Comprehensive Resiliency Evaluation for Dependable Embedded Systems

Yohan Ko

The Graduate School
Yonsei University
Department of Computer Science

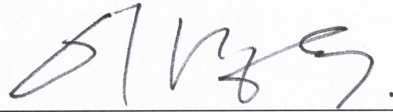
Comprehensive Resiliency Evaluation for Dependable Embedded Systems

A Dissertation
Submitted to the Department of Computer Science
and the Graduate School of Yonsei University
in partial fulfillment of the
requirements for the degree of
Ph.D. in Computer Science

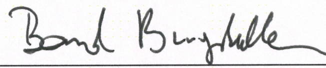
Yohan Ko

February 2018

This certifies that the dissertation
of Yohan Ko is approved.

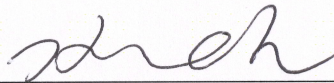


Thesis Supervisor: Kyoungwoo Lee

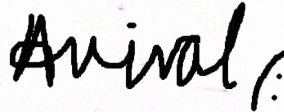


Thesis Committee Member: Bernd Burgstaller

Thesis Committee Member: Yo-Sub Han



Thesis Committee Member: Hyunok Oh



Thesis Committee Member: Aviral Shrivastava

The Graduate School
Yonsei University
February 2018

감사의 글

2017년은 유난히도 추웠다. 날씨가 추운 것은 그래도 견딜 수 있었지만 마음이 춥고 힘든 것은 꽤나 견디기가 어려웠다. 나이 서른이라는 숫자가 주는 의미가 너무나도 컸다. 어렸을 적 이런 생각을 했다. 나이가 서른 정도 되면, 집도 있고, 차도 있겠지. 조금씩 머리가 커지면서 집이나 차 둘 중 하나는 있겠지라는 생각으로 바뀌었다. 그리고 서른이 된 지금 아무것도 가진 것이 없다. 그리고 그런 가진 것 없는 내가 아직도 학생이라는 사실이 너무나도 추웠던 것 같다.

그럼에도 드디어 6년의 학업을 마무리할 무언가가 나왔다. 학위논문 한 편을 쓰기 위해서 다른 많은 논문을 써야만 했는데 그런 논문 하나하나를 쓸 때마다 부족한 완성도로 인해서 항상 고민이 앞섰다. 그럼에도 불구하고 부족한 논문이라도 제출을 하고 발표를 했던 것은 내 서재 속에 꽂혀있는 완벽한 한 문장보다는 졸문이라도 세상 빛을 본 글에 가치를 두는 내 이상한 자존심 때문이었다. 그리고 내가 쓴 글로 어떠한 형태가 되었던 세상에 이야기를 하고 싶었다. 내가 지금 무슨 공부를 하고 있는지, 이게 무슨 가치를 가지고 있는지에 대해 말이다.

6년이란 짧지 않은 시간, 고마운 사람들이 많다. 먼저, 내가 하고 있는 놀이를 연구라는 높이로 한걸음 도약시켜준 지도교수님과 연구실 친구들에게 감사의 말을 전하고 싶다. 연구가 힘든 이유는 내가 업이라고 생각하는 일이 남에게는 아무것도 아닌 일이 아닐 수도 있다는 두려움이다. 그래서 때로는 내 연구의 가치를 나보다 더 잘 이해하는 협력자의 눈길로, 때로는 연구의 가치를 냉정하게 평가하는 동료 연구자의 시선으로 균형 잡힌 연구를 할 수 있도록 도와준 이들이 없었다면 박사가 되는 연구가 아닌 그냥 나 혼자 하는 자기만족에 지나지 않았을 것이다.

내 주위를 지켜준 사람의 공로 역시 잊을 수 없다. 박사과정 자체가 (공부하고 있는 사람조차) 쉽게 이해할 수 없는 분야를 아무나 이해하지 못하는 수준까지 파고들어야 하는 외로운 과정이기 때문에 곁에 사람이 없다면 쉽게 지치기 마련이

다. 내가 지치는 것은 그래도 괜찮은데, 이 과정이 어려운 것은 주위사람 역시 지치게 하는 과정이기 때문이다. 공부하는 것에 미쳐서 돈을 벌어야 하는 장남의 위치를 망각해도 이해해주던 부모님과, 내 대신 경제적인 대들보 역할을 수행한 동생에게 감사의 말을 다시 한 번 전한다. 또한, 적지 않은 나이임에도 나와 함께 긴 겨울 새봄을 기다린 여자친구 역시 어쩌면 보이지 않는 터널을 같이 건너온 동반자일지 모른다.

마지막으로 내 자신에게도 감사의 말을 전하고 싶다. 박사 과정에 들어오면서 나는 몇 가지 목표를 세웠다. 그리고 그 목표를 꽤나 구체적으로 세우려고 노력했다. 하나의 좋은 학회 논문, 그리고 그 좋은 논문을 확장한 완성도 있는 저널 논문. 경제적인 부담을 조금이라도 완화할 수 있는 장학금 수혜. 그리고 외국계 기업에서의 인턴 생활. 지금 생각해보면 치열하게 산 덕분에, 아니 그보다 운이 좋은 덕분에 내가 원하는 것을 그래도 모두 이룬 생활이었다. 짧지 않은 기간임에도 늘 동기부여를 하려고 바쁘게 살았고, 그 바쁜 삶을 내가 살아있다는 증거로 여기며 살아온 시간이었다.

이렇게 감사의 말을 쓰고 있는 지금도 솔직히 말하면 두려움이 앞선다. 이제 계획을 세우고, 연구를 하고, 실험을 하고, 논문을 쓰는 과정에 익숙해졌는데 이제는 다른 세계로 가야한다는 그런 두려움이 말이다. 다만 그 두려움은 내가 학부 졸업을 앞두고 대학원이라는 새로운 공간으로 가야한다는 생각에 느끼던 두려움과는 다르다. 그때의 두려움이 새로운 일이라는 보이지 않는 것에 대한 두려움이었다면, 지금은 새로운 일을 할 수 있기 때문에 느끼는 기대감에 가까울지도 모른다. 아마 내가 박사를 준비하면서 배운 것은 컴퓨터 아키텍처가 아닌 그런 방법론일 것이다. (물론 컴퓨터 시스템도 충분히 배웠습니다.)

다시 한 번, 이 글을 읽고 있는 사람들에게도 전하고 싶다. 고맙습니다.

Contents

List of Figures .... **iii**

List of Tables .... **vii**

Abstract .... **vii**

Chapter 1. Introduction **1**

Chapter 2. Related Work **10**

2.1 Necessity of accurate and comprehensive vulnerability estimation 10

2.2 Vulnerability estimation for cache memory 13

Chapter 3. Our Approach **16**

3.1 gemV: Fine-grained and comprehensive vulnerability estimation 16

3.1.1 Fine-grained modeling 18

3.1.2 Modeling with both committed and squashed instructions 21

3.1.3 Comprehensive modeling 24

3.1.4 Modeling based on accurate and flexible gem5 simulator 25

3.1.5 Validated modeling 26

3.2 Accurate cache vulnerability estimation at a word-level granularity 30

3.2.1 Vulnerability estimation at a block-level granularity is inaccurate . 33

3.2.2 In-depth analysis of inaccurate block-level cache vulnerability estimation 40
3.2.3 Validation with fault injections 45
3.2.4 gemV-cache implementation 46
Chapter 4. Experimental Observations 52
4.1 gemV for fast and early design space exploration 52
4.1.1 gemV for hardware implementation 53
4.1.2 gemV for software development 62
4.1.3 gemV for system design 63
4.2 Tricky cache protection techniques 65
4.2.1 Incomplete parity checking achieves efficient protection 65
4.2.2 Fine-grained status-bits maximize the achieved parity protection . 71
4.2.3 ECC protection can be vulnerable for single-bit flips 75
Chapter 5. Conclusion 82
References 84
Abstract in Korean 93

List of Figures

Figure 1.1 Thesis overview: Comprehensive resiliency estimation with considering protection techniques 3

Figure 3.1 Fine-grained vulnerability tracking for pipeline queues for simple instructions such as load (red), add (blue), and store (green) 17

Figure 3.2 Inaccuracy of coarse-grained vulnerability estimation as compared to fine-grained one 20

Figure 3.3 History buffer should consider not only committed instructions but also squashed instructions for accurate vulnerability estimation 22

Figure 3.4 More than half of the vulnerability (i.e., vulnerabilities of pipeline queues and register renaming units) has not been considered in previous frameworks 25

Figure 3.5 Example demonstrating the vulnerability of a *data*, over different data accesses 31

Figure 3.6 Block-level and word-level vulnerability estimation exemplary scenarios without protection techniques 34

Figure 3.7 Inaccuracy of block-level CVF estimations. Block-level vulnerability estimation is up to 121% inaccurate for the benchmark *basicmath*. 36

Figure 3.8 CVF based on word-level modeling is proportional to the dirty state in general since vulnerability is mainly comes from eviction at dirty state. 38

Figure 3.9 Dramatic difference of block-level and word-level CVF for each block. If the vulnerability of a cache block is estimated based block-level modeling, it can be 5,700% inaccurate as compared to accurate word-level one. 40

Figure 3.10 SAD (Sum of Absolute Difference) is the sum of overestimation and underestimation of inaccurate block-level estimation as compared to the accurate word-level estimation. SAD can show the realistic inaccuracy of block-level estimation. 42

Figure 4.1 Architectural vulnerability factor among several benchmarks. AVF can vary from 7% to 16% by changing benchmarks. 53

Figure 4.2 Vulnerability and runtime show the same trend by changing issue width, but vulnerability is more sensitive than runtime. 54

Figure 4.3 LSQ size should be considered with both vulnerability and runtime. Vulnerability is slightly increasing with the increase of LSQ size, while runtime is decreasing. 55

Figure 4.4 Different hardware configurations generates interesting design space in terms of runtime and vulnerability. Vulnerability can be reduced by up to 81% with less than 1% runtime overhead by varying hardware configurations. 57

Figure 4.5 Vulnerability and runtime with different hardware configurations (*matrix multiplication*). Given hardware configuration, vulnerability can be reduced by up to 37% with less than 1% performance overhead by changing hardware configuration. 58

Figure 4.6 Vulnerability can be reduced by up to 56% within the same number of sequential elements. 60

Figure 4.7 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 by software changes. 61

Figure 4.8 Variation in runtime and vulnerability for *stringsearch* under different ISAs. Bars show vulnerability and diamond points indicate runtime . . 64

Figure 4.9 Vulnerability estimation scenarios with diverse parity checking protocols 66

Figure 4.10 Incomplete read parity checking (checking only at reads) achieves the highest resiliency among various parity checking protocols. Complete parity checking is more vulnerable than incomplete read parity checking even with the additional redundancy 68

Figure 4.11 In the design of a parity-protected cache, the power overheads caused by parity checking at reads are 30% lower than that when parity is checked on both reads and writes. 70

Figure 4.12 Vulnerability estimation examples with diverse status-bit configurations. Note that the granularity of dirty bit does not affect the vulnerability if a parity bit is implemented on block-level 71

Figure 4.13 Fine-grained parity with block-level dirty-bit reduces the vulnerability by only 2% as compared to block-level parity and dirty-bits. Fine-grained dirty-bit along with parity-bit per word is the best in terms of vulnerability (60% reduction). 74

Figure 4.14 Vulnerability estimation examples with diverse status-bit configurations. Note that checking ECC-bits at read operations is more vulnerable than that at both read and write operations. 75

Figure 4.15 Incomplete read ECC checking does not remove the vulnerability completely, while complete ECC checking provides zero vulnerability. . . . 77

Figure 4.16 Vulnerability estimation examples with diverse status-bit configurations on ECC protection. Note that ECC-bits are checked at just read operations. 78

Figure 4.17 CVF with ECC protections are affected by the granularity of ECC-bits. 80

List of Tables

Table 2.1 Comparison between vulnerability estimation tools 11

Table 3.1 gemV validation against exhaustive fault injection campaigns. 300 faults injected per component for each of the following benchmarks: *matrix multiplication*, *hello world*, *stringsearch*, *perlbench*, *gsm*, *qsort*, *jpeg*, *bitcount*, *fft*, and *basicmath* 28

Table 3.2 Validation of our models and implementation of word-level vulnerability estimation. For all the selected words, we can get the perfectly matched vulnerability as compared to the failure rates through exhaustive fault injection campaigns 51

Table 4.1 Effects of software configuration(algorithm, optimization level, and compiler) on runtime and vulnerability (*sorting*) 63

ABSTRACT

Comprehensive Resiliency Evaluation for Dependable Embedded Systems

Yohan Ko
Department of Computer Science
The Graduate School, Yonsei University, Seoul, Korea

When we consider a broad range of embedded systems, it is essential to consider multiple design parameters, such as performance, power, and even resiliency. A low power design is just as important as high performance since state-of-art embedded systems run on limited capacity batteries with a small form factor. In order to meet both requirements, the supply voltage is lowered through the aggressive technology scaling. However, decreasing the supply voltage only increases the vulnerability of the systems due to soft errors, which are transient faults induced mainly by energetic particles such as neutrons, protons, and even cosmic rays. In order to make mobile embedded systems resilient against soft errors, several redundancy-based techniques have been presented, but they lead to significant overheads in terms of performance, power consump-

tion, and hardware area. Selective protections have been presented as an alternative to cost-effective protections, but how can we ensure whether it is useful or not? We can estimate overheads in terms of runtime, energy, and area, but it is challenging to estimate resilience in a quantitative manner.

In order to perform early design space explorations, we have implemented gemV-tool, which estimates the resiliency of microarchitectural components in a processor based on a cycle-accurate gem5 simulator. If we can quantitatively estimate resiliency, it enables us to answer fundamental design questions such as: (i) Can hardware architects improve the resiliency by just configuring hardware options with comparable performance overheads? (ii) Can software engineers improve the hardware-level resiliency against soft errors? (iii) System designers can alternate ISAs, but how can they ensure that protection mechanisms for the previous ISA still works for alternative ISA?

Further, our framework can also provide the protection guideline since we can estimate the resiliency with considering protection techniques. In this work, we provide the protection guideline of parity-protected level 1 data cache for high-level resiliency with comparable overhead. First off, checking parity at reads only (and not at writes) provides better protection with fewer power overheads as compared to that at both reads and writes. Secondly, When implementing parity at the fine-grained granularity for much-improved protection as compared to coarse-grained parity implementation, the dirty-bits in the cache should also be applied at the same fine-grained granularity. Otherwise, there is no improvement in protection.

Key words : Resiliency, Soft error, Vulnerability, gemV

Chapter 1

Introduction

A soft error is a transient fault in semiconductor devices caused by some sources both internal and external to the chip. Energy carrying particles such as alpha particles, protons, low-energy neutrons, even cosmic rays contribute soft errors significantly [1, 2]. The critical charge is the minimum charge causing soft errors, and it is proportional to chip size and supply voltage. Since soft error rate is inversely proportional to critical charge, the soft error rate is exponentially increasing with aggressive technology scaling. Even though soft errors are not the permanent hardware malfunction, they can be essential even for human life. Embedded systems can be used for safety-critical applications such as automotive [3] and health-care systems.

Many techniques have been presented in various design layers to protect computing systems against soft errors for several years [4]. These protection methods incur overheads in terms of area, performance, and energy consumption since they are based on hardware redundancy or software redundancy. However, protection schemes are neither always useful nor continuously robust against soft errors, and sometimes they can fail to protect systems even with additional overheads [5]. Thus, protection techniques for embedded systems should be carefully chosen by considering trade-off relationship be-

tween resiliency and performance. Performance can be estimated by the runtime or the number of instructions executed per cycles, but the resiliency cannot be easily quantified in an accurate and timely manner.

In order to accurately calculate resiliency of microarchitectural components, neutron beam testing [6] and fault injection campaigns [7] have been exploited to quantify the resiliency against soft errors. Beam testing uses the cyclotron to expose computing systems to neutron-induced soft errors. In fault injection campaigns, faults are intentionally injected into the specific bit of the microarchitectural components in a processor at the particular time during the execution time. Since exhaustive fault injection campaigns need to inject faults into all the bits of the entire computing system at every cycle of the execution time, they are almost impossible [8]. Statistical fault injections based on probability theory have been presented to reduce the number of experiments [9]. However, the accuracy of statistical fault injection campaigns still relies on the number of injected faults. Further, fault injection campaigns and beam testing are costly and difficult to set up correctly, and they are often flawed [10, 11].

Since neutron beam testing and faults injections are too expensive and slow, a metric *vulnerability*, which is the number of bits which can incur system failures during the execution time in the processor [12, 13], has been presented as an alternative. Assume that a specific bit b in a microarchitectural component is written at time t, and it is read by CPU at time $t + n$. In this scenario, bit b is not vulnerable before t. If there are soft errors before write operations, they can be overwritten to a new value. However, read operations can make vulnerable periods since CPU can read corrupted data. Thus, bit b is vulnerable during the time interval between t and $t + n$. Vulnerability is estimated as

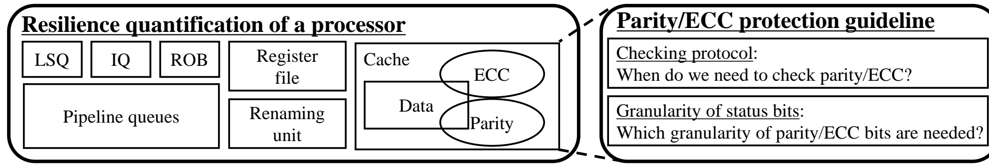


Figure 1.1: Thesis overview: Comprehensive resiliency estimation with considering protection techniques

$b \times n$ in this example, and its unit is $bit \times cycle$. The vulnerability of the entire processor is the summation of these vulnerabilities of all the microarchitectural components. Vulnerability estimation can be performed in a single simulation unlike fault injections since it can be done by tracing architectural behaviors of each component.

Several vulnerability modeling frameworks based on cycle-accurate simulators have been presented [13, 14, 15] in order to implement vulnerability modeling for a processor. However, their modeling is inaccurate, incomprehensive, and inflexible. First off, previous schemes cannot provide the accurate vulnerability estimation since they estimate the vulnerability at a coarse-grained granularity. Further, their modelings ignore the vulnerability of speculatively executed instructions (i.e., squashed instructions due to the misspeculation), as their presence in the pipeline can, in some cases, cause failures. Moreover, the accuracy of vulnerability from these tools has not been validated and published. Secondly, existing vulnerability modelings are not comprehensive since they have modeled the just subset of microarchitectural components in a processor. Lastly, previous modelings cannot provide configurable vulnerability estimation, such as various ISAs and multi-core systems, because of underlying simulators used.

In this manuscript, we present gemV: a tool for accurate, validated, and comprehensive vulnerability estimation based on gem5 [16] – a common cycle-accurate system-

level simulator [17] as shown in Figure 1.1. For example, gem5 explicitly models most of the microarchitectural components of an out-of-order processor, various ISAs (e.g., ARM, ALPHA, etc.), multicore processors, and even many system calls. Also, some of the key features of gemV that enable accurate vulnerability estimation are: (i) fine-grained modeling of hardware components through the use of RTL abstraction inside gem5 simulator, (ii) correctly modeling the vulnerability of both committed and squashed instructions. Moreover, exhaustive fault injection campaigns validate gemV to 97% accuracy with 90% confidence level. gemV also provides comprehensive vulnerability modeling for all the microarchitectural components of out-of-order processors.

gemV presents the efficient toolset for early design space exploration of resiliency in the presence of soft error failures. It enables us to answer fundamental design questions from many different perspectives. (i) **Microarchitecture designer**: Is a dual-issue processor more vulnerable than a single-issue processor? How does altering the issue width of the processor affect vulnerability? Reducing the issue width mitigates the number of vulnerable bits at a given time, but it could also increase the runtime. Since the vulnerability is related to runtime and hardware bits, the effect of varying the issue width can only be answered through quantitative experiments. In the same vein, can we decrease the vulnerability by just changing hardware configurations with comparable performance? (ii) **Software system designer**: Can software system designers improve the hardware-level resiliency against soft errors? In a program, the algorithm, the optimization level of the compiler can also affect the runtime and vulnerability. (iii) **Architecture designer**: Architecture designers can alternate ISAs for better performance, but how can they ensure that protection mechanisms for the previous ISA still work for

alternative ISA? The trade-offs between runtime and vulnerability can now be answered rapidly and accurately by using the gemV toolset.

In our demonstrations of the capabilities of gemV, we perform a broad range of design space explorations and observe that:

- Vulnerability decreases when increasing issue width from 1 through 3 for a benchmark. Beyond this, any increase in issue width does not have a noticeable effect on vulnerability. We also find that vulnerability varies by changing architectural parameters like the number of entries in reorder buffer (ROB), an instruction queue (IQ), load/store queue (LSQ), and pipeline queues. Among configurations, there is an interesting design configuration with 82% less vulnerability at most 1% performance penalty.

- A software designer can also use gemV to find the least vulnerable algorithm for a program. For example, we show that switching from a selection sort to a quicksort algorithm can affect the system vulnerability by 91% with the fixed configurations.

- With the perspective of system designers, it is interesting that the distribution of vulnerabilities among microarchitectural components is sensitive to the ISA. While protecting register rename map and register file will be the most effective in SPARC architecture (more than 75% vulnerability reduction), but the protection will only reduce the vulnerability by 21% in ARM architecture. In contrast, protecting history buffer and IQ will be the most effective in ARM architecture in our study.

Further, our framework can also provide the vulnerability with applying protection

schemes so that we can achieve protection guideline for each component. In a processor, a cache is one of the most sensitive microarchitectural components to soft errors [18]. Mitra et al. [19] note that soft errors in caches (unprotected SRAMs) contribute to around 40% in processors, and Shazli et al. [20] have shown that 92% of machine checks are triggered by soft errors at the level 1 and 2 caches. It is not only because caches occupy the majority of the chip area, but also because they have high transistor density and operate at low voltage swings [21]. Since CPU frequently accesses data in caches and written back to lower-level memory in case of write-back caches, some of the erroneous bits can be propagated to the lower-level memory or used by CPU. However, not all the soft errors in the cache memory can cause system failures (i.e., vulnerable) during all the execution time mainly due to several masking effects. Thus, there is a necessity to quantify the susceptibility of caches to know how many bits and how long cache data can be vulnerable.

