abraham and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design", in "Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE", pp. 1–10 (IEEE, 2013).

Desikan, R., D. Burger and S. W. Keckler, "Measuring experimental error in microprocessor simulation", in "ISCA", (ACM, 2001a).

Desikan, R., D. Burger, S. W. Keckler and T. Austin, "Sim-alpha: a validated, execution-driven Alpha 21264 simulator", UT Austin, Tech. Rep. (2001b).

Dixit, A. and A. Wood, "The impact of new technology on soft error rates", in "IRPS", (IEEE, 2011).

Entrena, L., M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela and C. Lopez-Ongil, "Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection", Computers, IEEE Transactions on **61**, 3, 313–322 (2012).

Fu, X., T. Li and J. Fortes, "Sim-SODA: A unified framework for architectural level software reliability analysis", in "MoBS", (2006).

Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in "WWC", (2001).

Henning, J. L., "SPEC CPU2006 benchmark descriptions", ACM SIGARCH Computer Architecture News **34**, 4 (2006).

Jeyapaul, R., "Systematic methodology for the quantitative analysis of pipeline register reliability in embedded systems", Tech. rep., Arizona State University (2015).

Jeyapaul, R. and A. Shrivastava, "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors", in "CASES", (2011).

Ko, Y., R. Jeyapaul, Y. Kim, K. Lee and A. Shrivastava, "Accurate cache vulnerability modeling in presence of protection techniques", in "Resiliency in Embedded Elenctronic Systems (REES)", (2015a).

Ko, Y., R. Jeyapaul, Y. Kim, K. Lee and A. Shrivastava, "Guidelines to design parity protected write-back l1 data cache", in "Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE", pp. 1–6 (IEEE, 2015b).

Lattner, C. and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", in "CGO", pp. 75–86 (IEEE, 2004).

Lee, I., M. Basoglu, M. Sullivan, D. H. Yoon, L. Kaplan and M. Erez, "Survey of error and fault detection mechanisms", UT Austin, Tech. Rep (2011).

Leveugle, R., A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence", in "DATE", (2009).

Li, X., S. Adve, P. Bose and J. Rivers, "SoftArch: An architecture-level tool for modeling and analyzing soft errors", in "DSN", (2005).

Mahatme, N., N. Gaspard, S. Jagannathan, T. Loveless, B. Bhuva, W. Robinson, L. Massengill, S.-J. Wen and R. Wong, "Impact of supply voltage and frequency on the soft error rate of logic circuits", Nuclear Science, IEEE Transactions on **60**, 6, 4200–4206 (2013).

Martinez-Alvarez, A., S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems", Dependable and Secure Computing, IEEE Transactions on **9**, 2, 159–172 (2012).

Michel, T., R. Leveugle and G. Saucier, "A new approach to control flow checking without program modification", in "Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium", pp. 334–341 (IEEE, 1991).

Moudgill, M., P. Bose and J. H. Moreno, "Validation of Turandot, a fast processor model for microarchitecture exploration", in "IPCCC", (IEEE, 1999).

Mukherjee, S. S., J. Emer and S. K. Reinhardt, "The soft error problem: An architectural perspective", in "HPCA", (IEEE, 2005).

Mukherjee, S. S., C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", in "Micro", (IEEE/ACM, 2003).

Naseer, R., Y. Boulghassoul, J. Draper, S. DasGupta and A. Witulski, "Critical charge characterization for soft error rate modeling in 90nm SRAM", in "ISCAS", (2007).

Nguyen, H. T. and Y. Yagil, "A systematic approach to SER estimation and solutions", in "IRPS", (IEEE, 2003).

Nicolaidis, M., *Circuit-Level Soft-Error Mitigation* (Springer, 2011).

Shrivastava, A., A. Rhisheekesan, R. Jeyapaul and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors", in "DAC", (2014).