Architectural vulnerability used to denote the resiliency of a single architecture component, while vulnerability is used to indicate that of the entire processor. In this manuscript, we use the term cache vulnerability to denote the architectural vulnerability of the cache since we analyze the cache resiliency as the domain of protection guideline. Cache vulnerability estimation at a block-level granularity is entirely inaccurate since the basic unit size of data accesses in caches is a word, not a block. For instance, the particular cache word is vulnerable when CPU reads just a single word of a block. However, block-level vulnerability estimation defines the whole block as vulnerable, not just the particular word. The average inaccuracy of block-level estimation is 37% as compared to more accurate our word-level one. Note that our word-level vulnerability

estimation includes byte-level granularity since we analyze word-level cache behaviors for vulnerability estimation. The average inaccuracy is not significant, but the actual wrong decision based on block-level behaviors can be worsened. It is because that the difference is aggregated statistics of entire cache blocks during the whole execution time. First off, block-level analysis can underestimate or overestimate vulnerability as compared to the word-level one, but the inaccuracy only can show the difference between the underestimation and overestimation. Secondly, the error of each block can be much larger than the average error of all the blocks. For example, the error of a particular block is up to 5,700%, while the mean error of all the blocks is only 121% for the same benchmark, *basicmath*.

Existing cache vulnerability estimation schemes also ignore protection techniques even though several methods have been presented for resilient cache memory. These techniques span the design spectrum from the circuit, microarchitecture, software, and even hybrid level. In practice, parity and error correction code (ECC) are the most popular cache protection techniques due to their design simplicity. Parity-based methods allow the error recovery by bringing data from lower-level memory as long as cache data is not updated by the processor (i.e., clean state). ECC-based techniques provide the error recovery regardless of the clean or dirty state. However, it can incur up to additional 50% hardware area, more than five times power consumption, and about 115% runtime overheads as compared to unprotected cache [22]. Parity protection is preferred for higher-level (e.g., level 1) caches while ECC protects lower-level caches (e.g., level 2 or other lower level caches) in common. There are several design choices when we implement parity and ECC protection, for example: When should we check for parity-

bit and ECC-bits – at read, write, or both read and write? At what granularity should we have parity-bit and ECC-bits? At what granularity should we have dirty-bit?

In order to correctly answer these questions, we need techniques to quantitatively and accurately estimate the susceptibility of cache data to soft errors with or without protection methods. We have validated the accuracy of our word-level estimation by extensive fault injection experiments. The logic to estimate vulnerability at a word-level granularity with the presence of protection techniques is much more involved than the logic to estimate vulnerability at a block-level granularity without considering protections. The primary source of complexity comes from the fact that i) the access time of each word should be logged for word-level estimation while the access time for a block is needed for block-level estimation; ii) vulnerability estimation at a word-level granularity may not be independent of the accesses of the other words in the same block.

The contribution of this manuscript includes accurate word-level vulnerability modeling and awareness of protection techniques as shown in Figure 1.1. First off, we have modeled more accurate word-level vulnerability modeling than previous block-level one since the basic unit of cache accesses is a word, not a block. Moreover, we have also validated our vulnerability modeling against exhaustive fault injection campaigns. Secondly, we have modeled cache vulnerability estimation without and with general protection techniques such as error detection codes (parity) and error correction codes (Hamming code). We explore the design space of parity and ECC protections with various protection configurations based on accurate word-level vulnerability estimation. Our analysis reveals several interesting and counterintuitive results for cache protection techniques.

- Checking parity at reads provides the better level of protection than checking par-

ity at both reads and writes. It is surprising since it is more intuitive to believe that checking parity on both occasions will provide better protection mainly due to more redundancy. The implication is that better protection can be achieved by simpler hardware and less overhead of parity checking power.

- In order to achieve higher levels of protection, both parity-bit and dirty-bit should be implemented at word-level of granularity. It can reduce the vulnerability by 60% as compared to the vulnerability without protections. However, only either parity-bit or dirty-bit at a word-level granularity does not protect caches efficiently, i.e., it can reduce the vulnerability by just 15% on average as compared to unprotected caches despite additional hardware overheads.

- Checking block-level ECC-bits only at reads can be still vulnerable because of other words' behaviors in the same block. About 10% of vulnerability comes from unprotected caches remains with checking at reads, while checking at both reads and writes provides zero vulnerability. If the perfect resiliency is required for caches, ECC should be checked at both reads and writes, or ECC-bits should be implemented at a word-level granularity.

Chapter 2

Related Work

2.1 Necessity of accurate and comprehensive vulnerability estimation

With a view to estimating the vulnerability for all microarchitectural components in a processor, previous works have exploited cycle-accurate, system-level, and software-based simulators as described in Table 2.1. Mukherjee et al. [13] proposed AVF (Architectural Vulnerability Factor) based on Asim [23] which simulates Itanium2-like IA64 processors. Li et al. [14] proposed SoftArch which models the error generation and propagation based on the probabilistic theory in Turandot simulator [24]. Sim-SODA [15] has been proposed to estimate the vulnerability of microarchitectures based on Sim-Alpha simulator [25]. However, previous works are inaccurate, incomprehensive, unavailable for public use, and inextensible.

First off, most of the existing techniques have estimated the vulnerability at a coarse-grained granularity although not all bits of a hardware structure are vulnerable for every instruction. In [13, 14], complex hardware structures in out-of-order processors such as IQ are modeled as bulk structures. For instance, the predicted next PC address is not vulnerable since it can only affect the performance by branch misprediction. On the other

Table 2.1: Comparison between vulnerability estimation tools

Tool	Accuracy	Comprehensiveness	Extensibility	Validation
Mukherjee-AVF [13]	Not accurate: Instruction window is treated as a coarse-grained bulk Only committed instructions are considered for vulnerability modeling	Register file and instruction queue are modeled for vulnerability estimation	IA-64 based architecture based on proprietary Asim [23] simulator	No published results
SoftArch [14]	Not accurate: Instruction window is treated as a coarse-grained bulk Only committed instructions are considered for vulnerability modeling	Register file and instruction queue are modeled for vulnerability estimation	Power-PC architecture based on proprietary Turandot [24] simulator	No published results
Sim-SODA [15]	Not accurate: Several hardware structures in the instruction fetch and issue logic are modeled as a single hardware structure Only committed instructions are considered for vulnerability modeling	Register file, instruction queue, reorder buffer, and load store queue are modeled for vulnerability estimation	ALPHA architecture based on open-source Sim-Alpha [25] simulator	No published results
gemV (Our proposal)	More accurate: Every structure is modeled based on fields that are really used (Section 4.2.2) Squashed instructions are also considered for vulnerability modeling (Section 3.1.2)	Register file, instruction queue, reorder buffer, load store queue, pipeline queues, and renaming units are modeled for vulnerability estimation (Section 3.1.3)	ARM, ALPHA, Power-PC, MIPS, X86, SPARC architectures with various configurations based on open-source gem5 [16] simulator (Section 3.1.4)	Validated through extensive fault injection (Section 3.2.3)

hand, the current PC address is vulnerable since it can cause incorrect program flow. In [15], several hardware structures in the instruction fetch and issue logic are modeled as a single hardware structure – ”instruction window.” They do not model individual hardware structures such as pipeline queues, instruction queue, and load/store queue. Thus, these components cannot be evaluated for the vulnerability modeling while gemV can estimate the vulnerability at a fine-grained granularity as described in Section 4.2.2.

Secondly, squashed instructions are ignored for the vulnerability estimation in previous works. An instruction can be “squashed” due to the misspeculation in an out-of-order processor. Under these conditions, most bits used by the instruction are considered not vulnerable, but individual bits can be still vulnerable. For instance, rename map holds the index mapping between architectural and physical registers. The rename map uses a history buffer to maintain the previous mapping of an architectural register. It is why when an instruction is squashed; the processor state can be rolled back to the last committed instruction. When an instruction is squashed, the history buffer can be vulnerable since it is read to roll-back the rename map. However, previous vulnerability estimation tools consider all squashed instructions to be not vulnerable, but gemV considers both committed and squashed instructions for vulnerability modeling as described in Section 3.1.2.

Thirdly, previous tools are incomprehensive in their vulnerability modeling since they estimate the vulnerability of just a small subset of the microarchitectural components of the processor. In [13, 14], they do not model the vulnerability estimation for register files, memory hierarchy, and pipeline structures. Sim-SODA considers more microarchitectural components than the other estimation tools, but it still does not model

the vulnerability estimation for pipeline queues and renaming units which contribute the system vulnerability significantly as described in Section 3.1.3.

Lastly, previous tools are inflexible and inaccurate due to the limitations of simulators they use. Vulnerability estimation techniques in [13, 14] use the proprietary and private tools which model Intel's Itanium 2-like processor and IBM's Power-PC, respectively. Sim-SODA estimates the vulnerability based on publicly available Sim-Alpha simulator, but it is limited to ALPHA and single-core processors. Moreover, the accuracy of vulnerability estimation can be suffered from inaccurate simulation since their modelings are based on simulated behaviors of components. Sim-Alpha has been shown to be up to 43% inaccurate in runtime estimations [26] as compared to real hardware architecture. On the other hand, gemV can provide the flexible and accurate vulnerability modeling by leveraging gem5 simulator as described in Section 3.1.4.

2.2 Vulnerability estimation for cache memory

Cache memory is one of the most vulnerable microarchitectural components in processors against soft errors. It is not only because those caches occupy lots of area in processors, but also because CPU frequently accesses that cache data and quickly propagated to lower-level memory. In order to improve the resiliency of cache memory without area cost, Li et al. [27] proposed early write-back policy. Early write-back policy combines the performance efficiency of write-back with the resiliency of write-through policy by exploiting the least recently used algorithm or dead-time based approaches. Manoochehri et al. [28] proposed the correctable parity protected cache (CPPC) to correct errors which can be detected by parity. CPPC corrects soft errors including spatial

multi-bit errors at the dirty state by multi-dimensional parity-bits without the severe overhead in terms of hardware area and performance. However, they can be still vulnerable to temporal multi-bit upsets and errors in the cache tag array and status bits such as dirty-bits.

Soft errors on variables do not induce system failures due to the software masking effects, e.g., errors in multimedia data in a program can degrade the quality of service, but they do not result in system failures. PPC (partially protected cache) [29] improved the resiliency with the comparable performance overheads by enhancing the software masking. PPC only protects failure-critical data such as control variables based on data profiling at the compile time. On the other hand, they do not protect multimedia data since errors on multimedia data cause loss in quality of service instead of system failures. Smart cache cleaning [30] protects specific cache blocks at specific periods by applying the hardware-software hybrid methodology. At the software level, we can protect data efficiently by software-based or hybrid-based selective protection, but the decision of importance in data is an incredibly complex task.

In order to mitigate the resiliency analysis overheads of cache memory and to provide the accurate resiliency reflecting various masking effects, CVF is proposed based on cache access patterns [31, 32]. Data in a write-back cache is vulnerable, if it will be read by the processor, or will be written back (e.g., eviction of a dirty cache line) into the memory. If it is overwritten or just discarded (e.g., eviction of a non-dirty cache line), then it is not vulnerable. In a system, the resiliency metric – *vulnerability*, is a measure of the probability of soft errors during the period when data is exposed in the cache which is predominantly dependent on the data access pattern of the program.

Vulnerability estimation of a cache block can be implemented at two granularity levels: **a) block-level** – when every access to a word in the cache-block, is considered to be an access to the whole block or every word in the cache-block has the same data access; **b) word-level** – when every access to a word in the cache-block, is considered as an access to each respective word in the block. In a cache-block composed of multiple words, the total vulnerability of the block is an accumulation of the vulnerabilities of the individual words in the block; which is based on the data access patterns of the words in the cache-block.

However, how can we measure the resiliency of caches without protections accurately? How much do these protection techniques afford as compared to the resiliency without protections? Thus, there is a necessity to quantify the susceptibility of caches against soft errors without protection or even with protection techniques. Further, we also need to implement vulnerability modeling for other microarchitectural components including cache memory to explore design space in terms of power consumption, performance, and resiliency.

Chapter 3

Our Approach

3.1 gemV: Fine-grained and comprehensive vulnerability estimation

A vulnerability has been used as an alternative metric for the failure rate of architectural components against soft errors. A bit b in a microarchitectural component at the specific time t during execution time is vulnerable if a soft error into (b, t) may result in system failure. If not, (b, t) is not vulnerable. The vulnerability is the sum of these vulnerable bits in microarchitectural components of a processor. The unit of vulnerability is $bit \times cycle$ in order to consider both time and space domains. Assume that 2 bits in a microarchitectural component are vulnerable during five cycles. The vulnerability of this microarchitectural component is 10 bit $\times$ cycles (= 2 bits $\times$ 5 cycles). In a processor, a bit which may induce failures should be tracked to estimate the vulnerability based on behaviors of microarchitectural components.

In this manuscript, we have implemented gemV-tool, which estimates vulnerability for microarchitectural components in a processor based on the cycle-accurate gem5 simulator. We have named our vulnerability modeling frameworks "gemV-tool" due to two following reasons. "V" of gemV-tool stands for both vulnerability and Roman numeral 5

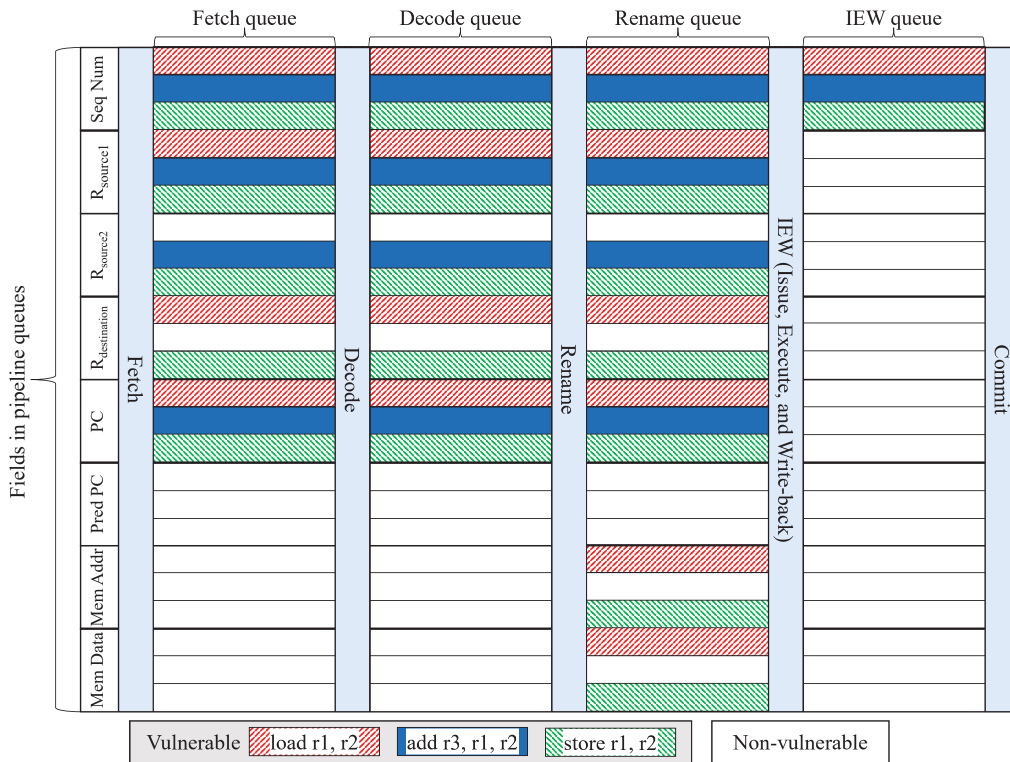


Figure 3.1: Fine-grained vulnerability tracking for pipeline queues for simple instructions such as load (red), add (blue), and store (green)

(5 from gem5). In modeling of gemV-tool, we consider single-bit soft errors throughout a program execution in caches for simplicity. The system vulnerability is the sum of vulnerabilities of all the microarchitectrual components in a processor. We use the ARM v7a processor architecture and have compiled our suite of benchmarks using GCC cross-compiler for ARM (ver. 4.6.2), run them on gemV-cache in system emulation mode, and gathered vulnerability statistics in just one simulation.

3.1.1 Fine-grained modeling

Fine-grained modeling is important because not all the bits of a hardware structure are vulnerable at the same time. Thus, vulnerability modeling should consider accessed bits for each microarchitectural component. Figure 3.1 shows the fine-grained vulnerability estimation for pipeline queues for simple instructions such as load (load r1, r2), add (add r3, r1, r2), and store (store r1, r2). Pipeline queues (fetch, decode, rename, and IEW: issue, execution, and writeback) hold the information of each instruction between pipeline stages. For example, fetch queue holds the data which will be used by the decode stage. Pipeline queue contains sequence number of instructions (SeqNum), source register index (R source 1 and 2), destination register index (R destination), PC, predicted next PC (PredPC), memory address (MemAddr), and data (MemData). In Figure 3.1, load instruction (load r1, r2) updates the data in r1 by accessing memory address in r2. And, r3 is updated by the addition of r1 and r2 through an add instruction (add r3, r1, r2). Store instruction (store r1, r2) updates the memory address r1 with the data stored in r2.

First off, our fine-grained vulnerability estimation tracks just accessed fields in pipeline queues, not all the fields in pipeline queues. For example, all the pipeline queues hold the predicted next PC address since processors use branch prediction for better performance. Even though branch prediction is incorrect, it only affects the performance and does not induce failures. Thus, the predicted next PC is not vulnerable regardless of instructions. And, instructions determine vulnerabilities of accessed fields differently. The destination registers (r1 and r3, respectively) are vulnerable since they are updated by these instructions. However, store instruction does not update destination register,

and it does not have vulnerable periods in destination register fields. On the other hand, load instruction uses one source register (r2), and the second source register index is not vulnerable. Vulnerable fields can be different between ALU and memory instructions. In Figure 3.1, ALU instruction (add) does not access the memory-related fields (memory address and data), while memory instructions (load and store) have the vulnerable periods in these fields. In [33], for an ARM-v7a pipeline, 71 bits are vulnerable at the rename queues for ALU instructions, while 132 bits are vulnerable to memory-reference instructions.

Secondly, our fine-grained vulnerability estimation only tracks the vulnerable duration of accessed fields. For example, the just sequence number is vulnerable after IEW stage since the other fields are not used at the commit stage. For memory operations (load and store), memory address and data are not vulnerable from fetch to rename stages. It is because that the memory reference is calculated by accessing physical registers after the rename stage. Thus, memory address and data can be overwritten although bits in these fields are flawed before the rename stage. If we estimate the vulnerability at a coarse-grained level, all the fields in pipeline queues are defined as vulnerable from fetch to commit stages.

Fine-grained modeling is also essential for cache memory for accurate vulnerability estimation. In [34], block-level tracking of vulnerability in the cache can lead to significant error since the basic unit of cache behaviors is a word, not a block. Cache memory consists of several blocks, and each block is composed of several words. Data is brought into the cache memory (incoming) and evicted at the block-level while its write and read operations can occur at the word-level. However, coarse-grained modeling considers

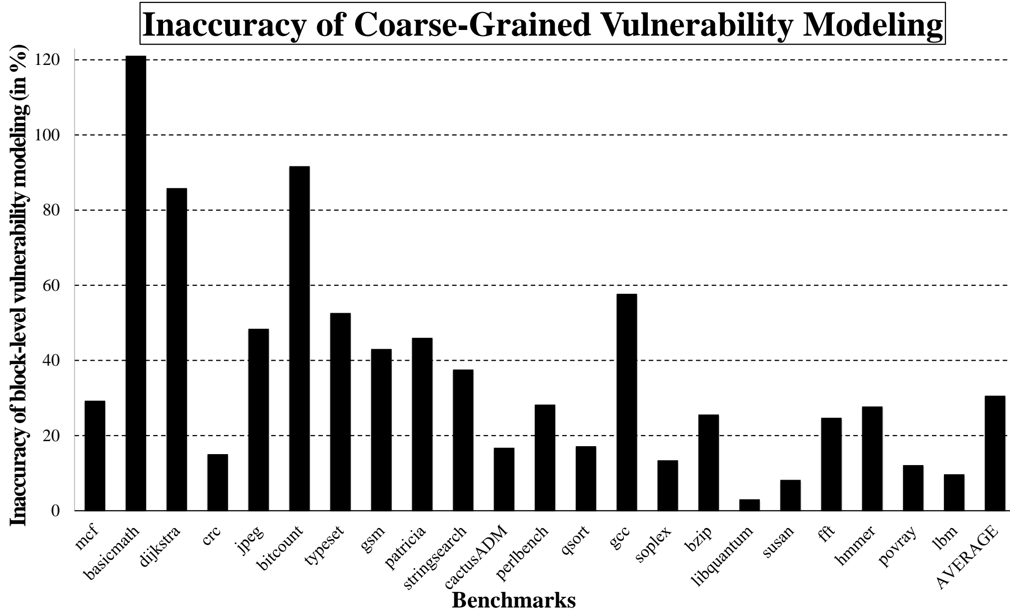


Figure 3.2: Inaccuracy of coarse-grained vulnerability estimation as compared to fine-grained one

every behavior in the cache memory as the block-level one, not the word-level one. On the other hand, our fine-grained vulnerability modeling tracks the word-level behaviors.

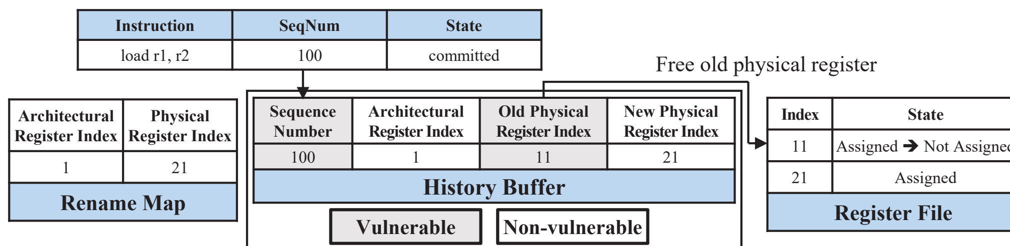
Overall, for the whole cache blocks, coarse-grained block-level vulnerability modeling can result in inaccurate estimation by 37% on average among several benchmarks from SPEC CPU2006 [35] and MiBench [36] suites as compared to the fine-grained word-level one as shown in Figure 3.2. Thus, the block-level cache vulnerability estimation can be incredibly inaccurate. Further, for a cache block, tracking vulnerability at the coarse-grained modeling overestimates its vulnerability by up to 57 $\times$ as compared to the fine-grained one for the benchmark *basicmath* in our study.

In order to achieve fine-grained vulnerability estimation in gemV, we instrument every hardware component modeled in the gem5 out-of-order processor with a *vulnera-*

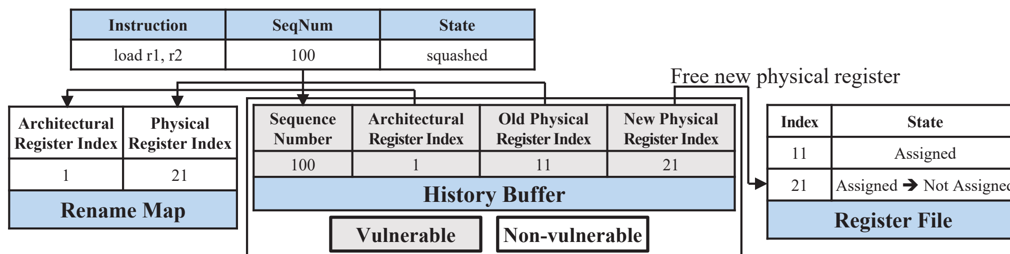
bility tracker, a data structure which tracks the read/write accesses on each field of each component and thereby computes their respective vulnerable periods at the fine-level granularity (bit-level). In our vulnerability tracker, with the knowledge of the type of instruction accessing the hardware, instruction specific vulnerability modeling can be applied. For instance, if an instruction is passing through the pipeline stage, the vulnerability tracker only tracks the vulnerable fields at the vulnerable time as shown in Figure 3.1. 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 vulnerable periods are computed accurately.

3.1.2 Modeling with both committed and squashed instructions

We also achieve accurate vulnerability estimation by handling the particular case of an instruction getting squashed. Previous works do not update the vulnerability in case of squashed instructions, but individual bits in specific microarchitectural components are still vulnerable. The rename map holds a mapping between architectural and physical registers. The rename map uses a history buffer to maintain the previous mapping of an architectural register. Figure 3.3 depicts the register renaming case for an exemplary instruction, load r1, r2, and currently architectural registers r1 and r2 are mapped to physical register index 10 and 20, respectively. For the source register, r2 in the rename map is accessed, and source physical register index remains to 20. For the destination register, r1 in the rename map is accessed, and then its physical register index, 10, is propagated to old physical register index in the history buffer. And, the destination register is newly mapped to 11, and new physical register index in the history buffer is updated with this renamed physical register index (11) since register renaming is needed



(a) If load instruction (load r1, r2) is committed, history buffer does not have to restore rename map



(b) If load instruction (load r1, r2) is squashed, history buffer must restore rename map

Figure 3.3: History buffer should consider not only committed instructions but also squashed instructions for accurate vulnerability estimation

to be recovered if the instruction is squashed.

The sequence number and old physical register index are vulnerable in case of committed instructions as shown in Figure 3.3(a). For the committed instruction, architectural register index and new physical register index in the history buffer are not vulnerable since rename map will hold the mapping between them. Thus, history buffer does not need to maintain them. However, sequence number and old physical register index in the history buffer are still vulnerable since this index will be accessed to free the corresponding physical register (errors in this field can free incorrect register). On the other hand, all the fields in the history buffer would be defined as vulnerable based on coarse-grained modeling. It can result in a 23% increase in vulnerability of history buffer estimated by coarse-grained methods as compared to fine-grained tracking for history buffer in a simple benchmark, *matrix multiplication*.

If the instruction is squashed, all the fields in the history buffer are vulnerable. Sequence number, architectural register index, and old physical register index are vulnerable since they are requested to undo the register renaming, 21 to 11 as shown in Figure 3.3(b). New physical register index is also vulnerable since it is accessed to free the corresponding physical register. Interestingly, sequence number and old physical register index in the history buffer are always vulnerable regardless of the instruction commitment (or squash). If we do not estimate the vulnerability in case of squashed instructions, we ignore 35% of the vulnerability of the history buffer as compared to the accurate modeling even for a simple benchmark, *matrix multiplication*.

For accurate and comprehensive estimations in gemV framework, we introduce data structure, $history$, to gem5 simulator to trace recent accesses of every field in the entry

of microarchitectural components with considering instruction commitment. Data structure $history$ consists of $tick$, $operation$, and $sequence\,number$. $tick$ has the timing information when an access to a microarchitectural component takes place. $operation$ holds a type of operations such as invalid, incoming, read, write, and eviction. $sequence\,number$ holds the order of instructions to trace whether an instruction is committed or squashed. gemV estimates the vulnerability of each component by keeping track of $history$ and analyzing behaviors and returns the system vulnerability as the sum of all the component vulnerabilities at the end of simulations.

3.1.3 Comprehensive modeling

gemV provides comprehensive vulnerability modeling since we have modeled the vulnerability of all microarchitectural components in out-of-order processors such as pipeline queues. We have also modeled the complete register renaming process between the physical register to architectural one by tracking behaviors and vulnerabilities in the rename map and history buffer with considering instruction commitment or not. Comprehensiveness is an outstanding quality to study the breakdown of vulnerabilities of a specific microarchitectural component as a portion of the total processor vulnerability. It 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. Figure 3.4 shows the breakup of processor vulnerability in the default configuration of gem5 ARM out-of-order processor running *stringsearch* benchmark. More than half of the total system vulnerability (54%) that we model has not been modeled in previous works (i.e., pipeline queues and renaming unit). Thus, gemV can provide the entire system-level vulnerability instead of

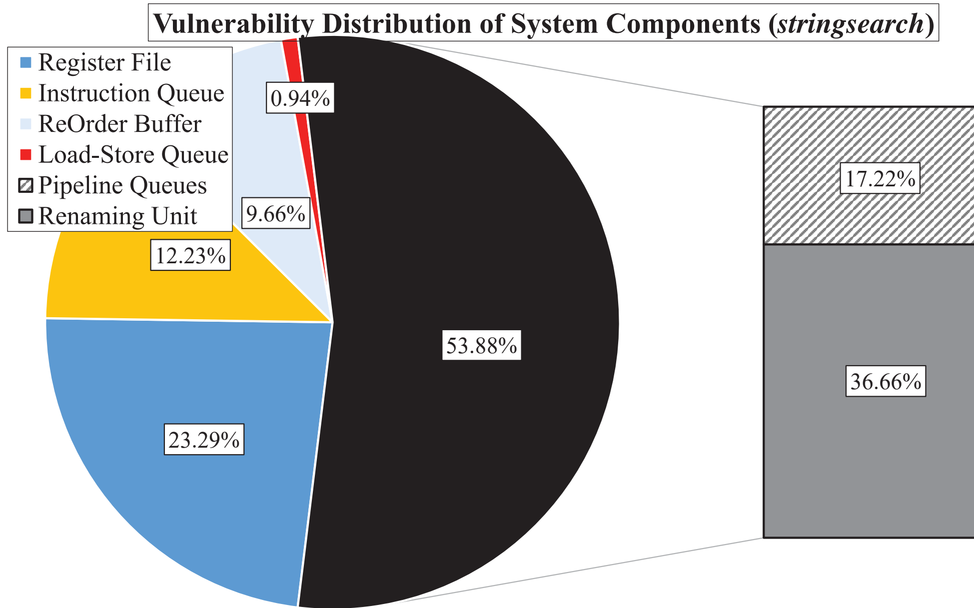


Figure 3.4: More than half of the vulnerability (i.e., vulnerabilities of pipeline queues and register renaming units) has not been considered in previous frameworks

the just sum of vulnerabilities of a subset of microarchitectural components.

3.1.4 Modeling based on accurate and flexible gem5 simulator

gemV can provide the accurate vulnerability modeling due to the accuracy of gem5 simulator. Since vulnerability modeling is based on simulated behaviors of each microarchitectural component, the accuracy of simulation affects that of vulnerability estimation. We have exploited gem5 simulator to implement the vulnerability modeling of out-of-order processors, and gem5 can provide up to 99% accuracy as compared to the real hardware board [17]. Moreover, gem5 simulator is updated actively by both developers and software engineers since it is based on open source infrastructure.

gemV can also perform flexible vulnerability modeling in its support for multiple

ISAs, multicores, and system call simulation. Due to this, gemV offers several advantages in vulnerability estimation over previous works. First of all, gemV can estimate vulnerability irrespective of the underlying ISA. It can be used in estimating vulnerability of the same program across different ISAs such as X86, ARM, SPARC, and ALPHA as demonstrated in Figure 4.8. Further, gemV can estimate the vulnerability of an application running on out-of-order processors in both single core and multi-core configurations.

Thus, gemV is capable of estimating vulnerability for commodity off-the-shelf (COTS) processors. We achieve this 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 [5] use parity protection for reads and writes in their caches. The vulnerability of programs running on such processors can be studied using gemV.

3.1.5 Validated modeling

In order to validate our vulnerability estimations in gemV, we perform extensive fault injection campaigns in all the microarchitectural components in gem5 as listed in Table 3.1. For each microarchitectural component, we inject a single bit-flip in a microarchitectural bit chosen at random, at randomly selected cycle per each execution of a program in gem5. We inject 300 faults per component for each of ten benchmarks from MiBench [36] and SPEC CPU2006 [35] for the comprehensive simulations. Note that gem5 simulator shares the same information of instructions among ROB, LSQ, and IQ, i.e., a bit flip into one component can affect the behaviors of all these three components.

Thus, we modify gem5 by duplicating fields in order to observe single bit flip's impact on a specific component exclusively.

In our fault injection campaigns, we run 300 simulations per microarchitectural component of each benchmark. Theoretically, 300 simulations are extensive enough to observe the statistical fault injection-based experiments regardless of an initial population size with the 90% confidence level [37]. Experimentally, we also validate that 300 runs can provide the stable results for all the components as compared to validation results with more than 300 runs. We have injected 1 through 2,000 single-bit flips randomly by incrementing 1 for each microarchitectural component into a benchmark. If the number of runs is smaller than 300, the accuracy is unstable. However, if the number of runs is equal to or larger than 300, the accuracy is stable. Indeed, the difference is less than 2% among all the runs over 300 in our simulations. Thus, 300 runs per microarchitectural component should be large enough to validate gemV with fault injection campaigns.

In our fault injection campaigns, we consider single bit faults, not multi-bit soft errors. With technology scaling, multiple-bit soft errors are increasing, and they should be considered for modern embedded systems. However, the multiple-bit error rate is much lower than that of single-bit errors. For instance, the soft error rate of double bit soft errors is just 1/100 as compared to that of single bit soft errors [38]. Thus, we do not consider multiple-bit soft errors in this manuscript for brevity's sake.

Out of 3,000 simulations for each microarchitectural components, we can observe matched or mismatched cases. For example, Table 3.1 shows 2,899 matched and 101 mismatched out of 3,000 fault injections for the register file. It is the matched case when a fault injection causes a failure and gemV returns that the fault-injected bit is

Table 3.1: gemV validation against exhaustive fault injection campaigns. 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**	**Matched Results**	**Mismatched Results**	**Accuracy (in %)**
Register file	3,000	2,899	101	96.63
Rename map	3,000	2,748	252	91.60
History buffer	3,000	2,781	219	92.70
Instruction queue	3,000	2,978	22	99.27
Reorder buffer	3,000	2,760	240	92.00
Load-store queue	3,000	2,979	21	99.30
Fetch queue	3,000	2,890	110	96.33
Decode queue	3,000	2,902	98	96.73
Rename queue	3,000	2,827	173	94.23
I2E queue	3,000	2,959	41	98.63
IEW queue	3,000	2,873	127	95.77
Overall Accuracy				96.78

vulnerable at the selected cycle or when a fault injection causes a non-failure, and it is non-vulnerable. Otherwise, it is the mismatched case. A simulation is declared as a failure in our experiments if either the system crashes or it results in the incorrect output. It is also a failure if the system halts due to a fault injection even though it returns the correct output. The vulnerability is estimated by gemV tool as described in the Section 3.1. Thus, the accuracy of validations of gemV with fault injection simulations for each component is defined as $\frac{The\,Number\,of\,Matched\,Cases}{The\,Total\,Number\,of\,Simulations}$. 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, we observe 2,899 matched ones and 101 mismatched results for the register file, giving us an accuracy of 96.63%.

Note that we need to adjust our validation numbers in order to calculate the overall accuracy of validations of gemV for the entire microarchitecture in a processor. We suppose that soft error rate is proportional to the size of each component, and so is the rate of fault injections. For instance, if the sizes of component A and B are 99 and 1, respectively, the soft error rate of A is 99 times larger than that of B. Under the assumption that accuracies of vulnerability estimations in A and B is 50% and 10%, respectively, the overall accuracy of A and B is $\frac{0.5\times99+0.1\times1}{99+1} = 49.60(\%)$, not $\frac{0.5+0.1}{2} = 30(\%)$. Thus, it is fair to define the overall accuracy of validations of gemV as $\frac{\sum_{Component=k}^{All\ Components} Size_k \times Accuracy_k}{Total\ Size}$ by considering the feature of soft error rate. Table 3.1 lists the results of our fault injection experiments for each microarchitectural component. The results show that component vulnerability estimated using gemV is about 97% accurate.

Vulnerability estimation from gemV seems highly accurate for all the microarchitectural components in processor since benchmarks have been chosen in order to minimize the software-level masking effects. However, there still exists 3% inaccuracy as compared to fault injections due to following reasons even though register data is read by committed instructions. First off, dynamically dead instructions can induce the mismatched case. If the result of an instruction is not used any more, this instruction is dynamically dead [13]. And, dynamically dead instructions do not affect the program output any longer. Thus, it is possible whether gemV concludes specific fields for a component when an instruction resides are vulnerable while this instruction does not cause the failure since it is dynamically dead.

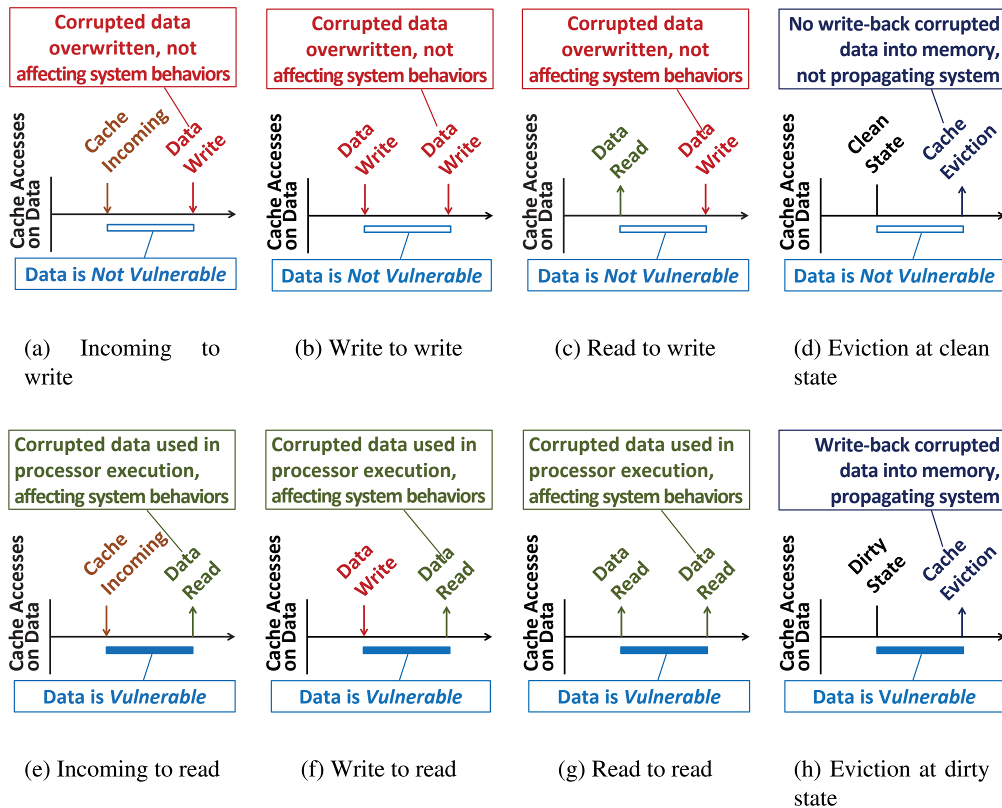
Secondly, vulnerability estimation can be mismatched with the failure rate due to

the masking effects from logical instructions [13]. Assume that the result of a logical AND instruction of values in two registers goes to the destination register. If the value of one input register is 0, then the result of this logical AND is always 0 even though fault injections might change the value in the other input register. On the other hand, the result of OR operations is always one if one input register data of OR instruction is 1.

Lastly, incorrect program flow can make vulnerability estimation mismatched since it is considered vulnerable in gemV while it can still result in the correct output in fault injections. For example, soft errors on PC address or branch target address can induce the incorrect program flow, but it may not affect the final program output in certain cases [39]. Our analyses reveal several mismatched cases as stated above such as dynamically dead instructions, logical masking effects, and uninfluential program flows. If we exclude these mismatched cases for the accuracy calculations, gemV can be more matched causing higher precision.

3.2 Accurate cache vulnerability estimation at a word-level granularity

In order to accurately estimate the cache vulnerability, we have implemented vulnerability estimation tools with protections for caches, named gemV-cache [5] by extending gemV-tool. The vulnerability of a cache block is the sum of vulnerable periods in cycles of all cache words from incoming through eviction. Thus, the vulnerability of a cache is the sum of vulnerabilities of all cache blocks during a program execution, and the unit of cache vulnerability is byte $\times$ cycle. We have performed extensive experiments with gemV-cache over benchmarks from MiBench [36] and SPEC CPU2006 [35] suites. We use the ARM v7a processor architecture with default L1 cache configuration as direct-



(a) Incoming to write

(b) Write to write

(c) Read to write

(d) Eviction at clean state

(e) Incoming to read

(f) Write to read

(g) Read to read

(h) Eviction at dirty state

Figure 3.5: Example demonstrating the vulnerability of a *data*, over different data accesses

mapped 4 KB with 64-byte block size. It is just one set of parameters for our simulation studies. gemV-cache is configurable as is the gemV-tool framework. For instance, we can also configure ISAs and number of cores which are not directly related to cache configurations.

In a system, the resiliency metric – *vulnerability*, is a measure of the probability of soft errors during the period that data is exposed in the cache, which is predominantly dependent on the data access patterns of the program as shown in Figure 3.5. First off, data is brought into the cache memory (incoming). If data is written by write operations after

incoming, it is not vulnerable as shown in Figure 3.5(a) since it does not affect system behaviors. If cache data is overwritten by write operation after write or read operations, it is not vulnerable since the soft error induced data can be overwritten. Based on this vulnerability definition, write operations always make periods non-vulnerable from the last behavior to the current write operation as shown in Figure 3.5(b) and Figure 3.5(c) since it overwrites the corrupted data (no impact on system behaviors and propagation) if a soft error occurred. If cache data is simply discarded (e.g., eviction of a non-dirty cache line), it is also not vulnerable as shown in Figure 3.5(d). Since cache data is identical to the data in lower-level memory, it does not update lower-level memory.

On the other hand, cache data is vulnerable if it will be read by a processor since it can affect system behaviors. If cache data is read after data incoming, it is vulnerable since reading corrupted data affects system behaviors as shown in Figure 3.5(e). Read operations always make periods vulnerable from the last behavior to the current read operation as shown in Figure 3.5(f) and Figure 3.5(g) since the corrupted data is read by processor execution, affecting the system behavior, i.e., inducing the high possibility to change the original system behaviors and to result in incorrect outputs or even system crashes. Data in a write-back cache can also be vulnerable if it will be written back into the lower-level memory since it propagates corrupted data to system memory. If cache data is written back at the dirty state, it is defined as vulnerable as shown in Figure 3.5(h).

CVF is the probability that a single-bit error in the cache will result in a system fault or failure. CVF us calculated as vulnerable bytes and their periods, vulnerability in byte $\times$ cycles, over the total cache size and access time as described in (3.1). The denominator of CVF equation is the same for block-level and word-level vulnerability estimation.

The denominator of CVF is the product of cache size in bytes and total execution time in cycles, and it is not different for vulnerability modeling between block-level and word-level. Indeed, the difference between block-level and word-level vulnerability modeling is numerator, vulnerability, as shown in Figure 3.6.

$$CVF = \frac{vulnerability\ (byte \times cycles)}{cache\ size\ (byte) \times total\ execution\ time\ (cycle)} \tag{3.1}$$

3.2.1 Vulnerability estimation at a block-level granularity is inaccurate

We have estimated the vulnerability at a word-level granularity since the basic unit of data access in the cache is a word, not a block, in order to achieve the vulnerability accurately. Figure 3.6 shows differences of vulnerabilities (shaded region) between word-level and block-level estimations under a simple scenario where a block (Block) containing two words (WORD0 & WORD1) is brought at t_0 and evicted at t_4. We consider two cases. The data stored in WORD0 is read at t_1, t_2, and t_3 in Case 1, while they are written in Case 2. A single-bit soft error for the entire scenario is assumed in a write-back cache, and each period (t_i, t_{i+1}) is considered as one cycle for brevity's sake. We also assume that each word contains one-byte data.

It is much more complex to estimate the vulnerability based on word-level vulnerability estimation than block-level one. The access information per word in a block is required and analyzed for word-level modeling, while the only access information per block for block-level modeling is required. Figure 3.6 shows two examples where previous block-level vulnerability modeling cannot provide the accurate modeling as compared to our word-level one. At both cases, one BLOCK contains two words such as WORD0 and WORD1, and read and write operations occur at the word-level while

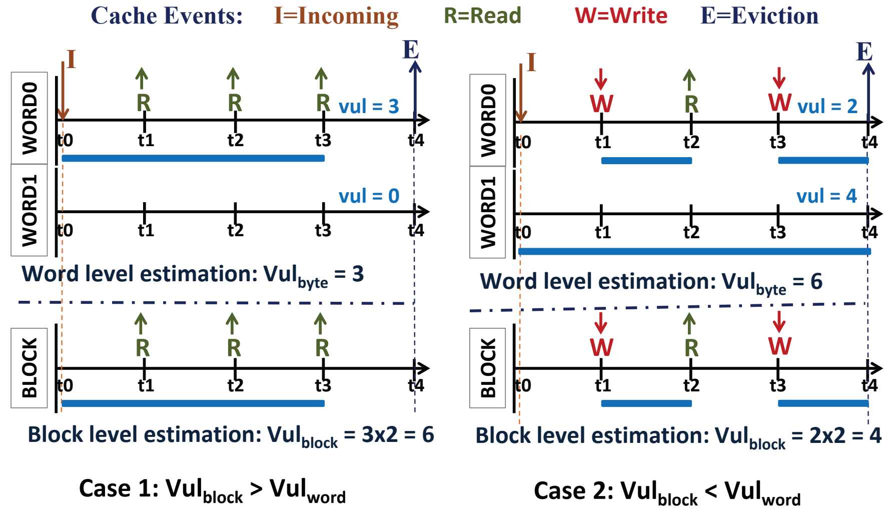


Figure 3.6: Block-level and word-level vulnerability estimation exemplary scenarios without protection techniques

incoming and eviction happens at the block-level. Also, note that each time interval between t_n and t_{n+1} is assumed to one cycle and the word size to one byte for simplicity. Case 1 consists three consecutive read operations of WORD0 at t_1, t_2, and t_3 after the incoming at t_0, and the eviction at t_4. Case 2 consists three consecutive write operations of WORD0 at t_1, t_2, and t_3 after the incoming at t_0, and the eviction at t_4.

In Case 1, the read operations at t_1, t_2, and t_3 make the period from t_0 to t_3 of WORD0 vulnerable according to the vulnerability definition. Note that the read operations of corrupted data can affect the system behaviors, i.e., vulnerable. The interval (t_3, t_4) of WORD0 and (t_0, t_4) of WORD1 are not vulnerable due to the eviction at the clean state at t_4. Note that the cache data is not written back to the lower memory, i.e., no propagation of corrupted data and non-vulnerable, if it is clean under the write-back policy. Thus, our accurate word-level modeling estimates three as the vulnerability of

this BLOCK under Case 1 where vulnerabilities of WORD0 and WORD1 are 3 and 0 byte $\times$ cycles, respectively. However, block-level estimation models this word access behaviors, i.e., read operations of WORD0 at t_1, t_2, and t_3 as block access ones (two bytes of block for three cycles) and it estimates the vulnerability as 6 byte $\times$ cycles (= 3 cycles $\times$ 2 bytes) as shown in Figure 3.6 (left one). Thus, block-level vulnerability overestimates, i.e., Vul_{block} (6 byte $\times$ cycles) is larger than Vul_{word} (3 byte $\times$ cycles) which is correct in Case 1.

In Case 2, the write operations at t_1, t_2, and t_3 make the period from t_0 to t_3 of WORD0 non-vulnerable according to the vulnerability definition. Note that the write operations onto the corrupted data can erase the impact of induced soft errors, i.e., non-vulnerable. However, the eviction at dirty state at t_4 makes (t_3, t_4) of WORD0 and (t_0, t_4) of WORD1 vulnerable. Note that the corrupted data will be propagated to lower-level memory at the eviction if it is dirty under the write-back policy. Thus, our accurate word-level modeling estimates five as the vulnerability of this BLOCK under Case 2 where vulnerabilities of WORD0 and WORD1 are 1 and 4 byte $\times$ cycles, respectively. However, block-level estimation models this word access behaviors, i.e., write operations of WORD0 at t_1, t_2, and t_3 as block access ones (two bytes of a block for three cycles) and they are all non-vulnerable. Thus, it just estimates the vulnerability as 2 byte $\times$ cycles (= 1 cycle $\times$ 2 bytes) as shown in Figure 3.6 (right one). Thus, block-level vulnerability underestimates, i.e., Vul_{block} (2 byte $\times$ cycles) is smaller than Vul_{word} (5 byte $\times$ cycles) which is correct in Case 2.

Figure 3.7 plots L1 data CVF without protection over benchmarks with gemV-cache. In Figure 3.7, X-axis represents benchmarks sorted in the ascending order of CVF and

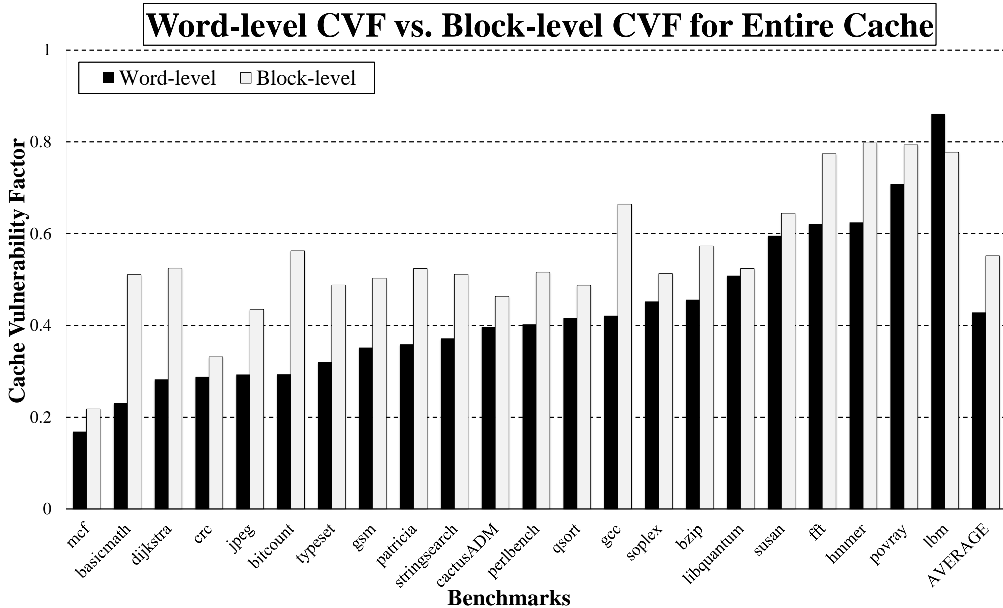


Figure 3.7: Inaccuracy of block-level CVF estimations. Block-level vulnerability estimation is up to 121% inaccurate for the benchmark *basicmath*.

Y-axis represents CVF. The dark bars show the CVF at our word-level granularity, while the light bars for each benchmark show the CVF when the vulnerability is estimated at a block-level granularity. We make several interesting observations from this graph. The difference between CVF estimations at word-level and block-level granularity varies with benchmarks. The maximum inaccuracy is 121% for *basicmath* benchmark, while the average is 37%. Even though the average inaccuracy is not much, the maximum inaccuracy is quite significant. Therefore, only block-level estimation can be significantly erroneous although block-level estimation is easier to understand and implement. More detailed analysis and breakdown of the differences will be in Section 3.2.2.

We also find that the block-level CVF is almost always (except for the *lbm* benchmark) larger than word-level CVF in Figure 3.7. It is because of two important reasons.

The first reason is that a read to a word is considered as a read to the whole block with block-level vulnerability estimation. The accesses to a cache block are not evenly distributed among words in the cache block. CPU will read some specific words more often than other words in common. For example, in *basicmath* benchmark, only 4 bytes out of 64 bytes in a cache block are read by the CPU for about 98% of the whole time. Moreover, 80% of cache operations are read operation. In this case, block-level estimation always updates the vulnerability of the entire block (which is inaccurate) while word-level estimation only updates the vulnerabilities of the specific words (which are accurate). The second reason comes into play when a clean cache block is evicted. For example, about 77% of cache blocks are evicted at the clean state in the *basicmath* benchmark. Block-level estimation calculates that the interval from the last behavior of a block to its eviction is non-vulnerable. However, word-level estimation calculates that each interval from the last behavior of each word to the block's eviction is non-vulnerable. The sum of the non-vulnerable periods of each word in word-level CVF is larger than the non-vulnerable period of the block in block-level CVF, which is why block-level CVF can be more significant than word-level CVF in most cases. However, word-level CVF is larger than block-level CVF for the benchmark *lbm*. The main reason is that data evict at the dirty state rather than the clean state. In the case of eviction at the dirty state, block-level estimation decides the interval between the last behavior of that block and the block's eviction vulnerable. However, accurate word-level estimation decides the interval between the last behavior of each word to the block's eviction in that case. Thus, the sum of vulnerable periods in each word can be more considerable. Indeed, 81% of evictions occur at the dirty state for the *lbm* benchmark, while 34% on average occur at the dirty

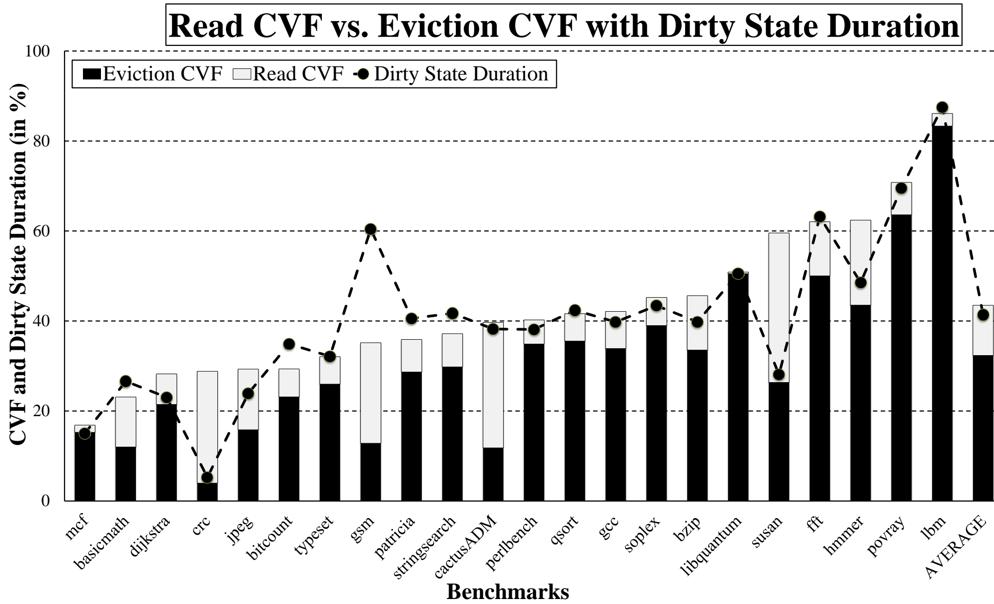


Figure 3.8: CVF based on word-level modeling is proportional to the dirty state in general since vulnerability is mainly comes from eviction at dirty state.

state for other benchmarks.

Another interesting observation from Figure 3.7 is that CVF varies quite significantly among the benchmarks. Benchmark *mcf* has CVF of just 0.17, but for *lbm* benchmark, the CVF is about 0.86. Namely, only 17% of lifetime is vulnerable in the benchmark *mcf*, while almost 90% of lifetime is vulnerable for *lbm*. CVF depends on several factors, including temporal locality of accesses. For instance, if a block is read several times in quick succession and not accessed after that, then it will be less vulnerable, than a block that is read intermittently over the long duration of times. In fact, CVF calculation is quite complex, and therefore we need a detailed algorithm to estimate it. If several factors have strong correlations with CVF, then we can exploit this information to estimate the vulnerability for further purposes, e.g., dynamic protection at run-time.

The dirty state duration of cache blocks is one of the main factors that affect CVF over benchmarks. Dirty state duration is defined as the sum of dirty states time over the execution time in percentage. Thus, dirty state duration of each cache block is the interval between the first write operation to the eviction. When a cache block is dirty, it needs to be written back to the lower level memory at its eviction, which can propagate the errors if they occurred. Thus, the larger portion of dirty state duration in the lifetime of cache blocks can increase the vulnerability as shown in Figure 3.8. In Figure 3.8, benchmarks are in an ascending order of CVF, and the dirty state duration follows this pattern in general. Therefore, cache vulnerability can be reduced if we can minimize the dirty state duration by protection techniques such as write-through or early write-back [27] policies. Some benchmarks, however, such as *crc*, *gsm* and *susan* do not follow this trend. In order to analyze the reason, we classify two CVFs: (i) Read CVF and (ii) Eviction CVF. Read CVF is defined as CVF when reads make the interval vulnerable at either clean or dirty state, and Eviction CVF is when an eviction makes the interval vulnerable at the dirty state.

Note that only reads and evictions at dirty state can make the interval vulnerable without protections. Figure 3.8 depicts Read CVF and Eviction CVF over benchmarks, and Eviction CVF takes up most portions of CVFs in benchmarks that follow the trend of dirty state durations in general. Interestingly, benchmarks *crc*, *gsm*, and *susan* present relatively larger portions of Read CVF such as 86%, 63%, and 56%, respectively in CVFs. In *crc* and *susan*, the dirty state duration is much smaller than CVF, and it means that these benchmarks have large portions of Read CVF at the clean state. Indeed, 96% and 97% of read CVFs occur at clean state in benchmarks *crc* and *susan*, respectively.

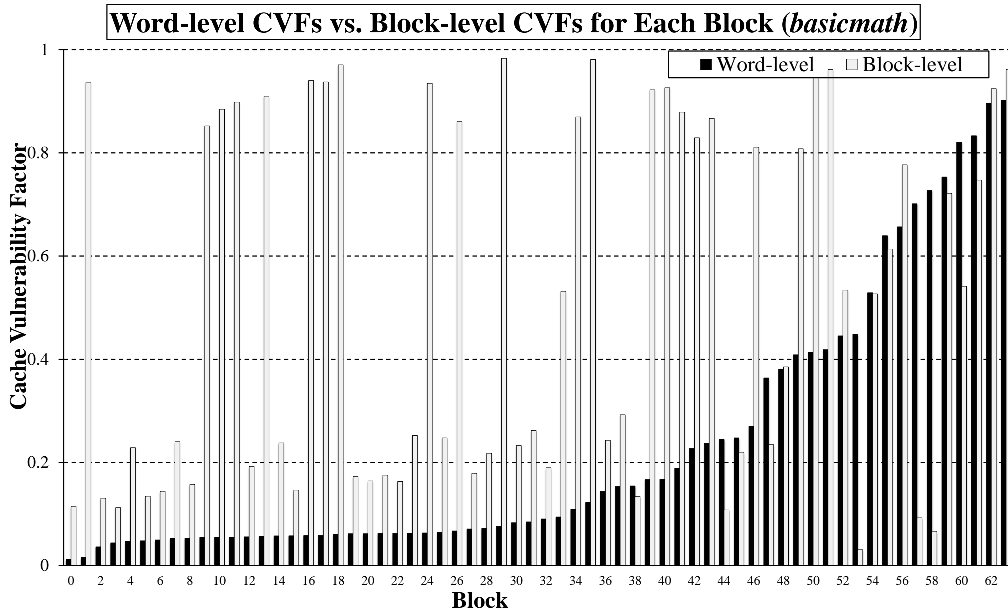


Figure 3.9: Dramatic difference of block-level and word-level CVF for each block. If the vulnerability of a cache block is estimated based block-level modeling, it can be 5,700% inaccurate as compared to accurate word-level one.

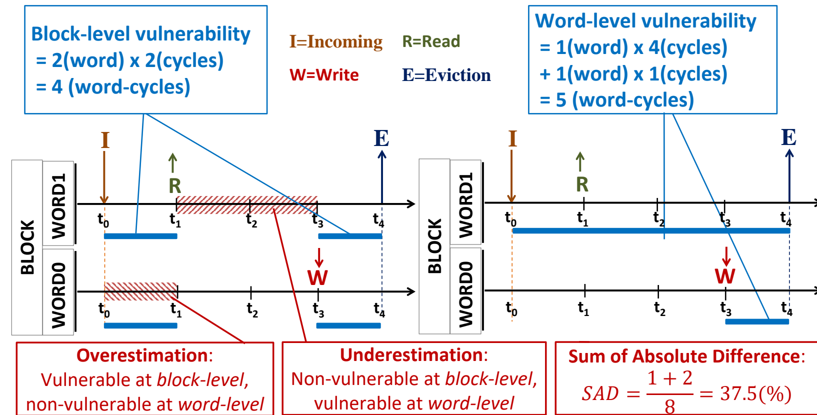
However, Read CVF also takes up many portions of CVF in *gsm*, but the dirty state duration is much larger than CVF in that case. On the contrary to other benchmarks, *gsm* shows that relatively small portions of dirty state durations are vulnerable (e.g., writes at the dirty state often happen to make long time periods non-vulnerable). Indeed, only 55% of dirty state durations are vulnerable in *gsm* while more than 90% of dirty state durations are vulnerable on average for the other benchmarks.

3.2.2 In-depth analysis of inaccurate block-level cache vulnerability estimation

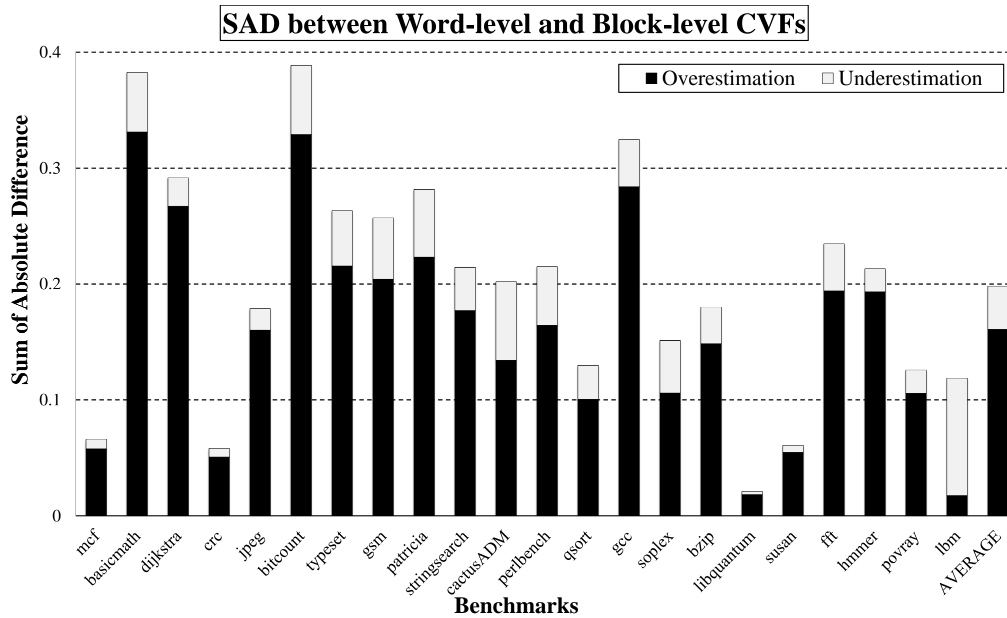
From Figure 3.7, we note that CVF estimation at block-level granularity can be inaccurate by 37% on average and by up to 121%. However, this is just the tip of the iceberg

– this is just the inaccuracy in the aggregate vulnerability statistics over all the data cache blocks. When we consider the vulnerability of a particular block or inaccuracy of vulnerabilities over specific time, the inaccuracy is even more dramatic. Figure 3.9 shows the significant inaccuracy of block-level CVF when CVF of each cache block is evaluated. The light bars show block-level CVF, and the dark bars show word-level CVF for all of 64 blocks in 4 KB direct-mapped cache with a benchmark, *basicmath*. A block (block number 1) shows up to 0.92 difference, and the average difference is about 0.36 which is even higher than the difference of the aggregate CVF (0.28). It is important since researchers have proposed partial protection techniques [30, 29] to protect selected blocks, rather than all the blocks in caches, against soft errors due to high overheads. If they implement block-level estimation to select blocks for partial protections, they will select wrong blocks causing no improvement of resiliency. Assume that the most vulnerable three blocks in data cache are entirely protected by ECC or other protection techniques for the benchmark *basicmath* in Figure 3.9. Block 18, 29 and 35 are selected and protected by block-level estimation since they have highest block-level CVF (0.97, 0.98, and 0.98). However, it only reduces the vulnerability by just 1.8% as compared to the vulnerability without protection since word-level vulnerabilities of block 18, 29, and 35 are just 0.06, 0.08, and 0.09, respectively. Block 61, 62 and 63 are selected for protection based on our word-level estimation since their word-level vulnerabilities are 0.83, 0.90, and 0.90. Thus, the vulnerability can be reduced by 18% as compared to no protection by protecting just three blocks out of 64 blocks.

Figure 3.10 shows the realistic accuracy of our word-level estimation as compared to block-level estimation. In Figure 3.10(a), we have estimated cache vulnerability



(a) Even though word-level and block-level vulnerability is the exactly same, block-level modeling is inaccurate due to overestimation and underestimation.



(b) The relative difference between word-level and block-level CVFs is just 0.28 as shown in Figure 3.7, but sum of absolute difference between them can be reach up to 0.39.

Figure 3.10: SAD (Sum of Absolute Difference) is the sum of overestimation and underestimation of inaccurate block-level estimation as compared to the accurate word-level estimation. SAD can show the realistic inaccuracy of block-level estimation.

based on word-level and block-level modeling under a simple scenario where a block (BLOCK) containing two words (WORD0 & WORD1) is brought at t_0 (incoming) and evicted at t_5. The data in WORD0 is read at t_1 and t_4, it they are written at t_2. The data in WORD1 is written at t_3. A single-bit soft error for the entire scenario is assumed in a write-back cache, and each period (t_i, t_{i+1}) is considered as one cycle for brevity's sake. And, we also assume that each word contains one-byte data. Read operations at t_1 and t_4 make periods (t_0, t_1) and (t_2, t_4) of WORD0 vulnerable according to the vulnerability definition. Note that the read operations of corrupted data can affect the system behaviors, i.e., vulnerable. The interval (t_1, t_2) of WORD0 is not vulnerable due to write operation t_2, and (t_0, t_3) of WORD1 is also not vulnerable due to write operation t_3. Note that the write operations onto the corrupted data can eliminate the impact of induced soft errors, i.e., non-vulnerable. However, the eviction at dirty state at t_5 makes (t_4, t_5) of WORD0 and (t_3, t_5) of WORD1 vulnerable. Note that the corrupted data will be propagated to lower-level memory at the eviction if it is dirty. Thus, our accurate word-level modeling estimates six as the vulnerability of this BLOCK where vulnerabilities of WORD0 and WORD1 are 4 and 2 bytes $\times$ cycles, respectively. On the other hand, block-level estimation models this word access behaviors, i.e., read operations at t_1 and t_4 of WORD0 as block operations, and (t_0, t_1) and (t_3, t_4) are all vulnerable. In the case of write operations at t_2 of WORD0 and at t_3 of WORD1 make the whole block non-vulnerable. Eviction at t_5 makes this block vulnerable because of the dirty state. Thus, block-level modeling also estimates the vulnerability as 6 bytes $\times$ cycles (= 2 bytes $\times$ 3 cycles).

Interestingly, both block-level and word-level modeling results in the same vulner-

ability (6 byte $\times$ cycle) as shown in Figure 3.10(a). However, block-level vulnerability estimation is still inaccurate due to overestimation and underestimation even though aggravated vulnerability is the exactly same. For the interval (t_0, t_1), block-level modeling (2 byte $\times$ cycle) overestimates the vulnerability as compared to word-level one (1 byte $\times$ cycle). On the other hand, block-level modeling (0 byte $\times$ cycle) underestimates the vulnerability as compared to word-level one (1 byte $\times$ cycle) for (t_2, t_3). Thus, we can compute word-level vulnerability by using overestimation and underestimation (vul_{word} = vul_{block} - $overestimation$ + $underestimation$). For instance, word-level vulnerability can be calculated as 6 (= 6 - 1 + 1) in Figure 3.10(a).

Inaccurate block-level estimation can overestimate or underestimate cache vulnerability as shown in Figure 3.10(a), but the entire CVF can just show the relative difference between overestimation and underestimation. The Sum of Absolute Difference (SAD) is the sum of incorrectly estimated CVF between block-level and word-level estimation. In Figure 3.10(a), CVF of block-level and word-level modeling is 0.6, so their relative difference is zero. However, SAD between block-level and word-level vulnerability estimation is 0.2, and it shows the realistic accuracy of block-level modeling. In Figure 3.10(b), X-axis represents set of benchmarks, and Y-axis represents the sum of overestimation and underestimation between block-level and word-level modeling. The upper light one at each bar in Figure 3.10(b) represents the portion that it is vulnerable at word-level estimation even though block-level estimation considers it non-vulnerable (underestimation), and the lower dark one is opposite (overestimation). Hence, the upper one can cause the under-protection, and the bottom one can cause the over-protection if hardware architects implement the protection technique for caches based on block-level

estimation. On an average, the SAD (overestimation + underestimation) is about 0.2 and it can reach up to 0.39 for a benchmark, *bitcount*. Interestingly, relative difference for the benchmark *cactusADM* is just 0.07 as shown in Figure 3.7, but their SAD is 0.20 as shown in Figure 3.10(b).

3.2.3 Validation with fault injections

In order to validate our vulnerability models and the implementation of the vulnerability estimations in gemV-cache, we have performed fault injection experiments on a cycle-accurate simulation infrastructure. Exhaustive fault injection experiments are infeasible. For example, to exhaustively validate the failure rate of a 256 byte direct-mapped cache with 128 bit cache-block, and a benchmark running for 1 million cycles, we will have to perform 128×1 million simulation runs. Since such exhaustive fault injection campaigns for the entire cache is not feasible, we perform exhaustive validation on some randomly selected cache blocks on a few benchmarks from `Livermore Loops` [40] and *Matrix Multiplication*. We have implemented in-house *Matrix Multiplication* benchmark for exhaustive fault injection. We have chosen simple benchmark suites in order to exclude software-level masking effects. We have injected single-bit faults at a specific block during a specific interval and compared its output to the correct one that the benchmark returns without faults. It is declared as a failure if they are different or a system crashes. Otherwise, it is a success. Assuming a single-bit fault model, we have run millions of simulations, and computed Failure RateEquation (3.2):

$$FailureRate = \frac{Num.\ of\ Simulations\ that\ failed}{Total\ Num.\ of\ simulations} \tag{3.2}$$

For validation, the failure rate should match CVF as defined in Equation (3.1).

Table 3.2 compares the failure rate from fault injection and CVF computed from *gemV-cache* for the respective programs. We can see that the *failure rate* and word-level CVF match perfectly; thus validating our vulnerability models and implementation. For the block-level vulnerability estimation, it can be up to 300% inaccurate for block number 8 for the benchmark *Livermore Loops 12*. On average, block-level vulnerability modeling is 52% inaccurate as compared to *failure rate* from fault injection campaigns even for these simple set of benchmarks.

3.2.4 gemV-cache implementation

In order to implement gemV-cache [5], we have developed algorithms that consider every behavior at the word-level in cache blocks and calculate the vulnerability accordingly. *algoECV* describes how to calculate vulnerability of cache without protection (Vul_{np}) and with block-level parity protection (Vul_{bp}). In this algorithm, $curTick$ and $recentTick$ represent the current tick and the tick of the most recent access to a block, respectively. $History$ is a data structure which stores the last behavior such as incoming, read, write, and eviction and the most recent tick. In $history$, we use $uncertain$ tick information to postpone the decision of its vulnerability are recorded. $Accessed$ is another data structure containing all the $History$ information of CPU requested words.

According to each operation such as Incoming, Write, Read, and Eviction, *algoECV* handles these tick values and associated data structures, and estimates the vulnerability of caches as shown in Algorithm 1. In the case of Incoming, it clears the dirty bit of the block (line 05) and saves the current tick to each word's h for every $History$ in the block (lines 07). In the case of Write operation, it sets the dirty bit of the block (line 10) and stores the current tick to h and resets the $uncertain$ (line 13 and 14) if

Algorithm 1: algoECV (Estimate Cache Vulnerability)

algoECV returns vulnerabilities of cache without protections (Vul_{np}) and with block-level parity protection (Vul_{bp}). Based on our algorithm, we can get vulnerabilities with and without protects by just one simulation.

```
01: curtick ←current tick;
02: recentTick ←tick of the most recent access of block;
03: switch Operation do
04:     case INCOMING:
05:         clear dirty bit of the block
06:         for all history h in the block do
07:             h.tick ← curTick
08:         end for
09:     case WRITE:
10:         set dirty bit of the block;
11:         for all history h in the block do
12:             if h ∈ Accessed then
13:                 h.tick ← curTick;
14:                 h.uncertain ← 0;
15:             else
16:                 h.uncertain += curTick − recentTick;
17:             end if
18:         end for
19:     case READ:
20:         for all history h in the block do
21:             if block is dirty then
22:                 Vul_bp += curTick − recentTick;
23:                 if h ∈ Accessed then
24:                     Vul_bp += h.uncertain;
25:                     h.uncertain ← 0;
26:                     Vul_np += curTick − h.tick;
27:                     h.tick ← curTick;
28:                 end if
29:             else if h ∈ Accessed then
30:                 Vul_np += curTick − h.tick;
31:                 h.tick ← curTick;
32:             end if
33:         end for
34:     case EVICTION:
35:         for all history h in the block do
36:             if block is dirty then
37:                 Vul_bp += curTick − recentTick + h.uncertain;
38:                 Vul_np += curTick − h.tick;
39:             end if
40:         end for
41: end switch
42: return Vul_bp, Vul_np;
```

it is accessed. Otherwise, the difference between the current tick and the most recent tick (the uncertain duration for the other words due to the write operation to the word in the same block) is added to the $uncertain$ tick (line 16). A write access to word(s) in the block can affect the other words' vulnerability estimation and this period from the last behavior for the other words needs to be kept $uncertain$ since the next behavior is going to make it vulnerable or not. Indeed, $uncertain$ becomes non-vulnerable if the next behavior to this word is a write operation (line 14) while it becomes vulnerable if it is a read operation (line 24) or eviction (line 37) when this block is dirty. It is why we accumulate these intervals in $history$ and reflect these effects in vulnerability calculation to deal with $uncertain$ periods according to neighbor word's behaviors. In case of Read operation, no protection can merely accumulate the time period from the last behavior to the vulnerability, Vul_{np}, and saves the current tick in $History$ (lines 26, 27, 30, and 31).

To the contrary, the block-level parity protection can recover the corrupted data if it is clean, which means non-vulnerable. However, if it is dirty, the difference between the current tick and the most recent tick needs to be added to the vulnerability, Vul_{bp}, (line 22) and further $uncertain$ ticks need to be added (line 24) if it is in $Accessed$. In case of Eviction, the difference between the current tick and the tick in $History$ is added to the vulnerability for no protection (line 38) while their difference and $uncertain$ in $History$ need to be added to the vulnerability for parity protection (line 37), similar to Read operation. Of course, these ticks are ignored to add up the vulnerability if it is clean. At last, *algoECV* returns Vul_{np} and Vul_{bp} for a block and the sum of all vulnerabilities of all the blocks in a cache is the vulnerability for a program. Similarly, we have developed

algorithms to estimate the vulnerabilities for word-level parity protections by modifying configurations of status bits and cache parameters. For instance, the word-level parity protection returns the same vulnerability as that of no protection except for the impacts of read operations at the clean state (line 30) since the word-level parity-bits clears all the vulnerabilities at reads if the cache is in the clean state.

In gemV-cache, we have implemented vulnerability estimation with various cache configurations such as protections (ECC or parity) and the granularity (dirty bits and parity and ECC bits). Further, all of these different protection schemes can be returned at once, i.e., just one simulation with gemV-cache. Our word-level vulnerability estimation on gemV-cache is much more complicated than the traditional block-level vulnerability modeling due to the following reasons. First off, many more state variables are needed for bookkeeping at the word-level modeling, as opposed to that at the block-level. The access information per word in a cache block needs to be recorded and is analyzed for word-level estimation. For instance, block-level modeling considers each word-access (read and write) as a block-level operation in case of unprotected caches and makes the whole block vulnerable when read. However, word-level modeling makes each word vulnerable or non-vulnerable according to their respective word-level accesses, and thereby estimates cache vulnerability accurately.

Secondly, the behavior of one word affects the perceived vulnerability of the other words in the same block; which becomes a more critical and complex factor when cache protection modeling is involved, at the fine-grained granularity. The vulnerability of one word in a cache-block can depend on the read and write accesses on a neighboring word. For instance, if one word in the cache block is written, then all the words in the same

block become dirty (since they share the same dirty-bit on a block). Even if the other words are not accessed, they become vulnerable. It is because that they will be written back into the memory, and not discarded during the eviction in a write-back cache.

Lastly, the protection granularity makes it more complicated to estimate cache resiliency through word-level vulnerability modeling. If we assume that the parity is encoded at every write operation (no decoding), and it is decoded to check whether an error occurred or not both at every read operation and at eviction. For instance, parity protection can be implemented at the word-level (a parity-bit per word) or block-level (a parity-bit per block), and our algorithm needs to log every word's time. On the other hand, different granularities of protections always return the same vulnerability statistics based on block-level vulnerability modeling.

Table 3.2: Validation of our models and implementation of word-level vulnerability estimation. For all the selected words, we can get the perfectly matched vulnerability as compared to the failure rates through exhaustive fault injection campaigns

Benchmark	Matrix Multiplication		
Block Number	4	9	10
Number of Simulations	1,084,544	107,136	914,048
Failure Rate	97.54	96.35	94.83
Word-level CVF	97.54	96.35	94.83
Block-level CVF	97.51	95.82	87.06
Inaccuracy	0.03	0.56	8.19

Benchmark	Livermore Loop 5		
Block Number	1	11	12
Number of Simulations	44,672	53,504	115,968
Failure Rate	99.64	96.76	3.07
Word-level CVF	99.64	96.76	3.07
Block-level CVF	99.43	94.79	9.69
Inaccuracy	0.22	2.04	215.61

Benchmark	Livermore Loop 8		
Block Number	0	1	3
Number of Simulations	12,895,872	2,977,152	700,672
Failure Rate	46.27	93.84	4.80
Word-level CVF	46.27	93.84	4.80
Block-level CVF	47.72	91.86	8.79
Inaccuracy	3.13	2.10	83.06

Benchmark	Livermore Loop 12		
Block Number	0	8	15
Number of Simulations	452,096	118,912	53,376
Failure Rate	23.28	0.70	97.54
Word-level CVF	23.28	0.70	97.54
Block-level CVF	24.58	2.80	96.40
Inaccuracy	5.56	300.00	1.17

Benchmark	Livermore Loop 18		
Block Number	2	3	5
Number of Simulations	19,280,640	1,448,704	1,534,592
Failure Rate	96.20	55.61	52.15
Word-level CVF	96.20	55.61	52.15
Block-level CVF	92.38	92.07	8.66
Inaccuracy	3.97	65.55	83.40

Chapter 4

Experimental Observations

4.1 gemV for fast and early design space exploration

gemV tool can return vulnerabilities of microarchitectural components for several benchmarks as depicted in Figure 4.1. Since the metric of vulnerability is $bit \times cycle$, the vulnerability tends to be increased for time-consuming benchmarks. In order to compare vulnerabilities in a fair manner, we have estimated architectural vulnerability factor (AVF) as described in Equation (4.1). AVF is the probability which soft errors cause system failures. For instance, 10% of soft errors may cause system failures for a benchmark *stringsearch*. AVF can vary from 7% (for a benchmark *patricia*) to 16% (for a benchmark *qsort*) depending on benchmarks.

$$AVF = \frac{Vulnerability\ (bit\ \times\ cycles)}{Total\ Size\ (bit)\ \times\ Execution\ Time\ (cycles)} \tag{4.1}$$

The value of gemV is in making possible fast DSE or Design Space Exploration at the early design stage. Radiation beam testing requires developers to build an entire working prototype before evaluating the resiliency, and even register-transfer level fault injection requires developers to bring down the design to synthesizable form before

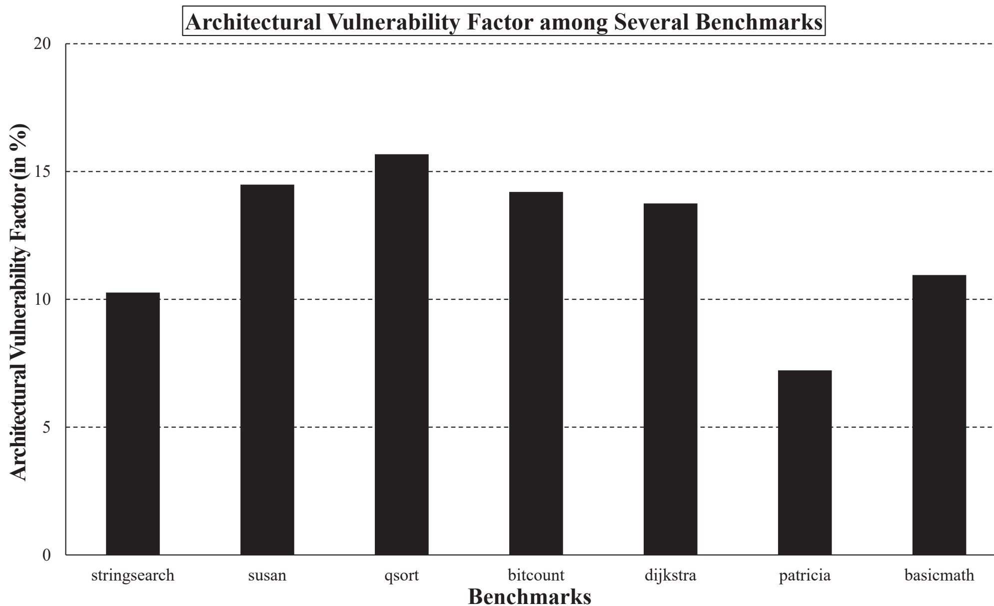


Figure 4.1: Architectural vulnerability factor among several benchmarks. AVF can vary from 7% to 16% by changing benchmarks.

resiliency can be quantified. As opposed to these, gemV allows hardware architects, software engineers, and system designers to evaluate the resiliency at a very early high-level design stage before implementing a physical prototype. In this manuscript, we do not consider cache vulnerability since it takes up the vulnerability too much and it is already explored in Section 4.2. Since cache size much more significant than other components, its vulnerability is much higher than other components.

4.1.1 gemV for hardware implementation

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 the one hand, a broader issue width could reduce the runtime and therefore

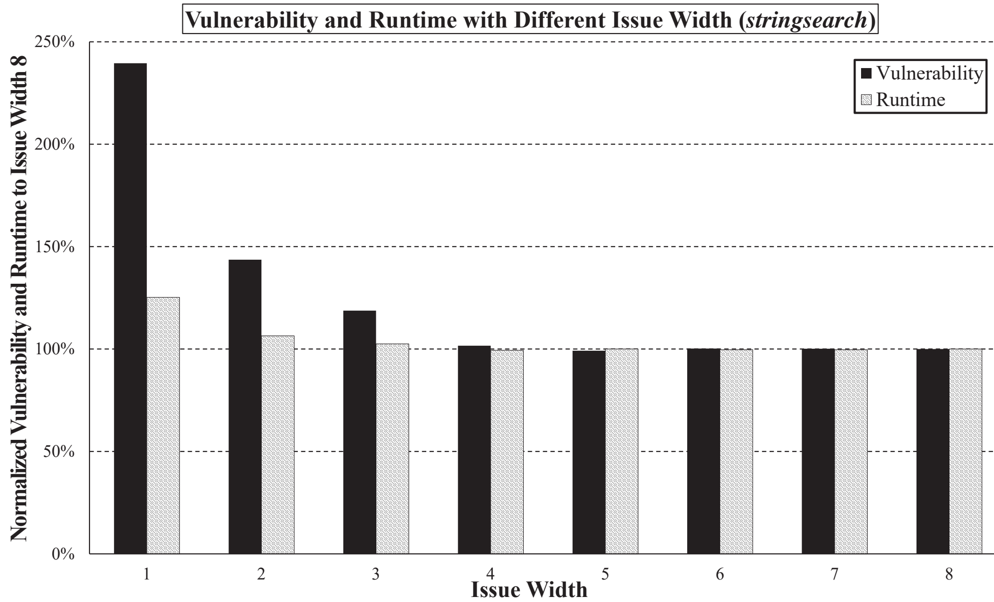


Figure 4.2: Vulnerability and runtime show the same trend by changing issue width, but vulnerability is more sensitive than runtime.

vulnerability. On the other hand, a larger issue width requires more sequential components in the processor, thus increasing the vulnerability. The overall effect on vulnerability is not apparent. With gemV, we can study the effect of such changes and quantitatively answer such difficult questions. For the benchmark *stringsearch*, we observe that vulnerability decreases when increasing issue width from 1 to 3. The vulnerability and the runtime are normalized to those of the basic configuration (issue width = 8) in Figure 4.2. It is interesting that the vulnerability and the runtime show the same trend according to the decreased size of the issue queue. More interestingly, the issue width affects the vulnerability more sensitively than the runtime. For example, the vulnerability can be increased to 240% with the size of the issue queue equal to 1 as compared to the basic configuration while the runtime can be increased to 125%.

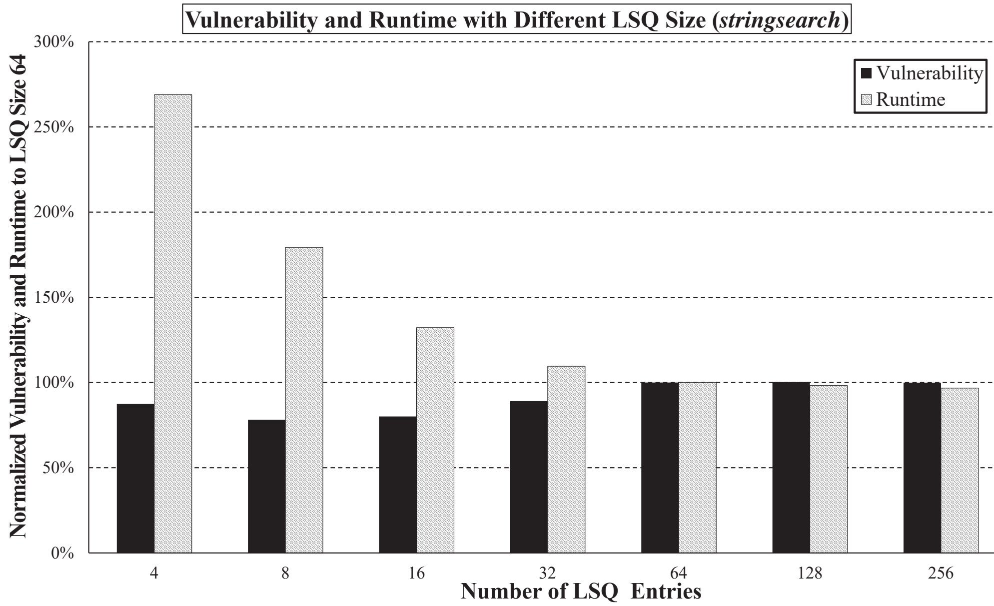


Figure 4.3: LSQ size should be considered with both vulnerability and runtime. Vulnerability is slightly increasing with the increase of LSQ size, while runtime is decreasing.

We have also run the experiments by varying one component with the others fixed for the benchmark, *stringsearch*. Figure 4.3 shows the vulnerability and the runtime normalized to those of the basic configuration (LSQ size = 64) by varying the size of LSQ with all other components fixed for the benchmark, *stringsearch*. When increasing the size of LSQ from 4 to 256, the runtime is improved as shown in Figure 4.3. On the other hand, the vulnerability starts decreasing with the increased size of LSQ but increasing after LSQ size is 16. Thus, gemV is useful to find out an attractive design space considering both the vulnerability and the runtime. Also, we can observe that the LSQ size is more sensitive to the performance than the vulnerability since the runtime ranges from -3% to 170% while the vulnerability ranges from -22% to less than 1% (opposed to issuing width as shown in Figure 4.2). Thus, the designers should be aware

of these sensitivities when selecting the size of microarchitectural components.

Extending this example to a larger design space, one interesting question is, that given an existing processor configuration, and performance leeway, how can we change some hardware configurations to minimize the vulnerability. It can be answered with gemV by plotting design points for runtime against the vulnerability. We fix the number of physical registers in the register file to 256. And, the number of entries in rename map, history buffer, and IEW queue is also fixed to 114, 86, and 8, respectively. We consider 64, 128, 192, 156, 320, and 384 as the number of entry for ROB, 4, 8, 16, 32, 64, 128, and 256 as that for LSQ and IQ, and 1 to 8 by incrementing 1 as that for pipeline queues such as fetch, decode, rename, and I2E ones. In Figure 4.4, the vulnerability and the runtime for configurations are normalized to those of the basic configuration with 192 entries for ROB, 64 entries for LSQ and IQ, and eight entries for all the pipeline queues. Among all of these, we can see four different groups of interesting points as in both positive values of Runtime and Vulnerability, positive values of Runtime and negative ones of Vulnerability, negative values of Runtime and positive ones of Runtime, both negative values of Runtime and Vulnerability. A positive value represents the increased vulnerability (runtime) in the percentage as compared to that of basic configuration while a negative one the reduced vulnerability. For example, (10, -5) implies 10% increased runtime (overhead) and 5% reduced vulnerability as compared to the basic configuration in Figure 4.4.

In this experiment, we randomly have selected the number of entries in ROB, LSQ, IQ, and pipeline queues in order to plot a design space for a benchmark *stringsearch* in MiBench suites [36] as shown in Figure 4.5. A hardware designer can use this de-

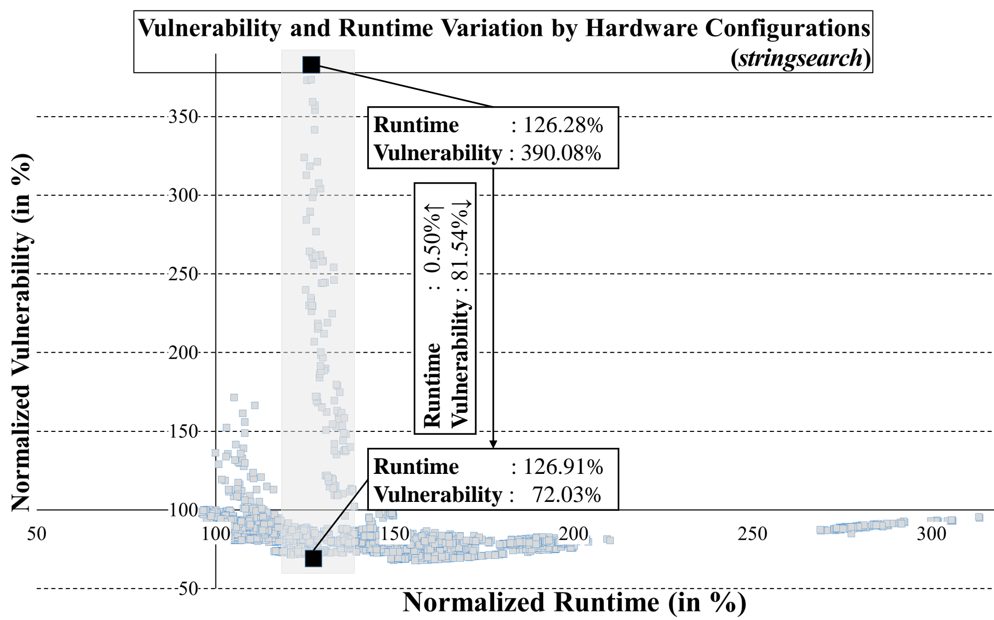


Figure 4.4: Different hardware configurations generates interesting design space in terms of runtime and vulnerability. Vulnerability can be reduced by up to 81% with less than 1% runtime overhead by varying hardware configurations.

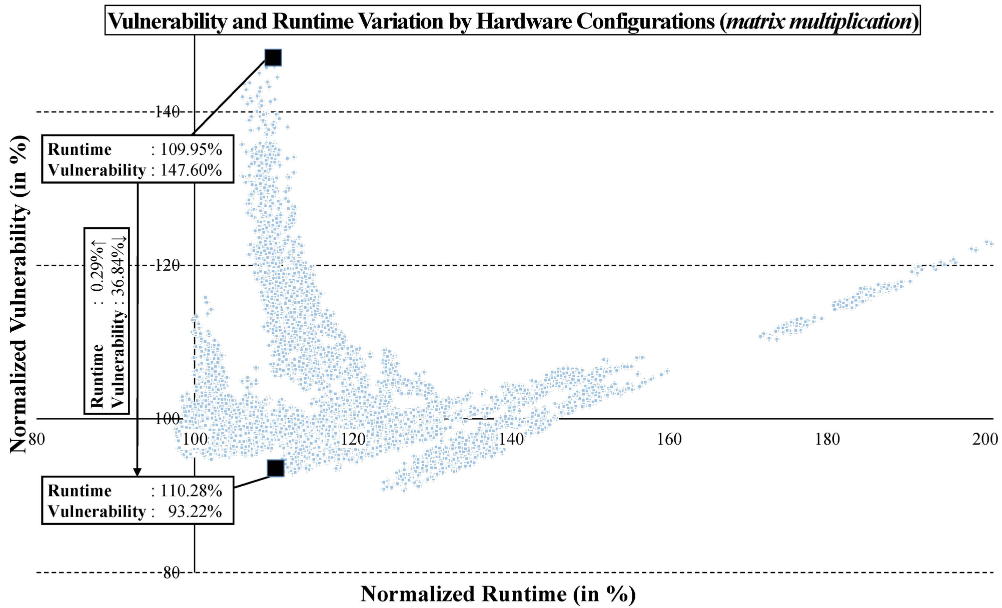


Figure 4.5: Vulnerability and runtime with different hardware configurations (*matrix multiplication*). Given hardware configuration, vulnerability can be reduced by up to 37% with less than 1% performance overhead by changing hardware configuration.

sign space to choose the required hardware configuration as dictated by runtime and vulnerability bounds. Given a specific runtime target, the hardware designer can now find several design points for vulnerability as shown by the gray band in Figure 4.5. It is interesting since we can reduce the vulnerability significantly by just changing configurations without any protections. The hardware architect is interested in the maximum vulnerability reduction within defined runtime bounds. We assume the runtime bound 1% as the tolerable overhead and the vulnerability can be reduced by up to 81% even without severe performance overhead among these design space in Figure 4.4.

We have run the gemV with varying all the possible configurations for a benchmark *matrix multiplication*. Thus, our simulation study runs more than 1.2 million to explore

all possible combinations of these diverse settings. The vulnerability ranges from -9% to 48% where the configuration selection can result in up to 48% vulnerability overhead as compared to the basic one. In this example, for a runtime overhead of 1%, it is possible to find a design point with 37% less vulnerability. Given any runtime or vulnerability overhead, it is now possible to find alternate design points with lower vulnerability or runtime with gemV. Thus, it is the exciting efficacy of gemV which can be very useful for designers to explore the expanded design space by considering both the vulnerability and the performance at the early design stage. Figure 4.5 presents the smaller design space with the larger number of configurations for the benchmark, *matrix multiplication*, than those for the benchmark, *stringsearch*. Indeed, the vulnerability can be reduced by 82% within the 1% runtime constraint for the benchmark *stringsearch*. It is mainly because of the scale of the benchmark, *stringsearch*, is much larger than that of *matrix multiplication*.

Another interesting observation is that each hardware component shows the different effects on the vulnerability reduction. We analyze these experimental results by changing one hardware configuration with all the others fixed. The varying numbers of the entries for ROB does not affect the vulnerability (up to about 14%) while those for LSQ and IQ can influence the vulnerability by up to 44% and 55%, respectively. Note that LSQ also affects the performance significantly by up to 85%. In the case of pipeline queues, they show the similar trends that the vulnerability can be increased by up to around 50% with the varying size of entries for each pipeline queue while the runtime can be increased by 20%. This analysis study can guide hardware designers to select the best configuration with the limited number of total entries, i.e., under the total limited

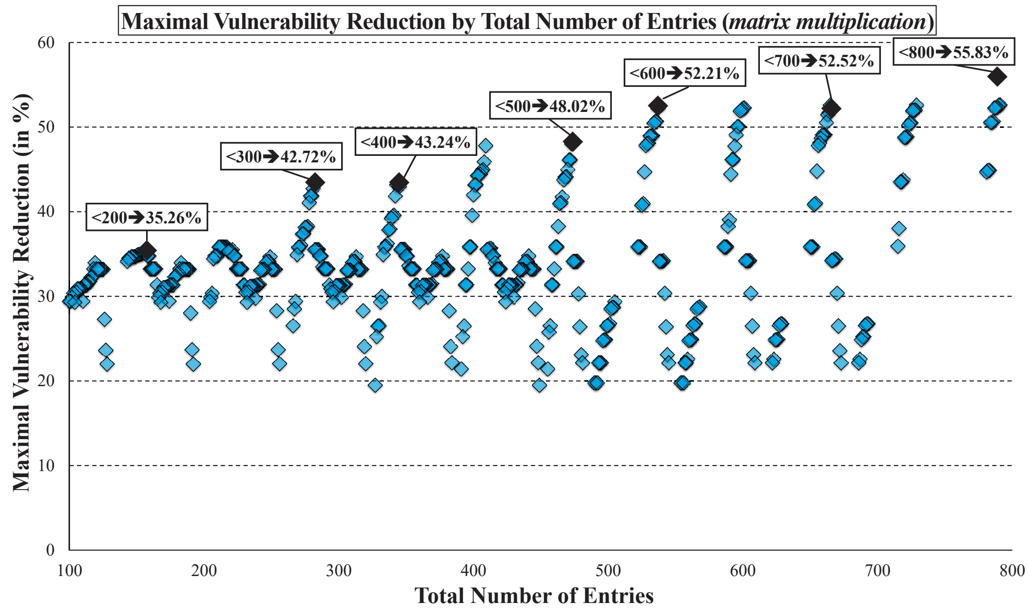


Figure 4.6: Vulnerability can be reduced by up to 56% within the same number of sequential elements.

size of sequential elements in the system. Indeed, Figure 4.6 shows the maximum vulnerability reduction according to the total number of entries in the system. The x-axis represents the total number of entries (the sum of the number of entries for each component), and Y-axis shows the most reduction of the vulnerability in the percentage. For instance, we can reduce the vulnerability by up to 56% as compared to the configuration with the maximum vulnerability under the number of entries, 793. And, vulnerability variation is becoming more abundant with increasing number of the sequential element, so vulnerability is becoming more critical for complex sequential elements.

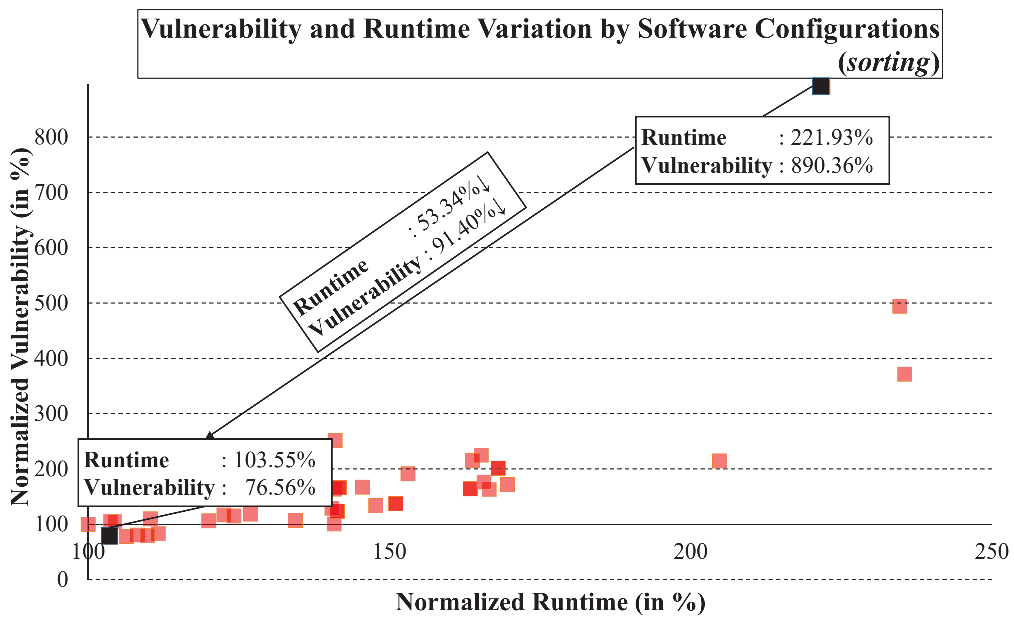


Figure 4.7: Different software configurations can generate interesting design space in terms of vulnerability on the same hardware. Vulnerability can be reduced by 91% without runtime overhead by software changes.

4.1.2 gemV for software development

gemV can also be used by software engineers 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 quicksort 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 experiment by establishing a baseline runtime and vulnerability for an insertion sort algorithm compiled with *gcc* at the highest (O3) level of optimization. Figure 4.7 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 [41]), 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. A 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 quicksort at O3 level of optimization reduces runtime by 53% and vulnerability by 91%.

Table 4.1 summarizes the effects of varying software configurations on the vulnerability and the runtime. In general, the vulnerability is much more sensitive to the software configurations than the runtime. The most potent software option is the sorting algorithm, and the vulnerability can be increased to up to 10 $\times$ with the selection sorting

Table 4.1: Effects of software configuration(algorithm, optimization level, and compiler) on runtime and vulnerability (*sorting*)

		Max (in %)	Min (in %)	Reduction
Algorithm	Runtime	113.95	11.23	10×
	Vulnerability	1005.44	23.44	43×
Optimization	Runtime	101.19	9.69	10×
	Vulnerability	739.46	6.06	123×
Compiler	Runtime	52.33	0.35	173×
	Vulnerability	314.08	5.16	62×

compiled by GCC with an O1 option as compared to the quick sorting compiled with the same compiler and the same option. Note that the maximum difference of the runtime can be up to about 114% for the sorting algorithm selection. Even the least factor on the vulnerability is the compiler, but it can still result in up to 314% variation in terms of the vulnerability while 52% at the most in terms of the runtime. It is also interesting that the option of the compiler optimization can affect the vulnerability by up to 739% while the runtime by up to 101%. This vulnerability-aware design space exploration in software can allow the software designer to meet specific requirements in runtime or vulnerability or both.

4.1.3 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 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. Figure 4.8 shows vulnerability and runtime under different ISAs such as ARM, SPARC, x86, and ALPHA for the *stringsearch* benchmark, with no

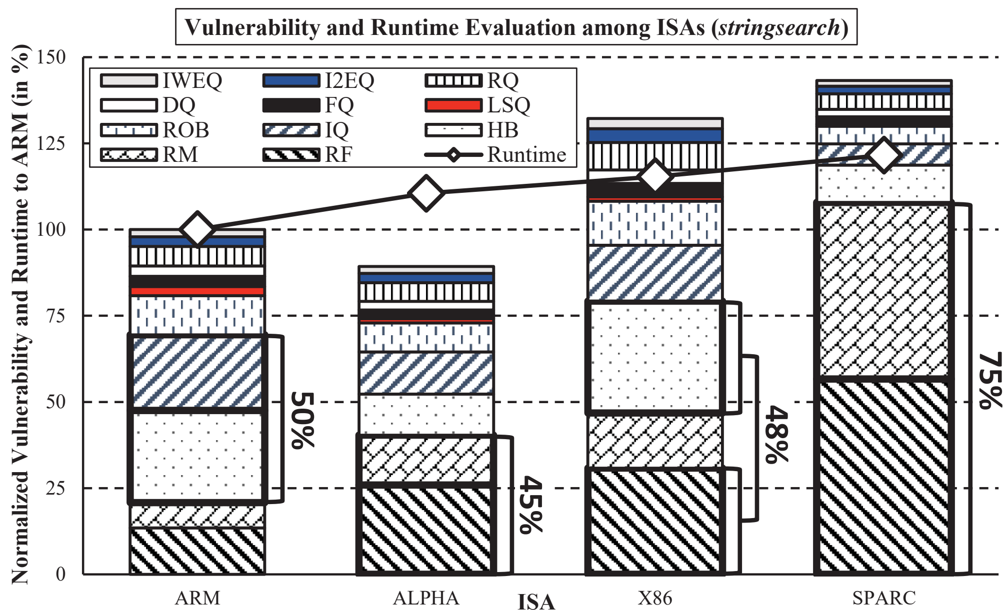


Figure 4.8: Variation in runtime and vulnerability for *stringsearch* under different ISAs. Bars show vulnerability and diamond points indicate runtime

change in hardware and software configurations. Baseline vulnerability and runtime are established on the ARM ISA. A benchmark, *stringsearch*, running on an ALPHA is 38% less vulnerable than an that on 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. Figure 4.8 shows the detailed breakdown of each component such as HB (history buffer), RM (rename map), LSQ, IQ, IEWQ (IEW queue), I2EQ, RQ (rename queue), DQ (decode queue), FQ (fetch queue), 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 the case of SPARC

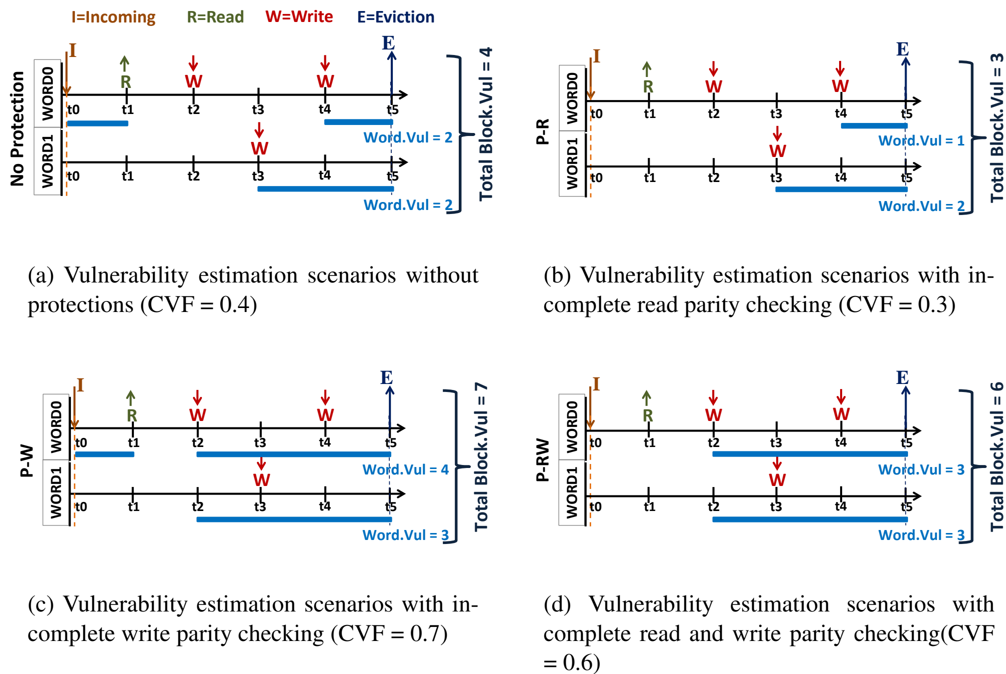
and ALPHA respectively. In this example, a protection mechanism such as ECC can be applied to the register file on the SPARC processor. However, the same protection is not very useful on the ARM processor as the RF contributes only 21% to the system vulnerability.

4.2 Tricky cache protection techniques

Most existing vulnerability estimation toolsets only allow modeling the vulnerability of unprotected caches. However, we need a vulnerability estimation toolset when protections are introduced to cache memory since soft errors are becoming a real threat. gemV-cache can estimate the vulnerability of caches with parity and ECC protections as described in Section 3.2.4. One straightforward and power-efficient cache protection technique, widely implemented in the cache architecture of most existing commodity processors (e.g., ARM [42], Intel [43]), available in the market today, is parity-bit based protection against single-bit errors. And, another effective method to protect the cache memory against soft errors is applying ECC to the entire cache memory. In this manuscript, we model data cache vulnerability of a parity-protected and ECC-protected cache at the word-level granularity in the gemV-cache toolset. And, we also study the impact of the design parameters on the protection achieved.

4.2.1 Incomplete parity checking achieves efficient protection

The critical hardware component involved in the design of parity-bit based protection in the cache is the *Parity-Generator/Checker* - which generates the 1-bit parity for the data stored in a cache block, and also updates the parity-bit when any data update occurs on the respective cache block. The same hardware component will also be used to detect



(a) Vulnerability estimation scenarios without protections (CVF = 0.4)

(b) Vulnerability estimation scenarios with incomplete read parity checking (CVF = 0.3)

(c) Vulnerability estimation scenarios with incomplete write parity checking (CVF = 0.7)

(d) Vulnerability estimation scenarios with complete read and write parity checking(CVF = 0.6)

Figure 4.9: Vulnerability estimation scenarios with diverse parity checking protocols

errors by comparing the current parity-bit value with the stored parity-bit value in the stored data. For instance, i) on only read access (P-R), the parity checker is accessed to verify the parity-bit (PowerQUICC III [44] and ARM1156T2S [45]); ii) on only write access (P-W), the parity checker is accessed to verify the parity-bit, and iii) on both read and write access (P-RW), the parity checker is access to check the parity-bit (ARM Cortex A8 [46] and AM3359 [47]). Note that parity-bit is checked (decoded) before read or write operations.

Figure 4.9 depicts the vulnerability estimation according to the parity checking configurations. We assume that a block which is composed of two words (WORD0 and WORD1) in this scenario for Figure 4.9 has one parity-bit. And, parity-bit can be

checked at read operations (P-R / incomplete read parity checking), at write operation (P-W / incomplete write parity checking), and at both read and write operations (P-RW / complete read and write parity checking). Data is brought at t_0 and evicted t_5 in this block. Data stored in WORD0 is read at t_1 and written at t_2 and t_4, and data stored in WORD1 is written at t_3. Without protections, (t_0, t_1) of WORD0 is vulnerable due to read operation, and (t_4, t_5) of WORD0 and (t_3, t_5) of WORD1 is vulnerable as shown in Figure 4.9(a).

With *complete read and write parity checking*, it has zero vulnerability in case of the clean state as shown in Figure 4.9(d). If errors are detected at clean state, clean data in the lower-level memory can be brought to the cache to correct errors. However, it is always vulnerable after first write operation at t_2 since soft errors can be detected, but there is no same data in the lower-level memory in case of the dirty state. In P-RW, parity check during the first write at t_2 can detect and recover an error while it cannot recover after then (such as at t_3, t_4, and t_5). We can correct soft errors if we have additional recovery mechanisms such as checkpoint and rollback [48]. However, we do not consider the additional protection techniques except parity and ECC in this manuscript. On the other hand, with *incomplete read parity checking*, it has the least vulnerability among these checking configurations as shown in Figure 4.9(b). The periods (t_2, t_4) of WORD0 and (t_2, t_3) of WORD1 are not vulnerable since errors in these periods can be overwritten due to the write operations at t_3 and t_4. Note that parity bit is decoded before read (P-R) and before read and write (P-RW) operations. Interestingly, P-W is more vulnerable than unprotected cache even with the additional redundancy as shown in Figure 4.9(c). In P-W, read at the clean state makes vulnerable periods since it does not check the parity

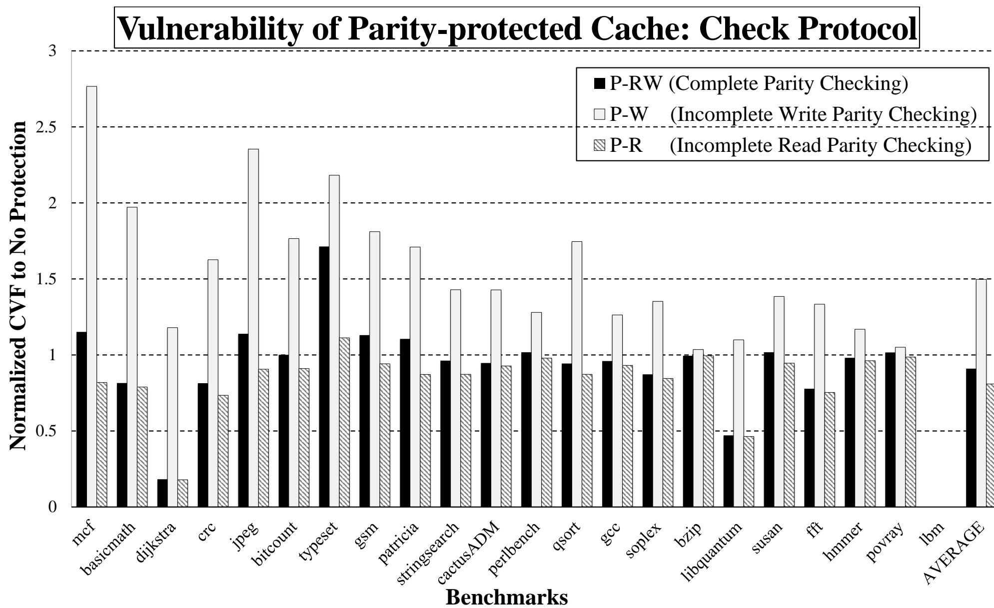


Figure 4.10: Incomplete read parity checking (checking only at reads) achieves the highest resiliency among various parity checking protocols. Complete parity checking is more vulnerable than incomplete read parity checking even with the additional redundancy

bit.

Figure 4.10 clearly shows the efficacy of parity protections with *incomplete read parity checking* with benchmarks. In Figure 4.10, X-axis represents benchmarks, and Y-axis represents the normalized vulnerability of each parity checking configuration to that of no protection. *Complete parity checking* reduces the vulnerability by only 5%, while *incomplete read parity checking* reduces it by 15% on average. However, the vulnerability is worsened by 56% with *incomplete write parity checking* as shown in Figure 4.9(b). It is interesting that *incomplete read parity checking* is the most effective way to reduce the vulnerability among parity checking protocols despite the lesser checking overhead in terms of hardware and power consumption as shown in Figure 4.11. The effectiveness

of parity techniques with *incomplete read parity checking* depends on the characteristics of benchmarks such as cache access patterns. For instance, parity protection with *incomplete read parity checking* for *crc* can decrease the vulnerability by 82% as compared to that of no protection. We have observed that the efficacy of parity protection is affected by some vulnerable periods at the clean state. Note that parity protections cannot bring data from the lower-level memory at the dirty state. In our experiments, more than 80% of vulnerable periods occur at the clean state in the benchmark *crc* in the case of no protection as stated in Section 3.2.2 and these intervals can be effectively corrected by parity protections. For the same reason (a high portion of the clean state), parity is also effective for the benchmark, *susan*. On the other hand, vulnerable periods at the clean state are only 7% in the benchmark *gsm* and thus it is less effective and even up to 71% worse as shown in Figure 4.10.

Figure 4.11 plots the relative power overheads incurred by the parity-checking configurations for 4 KB cache architecture across benchmarks. Y-axis in Figure 4.11 represents the normalized energy consumption to that of unprotected cache. To estimate power consumption, we compute the read/write power of this parity-checking protocol implementation in the cache by manipulating CACTI 5.0 [49] for 45 nm technology node. We have designed the unit for this cache, synthesized it in 45 nm technology, and obtained power numbers using PowerMill [50] in order to estimate the power of the parity generation/checking hardware logic. We can observe that when checking the parity value on both reads and writes (P-RW). Thus P-RW incurs a power overhead of around 103% for only 5% cache protection. On the other hand, an implementation of parity-checking on only reads (P-R), incurs a power overhead of only 71% for around

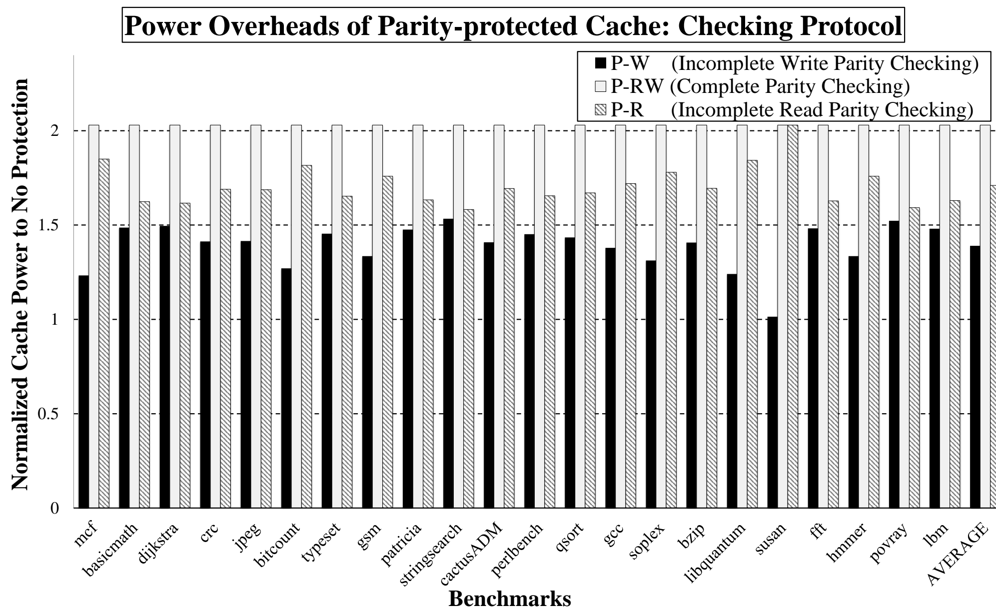
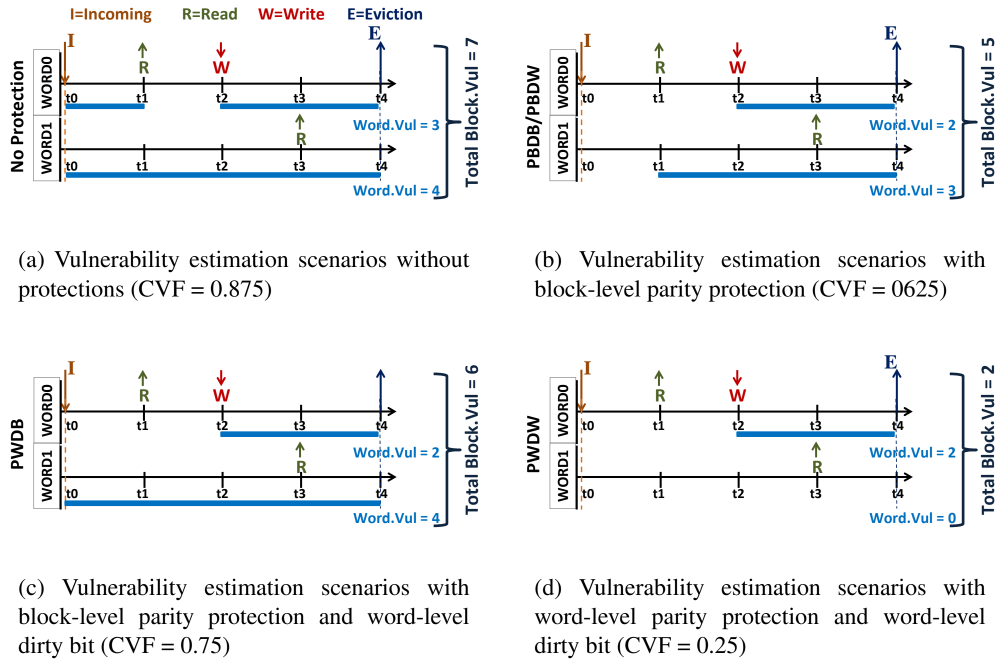


Figure 4.11: In the design of a parity-protected cache, the power overheads caused by parity checking at reads are 30% lower than that when parity is checked on both reads and writes.

15% improved cache protection; achieving power-efficient cache protection. It should be noted that parity is implemented at the block-level granularity. The power overhead comes from P-W is the least (39%) among parity checking protocols, but it increases the vulnerability by 56% on average as compared to no protection.

In short, parity-checking, when performed over read accesses alone, provides the better level of parity protection (avg. 15% and max. 82%) to the cache at 30% lower power overheads compared with parity-checking when performed over both read and write accesses. We have run several simulations varying in cache size and cache associativity and observe that the power-efficient protection achieved through the P-R parity-checking protocol, is consistent across cache configurations.



(a) Vulnerability estimation scenarios without protections (CVF = 0.875)

(b) Vulnerability estimation scenarios with block-level parity protection (CVF = 0625)

(c) Vulnerability estimation scenarios with block-level parity protection and word-level dirty bit (CVF = 0.75)

(d) Vulnerability estimation scenarios with word-level parity protection and word-level dirty bit (CVF = 0.25)

Figure 4.12: Vulnerability estimation examples with diverse status-bit configurations. Note that the granularity of dirty bit does not affect the vulnerability if a parity bit is implemented on block-level

4.2.2 Fine-grained status-bits maximize the achieved parity protection

Another key design parameter involved in the design of a parity-protected cache, is the configuration of the status bits (*parity-bit* and *dirty-bit*), which adds to the hardware overhead. In addition, the vulnerability definition of the parity-protected cache is dependent on the status bit configurations. In a parity-protected cache, a parity-bit can be implemented at the block-level (Itanium 2 [51]) and word-level (PowerQUICC III [44], Cortex R4 [52], and CPPC [28]), and dirty-bit can be also implemented for block-level (Cortex R4 [52]) and word-level (CPPC [28]). Figure 4.12 demonstrates the vulnerability definition of a cache block composed of 2 words (WORD0 and WORD1).

In Figure 4.12, data is brought in the cache at t_0 and evicted at t_4. Data in WORD0 is read at t_1 and written at t_2, and WORD1 data is read at t_3. In case of unprotected cache, (t_0, t_1) of WORD0 and (t_0, t_3) of WORD1 is vulnerable due to read operations, and (t_2, t_4) of WORD0 and (t_3, t_4) of WORD1 is also vulnerable because of eviction at dirty state as shown in Figure 4.12(a).

We have implemented with the P-R (parity-check on reads only) protocol, for the following status bit settings:

PBDB (Parity per Block and Dirty per Block): *Coarse-grained* status bits – Since the entire cache block is configured with one *parity-bit*, a read access on any one word (in non-dirty blocks) can trigger the recovery of the entire cache block; since the single parity-bit cannot identify the exact word that is erroneous as shown in Figure 4.12(b). Also, since the entire cache block is configured with one *dirty-bit*, a write access on any one word makes the entire cache block dirty; thereby rendering every word of the block unrecoverable (vulnerable), on read accesses after that as described in *algoECV*.

PBDW (Parity per Block and Dirty per Word): *Medium-grained* status bits – In this configuration, though each word in the block is configured with its respective dirty-bit, the vulnerability definition does not differ from that of the PBDB configuration as shown in Figure 4.12(b). If any one word of a cache block is dirty (based on its respective dirty-bit), the entire cache block will have to be considered dirty; because the single parity-bit cannot know which word in a cache block has corrupted values.

PWDB (Parity per Word and Dirty per Block): *Medium-grained* status bits – If a parity-

bit is associated with every word in the cache-block, parity-checks on read accesses can identify single-bit errors, and also trigger the targeted recovery of the erroneous word in case of the clean state. Owing to this targeted recovery mechanism, the vulnerability of the nearby words during read accesses are not affected. For instance, we see that the vulnerability of WORD0 and WORD1 are defined by the read/write accesses on the respective words only as shown in Figure 4.12(c). Since the entire cache block is configured with one *dirty-bit*, a write on any one word renders the whole cache block dirty, and therefore an updated word in the cache block cannot be identified. It affects the recovery mechanism and therefore renders the entire cache block vulnerable.

PWDW (Parity per Word and Dirty per Word): *Fine-grained* status bits – If every word in the cache block is associated with its respective dirty-bit and parity-bit, the fine-grained status bit configuration helps achieve increased parity-based protection. Since each word has its respective *parity-bit*, targeted recovery is possible during read accesses in case of the clean state. Also, since each word has its respective *dirty-bit*, the updated words can be identified accurately; thus assisting in the targeted recovery mechanism. In Figure 4.12(d), we see that WORD1 is non-vulnerable from incoming to eviction, because the WORD1 has never been updated by the program, and only the words that have been updated are vulnerable.

Figure 4.13 shows the effectiveness of parity protections by varying the granularity and configuration of the status-bits (such as *parity-bit* and *dirty-bit*) which can induce hardware overhead in a parity-protected cache implementation. In Figure 4.13, X-axis represents benchmarks, and Y-axis represents normalized vulnerability of each status-

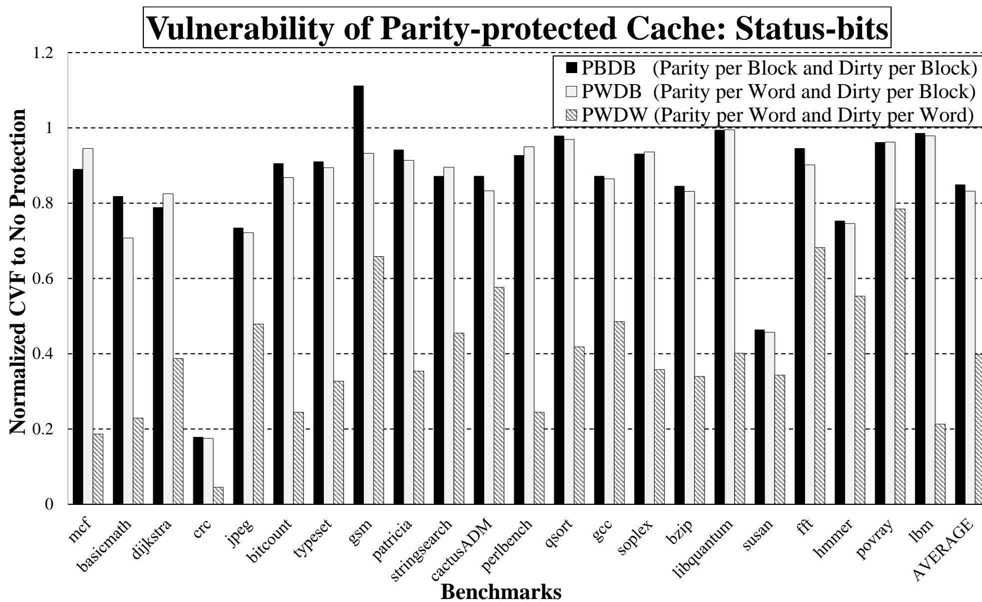
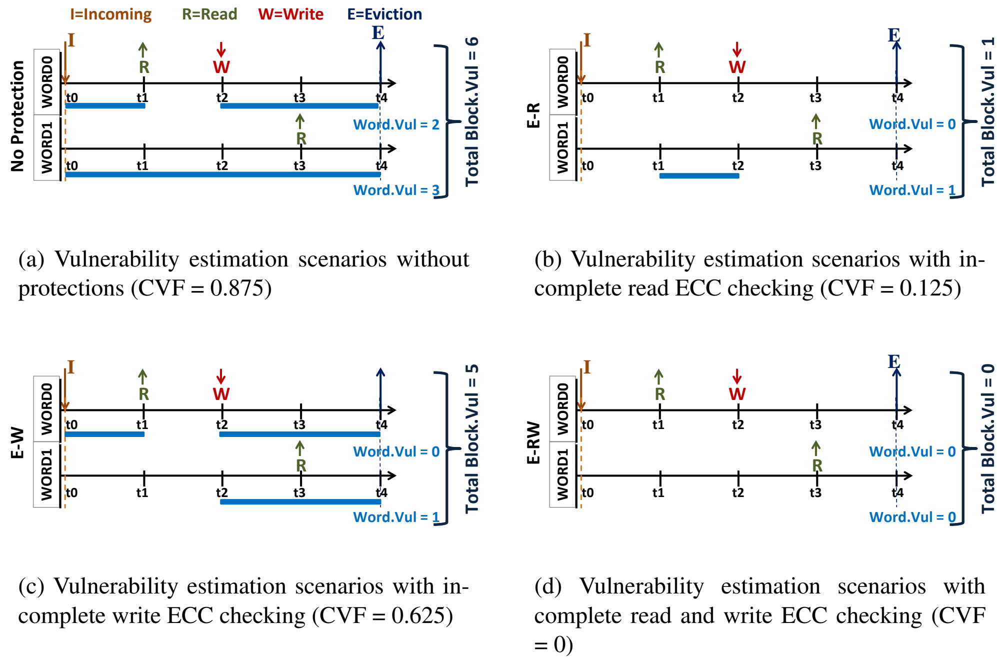


Figure 4.13: Fine-grained parity with block-level dirty-bit reduces the vulnerability by only 2% as compared to block-level parity and dirty-bits. Fine-grained dirty-bit along with parity-bit per word is the best in terms of vulnerability (60% reduction).

bits granularity and configuration by the vulnerability without protections. The coarse-grained PBDB configuration reduces the vulnerability 15% on average. Interestingly, medium-grained PBDW configuration (as in ARM Cortex R4 [52]) reduces only 17% on average even though it needs parity bit per every word in cache blocks. The fine-grained PWDW reduces the vulnerability 60%, and it achieves the maximum level of protection.

In short, it is interesting that the granularity of parity-bits does not affect the vulnerability much without fine-grained dirty-bits. It is mainly because that it cannot locate which word is dirty or clean. Hence, hardware architects have to change the granularity of both parity-bits and dirty-bits in order to reduce the vulnerability efficiently.



(a) Vulnerability estimation scenarios without protections (CVF = 0.875)

(b) Vulnerability estimation scenarios with incomplete read ECC checking (CVF = 0.125)

(c) Vulnerability estimation scenarios with incomplete write ECC checking (CVF = 0.625)

(d) Vulnerability estimation scenarios with complete read and write ECC checking (CVF = 0)

Figure 4.14: Vulnerability estimation examples with diverse status-bit configurations. Note that checking ECC-bits at read operations is more vulnerable than that at both read and write operations.

4.2.3 ECC protection can be vulnerable for single-bit flips

We need to consider two kinds of critical hardware components of ECC based protection in the cache; ECC checking protocol and the granularity of status bits, especially ECC-bits. In ECC protections, dirty bit does not affect the protection efficacy since it can correct soft errors regardless of dirty status. First off, Figure 4.14 depicts the vulnerability estimation according to the ECC checking configurations. In Figure 4.14, we assume there are ECC-bits implemented per a single block, i.e., block-level ECC protection. In Figure 4.14, a block is composed of two words (WORD0 and WORD1), and cache data is brought at t_0 and evicted at t_4. And, cache behavior is the exactly same

with Figure 4.12

ECC checking at both read and write operations (complete read and write ECC checking or E-RW) provides the complete resiliency, which means zero vulnerability. In the case of ECC-protection, it can correct soft errors regardless of dirty status if detected. Since ECC-bits are checked at every behavior in E-RW, it can detect and correct all the single-bit soft errors as shown in Figure 4.14(d). In incomplete write ECC-checking or E-W, read operations always make vulnerable periods as shown in Figure 4.14(c) since it does not check ECC-bits at read operations.

Interestingly, ECC checking at reads (incomplete read ECC checking or E-R) is still vulnerable even for single-bit flips as shown in Figure 4.14(b), while incomplete read checking provides the better vulnerability than complete read and write checking in case of parity protection. ECC checking at read operations can be vulnerable due to the behaviors of other words in the same block. As depicted in Figure 4.14(b), the period (t_1, t_2) of WORD1 is vulnerable since a write operation of WORD0 at t_2 generates new check bits for the whole block. And, the erroneous data could be included if an error occurred (t_1, t_2) at WORD1. And, this erroneous data can be propagated to CPU due to the read operation at t_3. It is interesting that incomplete ECC protection such as E-R and E-W cannot guarantee the perfect fault coverage (zero vulnerability) even for single-bit bit flips according to the ECC checking protocols.

Figure 4.15 shows CVF with complete and incomplete read ECC protections as compared to CVF of no protection. ECC protection may provide the perfect error recovery against soft errors regardless of the cache state. However, it is interesting that incomplete block-level ECC protections do not entirely remove the vulnerability. On average,

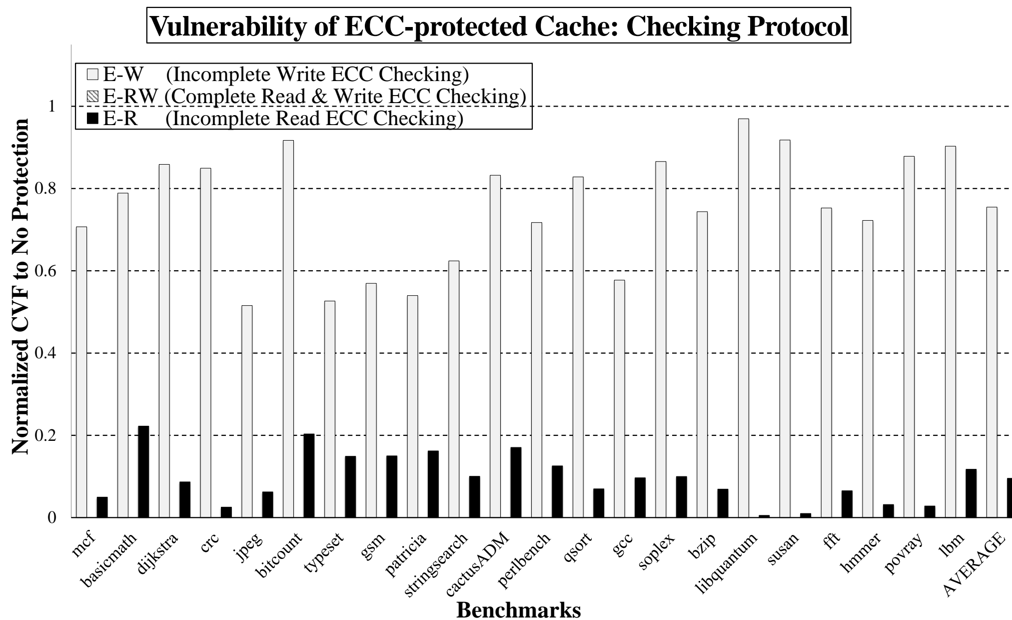
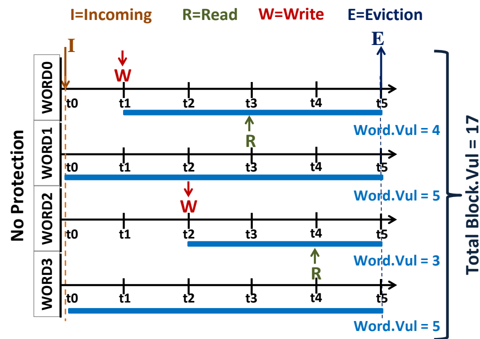


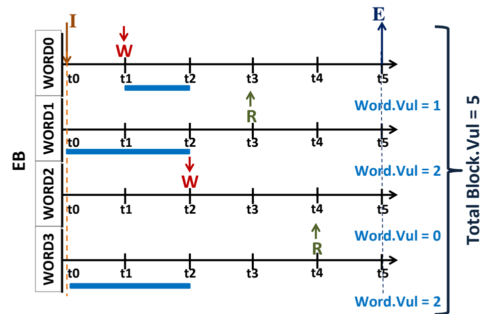
Figure 4.15: Incomplete read ECC checking does not remove the vulnerability completely, while complete ECC checking provides zero vulnerability.

incomplete read ECC and incomplete write ECC reduce the vulnerability by 90% and 25%, respectively, over benchmarks. In the case of ECC protections, the frequency of write accesses affects the effectiveness of vulnerability reductions. For instance, only 1% of total accesses in *susan* are the write operation and its vulnerability can be efficiently reduced to almost zero by the block-level ECC protection with the incomplete read checking protocol.

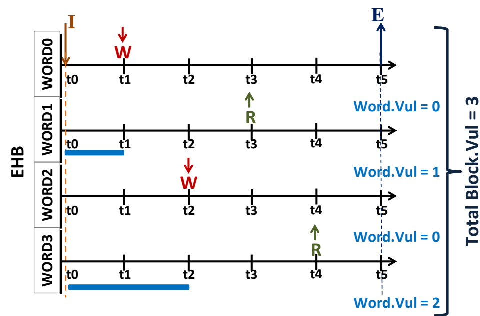
Another key design parameter to the design of ECC-protected cache is the configuration of status bits, especially ECC-bits. Figure 4.16 shows the vulnerability estimation with varying the granularity of ECC-bits under a sample scenario. Note that ECC-bits are checked at only read operations since ECC-checking at both read and write operations makes vulnerability zero. In Figure 4.16, a block is composed four words (WORD0,



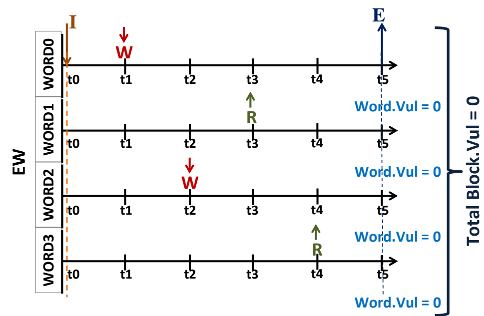
(a) Vulnerability estimation scenarios without protections (CVF = 0.85)



(b) Vulnerability estimation scenarios with block-level ECC protection (CVF = 0.25)



(c) Vulnerability estimation scenarios with half block-level ECC protection (CVF = 0.15)



(d) Vulnerability estimation scenarios with word-level ECC protection (CVF = 0)

Figure 4.16: Vulnerability estimation examples with diverse status-bit configurations on ECC protection. Note that ECC-bits are checked at just read operations.

WORD1, WORD2, and WORD3) in order to separate ECC-bits per half-block and ECC-bits per word. Cache data is brought into this block at t_0 and evicted at t_5. Data stored in WORD0 and WORD2 is written at t_1 and t_3, respectively. Data in WORD1 and WORD3 is read at t_2 and t_4, respectively. Without protections, (t_1, t_5) of WORD0, (t_0, t_5) of WORD1, (t_2, t_5) of WORD2, and (t_0, t_5) WORD3 are vulnerable as shown in Figure 4.16(a).

With block-level ECC protection or EB, write operation of other words in the same block can make vulnerable periods as shown in Figure 4.16(b). For instance, (t_1, t_2) of WORD0 and (t_0, t_2) of WORD1 are vulnerable due to the write operation of WORD2 at t_2. However, (t_1, t_2) of WORD0 and WORD1 are not vulnerable with half block-level ECC protection or EHB as shown in Figure 4.16(c). In EHB, WORD0 and WORD1 are protected by ECC protection, and their vulnerability estimation is not affected by other words such as WORD2 and WORD3. However, (t_0, t_1) of WORD1 is still vulnerable even with half block-level ECC protections due to write operation of WORD0 at t_1. In the case of word-level ECC protection or EW, there are no vulnerable periods as shown in Figure 4.16(d).

Figure 4.17 shows CVF according to the granularity of ECC-bits with the incomplete read ECC checking protocols. As we described before, 10% of the lifetime is still vulnerable by the block-level ECC protection when we check ECC-bits only at reads. CVF can be decreased by 38% by two sets of ECC-bits per block as compared to ECC per a block. We have used 64 bytes as a single block, so we apply ECC-bits per 32 bytes as EHB (ECC-bits half block) implementation for our environments. ECC protection with ECC-bits per word can eliminate the entire vulnerable periods, but it requires

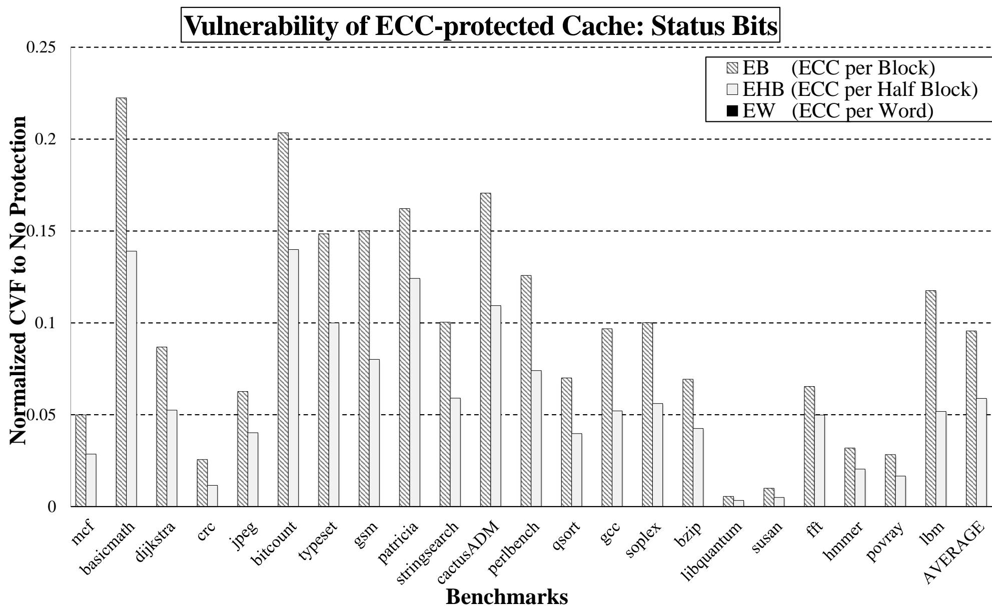


Figure 4.17: CVF with ECC protections are affected by the granularity of ECC-bits.

ECC-bits per each word. Thus, there are two methodologies in order to protect cache memory perfectly by ECC protection; complete read and write ECC checking and fine-grained ECC-bits. The former checks ECC-bits more frequently than the later due to the additional checking at write operations. And, the later can incur more considerable area overhead than the former since the latter requires ECC-bits per each word, not a block. The interAptiv [53] processor has ECC-bits per, and it checks ECC-bits at both read and write operations. However, we do not have to check ECC-bits at both reads and writes if there are ECC-bits per word for the single-bit flips.

In short, we need to be careful in the implementation of parity and ECC protections on caches. We can think that parity should be checked for write and read operations to improve the resiliency of cache memories. It is, however, interesting that parity implementations of additional checks at writes (decoded and encoded as well) can even

increase the vulnerability. Over benchmarks, fine-grained protections (parity-bit and dirty-bit at a word-level granularity) with checks at read operations (incomplete read parity protection) can decrease the vulnerability by about 15% while those with checks at both read and write operations (complete parity protection) can only reduce the vulnerability by about 5%. It is also interesting that block-level ECC checks at both read and write operations (complete ECC protection) can make the vulnerability zero, while ECC checks at read operations only (incomplete read ECC protection) do not bring the vulnerability down to zero.

Chapter 5

Conclusion

Soft errors are becoming a real threat to modern embedded systems. Caches are the most susceptible to soft errors and several protection techniques based on parity and ECC have been presented. However, no existing scheme can accurately estimate how effective these protection techniques in terms of vulnerability. To this end, we propose a protection-aware vulnerability estimation by gemV-cache at the fine-grained word-level modeling. Our experiments with gemV-cache find out several interesting results: (i) block-level modeling and estimation is inaccurate as compared to word-level one, (ii) parity protection is not a good option in case of the early dirtiness, (iii) parity checking at only read operations is the more efficient in terms of vulnerability and power consumption than that at both read and write operations, (iv) the granularity of either only parity-bit or dirty-bit does not affect the vulnerability mainly, (v) introduction of both parity-bit and dirty-bit per word can significantly improve the efficacy of parity protections, and (vi) ECC protection can be vulnerable if block-level ECC bits are checked only at read operations, not at both read and writes.

Several protection techniques against soft errors have been presented ever since resiliency became an important design concern in modern embedded systems. The ne-

cessity to quantitatively study the effectiveness of such protection techniques have led to resiliency quantification schemes such as exhaustive fault injection campaigns and neutron-induced beam testing. Since they are expensive and difficult to perform, several vulnerability estimation tools have been proposed. However, previous vulnerability estimation tools are incomprehensive, inaccurate, and inflexible. In this manuscript, we presented gemV, a comprehensive and accurate vulnerability estimation based on the cycle-accurate simulator gem5. We also showed that our tool had been validated against fault injection experiments into all the microarchitectural components. In order 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-dependent soft error protection techniques.

In the future, gemV will also model and characterize the effects of software level masking effects such as dynamically dead instructions, uninfluential program flow changes, etc. It will improve the accuracy of gemV-tool. Further, we will also validate the accuracy of gemV tool through neutron-induced beam testing. We have validated our vulnerability modeling by statistical fault injection, and we assume that soft error rate of each microarchitectural component is proportional to its size. However, we cannot ensure whether the assumption that we have made is valid or not. Assume that the size of microarchitectural component A and B is 100 and 200. Based on our assumption, the soft

error rate of B is double as compared to that of A. However; it is possible neutron cannot reach to microarchitectural component B through real neutron beam testing. Thus, we need to perform neutron-induced beam testing in order to validate the modeling of gemV-tool.

References

[1] R. Baumann, “Soft errors in advanced computer systems,” *IEEE Design Test of Computers*, vol. 22, pp. 258–266, May 2005.

[2] A. Dixit and A. Wood, “The impact of new technology on soft error rates,” in *IEEE International Reliability Physics Symposium (IRPS)*, pp. 5B.4.1–5B.4.7, April 2011.

[3] J. Yoshida, “Toyota case: Single bit flip that killed,” *Electronic Engineering Times (EE Times)*, vol. 8, 2013.

[4] I. Lee, M. Basoglu, M. Sullivan, D. H. Yoon, L. Kaplan, and M. Erez, “Survey of error and fault detection mechanisms,” *UT Austin, Techical Report*, 2011.

[5] Y. Ko, R. Jeyapaul, Y. Kim, K. Lee, and A. Shrivastava, “Guidelines to design parity protected write-back l1 data cache,” in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 24:1–24:6, ACM, 2015.

[6] S. Jahinuzzaman, B. Gill, V. Ambrose, and N. Seifert, “Correlating low energy neutron ser with broad beam neutron and 200 mev proton ser for 22nm cmos tri-gate devices,” in *IEEE International Reliability Physics Symposium (IRPS)*, pp. 3D.1.1–3D.1.6, 2013.

[7] L. Entrena, 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,” *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 313–322, 2012.

[8] H. T. Nguyen and Y. Yagil, “A systematic approach to ser estimation and solutions,” in *IEEE International Reliability Physics Symposium (IRPS)*, pp. 60–70, March 2003.

[9] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, “Design and evaluation of system-level checks for on-line control flow error detection,” *IEEE Transactions on Parallel and Distributed Systems (TPCS)*, vol. 10, no. 6, pp. 627–641, 1999.

[10] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, “Quantitative analysis of control flow checking mechanisms for soft errors,” in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 13:1–13:6, 2014.

[11] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design,” in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–10, May 2013.

[12] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, “The soft error problem: An architectural perspective,” in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 243–247, Feb 2005.

[13] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-

performance microprocessor," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 29–, 2003.

[14] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Softarch: An architecture-level tool for modeling and analyzing soft errors," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 496–505, June 2005.

[15] X. Fu, T. Li, and J. Fortes, "Sim-SODA: A unified framework for architectural level software reliability analysis," in *Workshop on Modeling, Benchmarking and Simulation (PBMS)*, 2006.

[16] N. Binkert, 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*, vol. 39, no. 2, 2011.

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

[18] S. Mittal and J. S. Vetter, "Reducing soft-error vulnerability of caches using data compression," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 197–202, 2016.

[19] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer*, vol. 38, pp. 43–52, Feb 2005.

[20] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli, "A field analysis of

system-level effects of soft errors occurring in microprocessors used in information systems," in *IEEE International Test Conference (ITC)*, pp. 1–10, Oct 2008.

[21] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski, "Critical charge characterization for soft error rate modeling in 90nm sram," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1879–1882, May 2007.

[22] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's l1 data cache," in *IEEE International Conference on Computer Design (ICCD)*, pp. 499–505, Oct 2006.

[23] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, *et al.*, "Asim: A performance model framework," *IEEE Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[24] M. Moudgill, P. Bose, and J. H. Moreno, "Validation of Turandot, a fast processor model for microarchitecture exploration," in *IEEE International Performance, Computing and Communications Conference (IPCCC)*, pp. 451–457, Feb 1999.

[25] R. Desikan, D. Burger, S. W. Keckler, and T. Austin, "Sim-Alpha: A validated, execution-driven Alpha 21264 simulator," *UT Austin, Technical Report*, 2001.

[26] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 266–277, 2001.

[27] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Soft error and energy consumption interactions: A data cache perspective," in *International*

Symposium on Low Power Electronics and Design (ISLPED), pp. 132–137, Aug 2004.

[28] M. Manoochehri, M. Annavaram, and M. Dubois, "CPPC: Correctable parity protected cache," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 223–234, 2011.

[29] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 411–420, 2006.

[30] R. Jeyapaul and A. Shrivastava, "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors," in *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 105–114, 2011.

[31] W. Zhang, "Computing cache vulnerability to transient errors and its implication," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pp. 427–435, Oct 2005.

[32] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli, "Balancing performance and reliability in the memory hierarchy," in *IEEE Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 269–279, 2005.

[33] R. Jeyapaul, "Systematic methodology for the quantitative analysis of pipeline register reliability in embedded systems," tech. rep., Arizona State University, 2015.

[34] Y. Ko, R. Jeyapaul, Y. Kim, K. Lee, and A. Shrivastava, "Accurate cache vulnerability modeling in presence of protection techniques," in *ESWEEK Workshop on Resiliency in Embedded Electronic Systems*, 2015.

[35] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, Sept. 2006.

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop of the Workload Characterization (WWC)*, pp. 3–14, 2001.

[37] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 502–506, 2009.

[38] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian, "Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach," in *ACM Multimedia*, vol. 8, pp. 319–328, 2008.

[39] N. Wang, M. Fertig, and S. Patel, "Y-branches: When you come to a fork in the road, take it," in *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 56–, 2003.

[40] F. H. McMahon, "The Livermore Fortran Kernels: A computer test of the numerical performance range," tech. rep., Lawrence Livermore National Lab., 1986.

[41] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program

analysis & transformation," in *IEEE International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pp. 75–, IEEE Computer Society, 2004.

[42] R. Phelan, "Addressing soft errors in ARM core-based designs," *White Paper, ARM*, 2003.

[43] M. Demshki and R. Shiveley, "Advanced reliability for intel xeon processor-based servers," in *Intel Corporation*, 2010.

[44] F. Semiconductor, "Error correction and error handling on powerquicc iii processors," 2007.

[45] ARM, "ARM1156T2-S technical reference manual," 2007.

[46] ARM, "Cortex-A8 processor," 2014.

[47] Texas Instruments, "AM3359 sitara processor," 2011.

[48] N. J. Wang and S. J. Patel, "ReStore: Symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.

[49] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," *HP Laboratories, April*, vol. 2, 2008.

[50] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski, "The design and implementation of powermill," in *ACM International Symposium on Low Power Design (ISLPED)*, pp. 105–110, 1995.

[51] C. McNairy and D. Soltis, “Itanium 2 processor microarchitecture,” *IEEE Micro*, vol. 23, no. 2, pp. 44–55, 2003.

[52] ARM, “ARM Cortex-R4 processor,” 2010.

[53] Imagination, “interAptiv multiprocessing system datasheet,” 2012.

국문 초록

내장형 시스템을 위한 신뢰성 측정 방법

고요한
컴퓨터과학과
연세대학교 일반대학원

내장형 시스템을 설계함에 있어서 시스템 디자이너는 성능, 전력, 심지어 신뢰성에 이르기 까지 여러가지 요소를 두루 고려해야만 한다. 내장형 시스템의 경우는 작은 폼팩터로 인해서 배터리 성능에 분명한 제약이 있고, 이로 인해 고성능만큼이나 혹은 그 이상으로 저전력이 중요성 역시 날로 강조되고 있다. 현재는 많은 시스템에서 저전력 컴퓨팅을 위해서 기존의 컴퓨팅 환경보다 훨씬 공격적인 동적 전압 스케일링을 사용하고 있는데, 이는 소프트 에러에 대한 취약성을 높일 수 있다. 소프트 에러란 한 번 에러가 발생하면 하드웨어 자체의 손상으로 인해서 복구되지 않는 영구적 하드 에러와는 달리 에러가 발생하더라도 일시적으로만 바뀌는 에러를 의미한다. 이러한 소프트 에러는 저전력 이슈뿐만 아니라 최근의 하드웨어 소형화와 경량화에 따라 점점 더 심각한 문제로 부각되고 있다. 또한, 이는 단순히 위협 요소 정도에 그치지 않고, SUN 사의 서버 피해와 같은 실질적인 경제적 피해를 일으키고 있는 추세이다. 이에 따라 최신의 임베디드 시스템을 판매하는 회사는 소프트 에러의 피해를 최소화하기 위하여 다양한 보호 방안이 다양한 형태에서 연구하고 있다.

그러나 소프트 에러로부터 시스템을 보호하기 위해서 하드웨어 전체 실행 시간 동안 시스템 전체를 보호하는 방법은 시스템의 성능, 면적, 전력 등에 큰 과부하를 일으킨다. 예를 들어, 해밍 코드처럼 하드웨어를 수정하여 캐시와 같은 메

모리 시스템을 보호하는 경우엔 신뢰성을 높일 수 있는 반면, 면적, 전력, 성능 등에서 지나친 오버헤드를 발생시킨다. 또한, 하나의 태스크를 수행하면서 프로세서의 모든 부분이 소프트 에러에 취약하지는 않은 만큼 실행 시간 내내 전체 시스템을 보호하는 방법은 효율성에서도 많은 의구심을 사고 있다. 이로 인해서 현재는 선택적 보호 방법이 제시되고 있다. 그러나 이러한 선택적 보호 방법이 효율적인지 어떻게 증명할 수 있을까? 보호 방법을 적용할 때 실행 시간이나 전력 소비에 대한 오버헤드는 비교적 계산이나 추정이 쉽게 가능하다. 그러나 소프트 에러에 대한 신뢰성을 수치적으로 측정하는 것은 쉽지 않다.

본고에서는 디자인 공간을 답사하기 위해서 시스템 시뮬레이터 gem5를 이용하여 프로세서 컴포넌트의 신뢰성을 측정하는 gemV-tool 프레임워크를 제안한다. 시스템 시뮬레이터 기반 신뢰성 측정 프레임워크를 통해서 아래와 같은 질문에 답할 수 있다. 먼저, 하드웨어 제작자가 하드웨어 설정을 통해서 신뢰성을 향상시킬 수 있는가? 소프트웨어 엔지니어는 소프트웨어 개발을 통해서 하드웨어의 신뢰성을 향상시킬 수 있는가? 시스템 디자이너는 ISA를 바꿀 수 있는데, 이전 ISA에서 동작하던 신뢰성 향상 방안이 변경된 ISA에서 동작할 것이라고 판단할 수 있을까? 또한, 최근 내장형 시스템은 이미 보호 방법이 적용되어 있는 경우가 많은데, 이를 위한 보호 방법을 적용할 시의 신뢰성 측정 모듈 역시 제공하고 있다.

또한, 본 프레임워크는 다양한 보호 방법을 고려한 신뢰성 측정이 가능하므로 보호 방법 지침을 제공할 수 있다. 본고에서는 캐시 메모리의 중요성을 고려하여 캐시 메모리의 패리티 보호 방안에 대하여 탐색하였다. 먼저, 읽기 연산에 대하여만 체킹을 하는 경우 (즉, 쓰기 연산에 대해서는 하지 않는 경우) 오히려 읽기와 쓰기 두 연산 모두에 대해서 체킹을 하는 경우보다 적은 체킹을 했음에도 높은 신뢰성을 보였다. 더불어, 상태 비트(패리티, 더티)의 입도에 대해서도 검증을 실시하였는데 이 역시 흥미로운 결과를 보였다. 예를 들어, 높은 신뢰성을 위해 패리티 비트를 워드 레벨로 적용할 경우엔 더티 비트 역시 워드 레벨로 적용해야만 한다. 그렇지 않은 경우엔 (즉, 더티 비트는 여전이 블록 레벨일 경우) 하드웨어 오버헤드가 증가했음에도 신뢰성 증가폭이 거의 없었다.

핵심단어: 소프트 에러, 신뢰성, 취약성