electronics

MDPI

Article

gemV-tool: A Comprehensive Soft Error Reliability Estimation Tool for Design Space Exploration

Hwisoo So [1], Yohan Ko [2,*], Jinhyo Jung [1], Kyoungwoo Lee [1] and Aviral Shrivastava [3]

[1] Department of Computer Science, Yonsei University, 50 Yonsei-ro, Seoul 03722, Republic of Korea; shs7719@yonsei.ac.kr (H.S.); jinhyo.jung@yonsei.ac.kr (J.J.); kyoungwoo.lee@yonsei.ac.kr (K.L.)
[2] Division of Software, Yonsei University, 1 Yonseidae-gil, Wonju 26493, Republic of Korea
[3] School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, 660, S Mill Ave, Tempe, AZ 85281, USA; aviral.shrivastava@asu.edu
* Correspondence: yohan.ko@yonsei.ac.kr

Abstract: With aggressive technology scaling, soft errors have become a major threat in modern computing systems. Several techniques have been proposed in the literature and implemented in actual devices as countermeasures to this problem. However, their effectiveness in ensuring error-free computing cannot be ascertained without an accurate reliability estimation methodology. This can be achieved by using the *vulnerability* metric: the probability of system failure as a function of the time the program data are exposed to transient faults. In this work, we present a gemV-tool, a comprehensive toolset for estimating system *vulnerability*, based on the cycle-accurate gem5 simulator. The three main characteristics of the gemV-tool are: (i) *fine-grained modeling:* vulnerability modeling at a fine-grained granularity through the use of RTL abstraction; (ii) *accurate modeling:* accurate vulnerability calculation of speculatively executed instructions; and (iii) *comprehensive modeling:* vulnerability estimation of all the sequential elements in the out-of-order processor core. We validated our vulnerability models through extensive fault injection campaigns with $<3\%$ correlation error and 90% statistical confidence. Using the gemV-tool, we made the following observations: (i) the vulnerability of two microarchitectural configurations with similar performance can differ by 82%; (ii) the vulnerability of a processor can vary by more than $10\times$, depending on the implemented algorithm; and (iii) the vulnerability of each component in the processor varies significantly, depending on the ISA of the processor.

Keywords: soft error; transient fault; fault tolerance; embedded systems; protection technique

check for updates

Citation: So, H.; Ko, Y.; Jung, J.; Lee, K.; Shrivastava, A. gemV-tool: A Comprehensive Soft Error Reliability Estimation Tool for Design Space Exploration. *Electronics* **2023**, *12*, 4573. https://doi.org/10.3390/electronics12224573

Academic Editor: Bahman Javadi

Received: 16 September 2023
Revised: 25 October 2023
Accepted: 29 October 2023
Published: 8 November 2023

1. Introduction

A soft error is a change of value in a storage cell or on a circuit line induced by external sources. This commonly occurs when energy-carrying particles, such as alpha particles, protons, low-energy neutrons, and cosmic rays, collide with the chip [1,2]. When the charge caused by the collision exceeds the critical charge, which is proportional to the chip size and supply voltage, a soft error can occur. Therefore, soft error rates are inversely proportional to the critical charge, and they are increasing exponentially due to aggressive and continuous technology scaling. Even though soft errors are only temporary defects, they are no less dangerous than permanent hardware malfunctions. For instance, soft errors in embedded systems used for satellites [3], automotives [4], or healthcare systems can be critical to human life.

Many techniques have been presented in various design layers to protect computing systems against soft errors [5]. These protection methods have severe overhead in terms of area, performance, and energy consumption. Despite these disadvantages, protection schemes are neither always useful nor continuously robust against soft errors, and sometimes fail to protect systems [6]. Thus, protection techniques should be carefully chosen by considering the trade-off between reliability and performance. While performance can be estimated or quantified using runtime or the number of instructions executed per cycle, no widespread methods exist that estimate reliability in both an accurate and timely manner.

Traditionally, neutron beam testing [7,8] and fault injection campaigns [9] have been used to estimate the reliability of a system against soft errors. Neutron beam testing uses a cyclotron to expose computing systems to neutron-induced soft errors. These experiments are very costly, and the environments that support the experiments are severely limited. Even though [8] can estimate the vulnerability against soft errors at the early stage using bare-metal applications, they still suffer from expensive facility setups for neutron beam testing. Fault injection campaigns intentionally inject errors into specific bits in a processor at specific cycles during the runtime. However, exhaustive fault injection campaigns need to inject faults into all bits of the entire computing system at every cycle of the execution time, which is almost impossible [10]. Statistical fault injection campaigns based on probability theory have been presented to minimize the number of experiments, but the number of injections required to obtain meaningful results is still large [11]. In addition, fault injection campaigns and beam testing are not only costly and challenging to set up but are also often flawed [12,13].

The *vulnerability* metric [14,15] has been presented as an alternative to slow, expensive neutron beam testing and fault injection. This metric is measured in $bit \times cycles$, and it calculates the number of bits that are vulnerable, or may incur system failures in the presence of an error, and the duration of time the bits are vulnerable. For example, assume that bit b in a microarchitectural component is written at time t, and is read by the CPU at time $t + n$. In this simple scenario, bit b is not vulnerable before time t because even if its data before the write operation are corrupted, they would be overwritten to the correct value. On the other hand, bit b is vulnerable during the time interval between t and $t + n$, because errors in this interval cause the corrupted data to be read by the CPU. The vulnerability of the entire processor is the summation of these intervals for all the microarchitectural components. Unlike fault injection campaigns, which require a large number of trials, vulnerability estimation can be performed in a single simulation by tracing the read/write behaviors in each component. Thus, vulnerability modeling is an effective and quick way to explore the design space in terms of reliability and performance [16,17] for hardware configuration, software engineering, and system design.

Several vulnerability estimation modeling techniques based on cycle-accurate simulators have been presented [15,18,19]. However, these techniques are inaccurate, incomprehensive, and inflexible. First, the accuracies of the previous vulnerability estimation schemes are limited because they estimate the vulnerability at a coarse-grained granularity. In addition, their modeling techniques ignore the vulnerability of speculatively executed instructions (i.e., squashed instructions due to misspeculation), even though their presence in the pipeline can, in some cases, cause failures. Note also that their accuracies have not been comprehensively validated or published. Second, existing vulnerability modeling techniques are not comprehensive as they model only a subset of the microarchitectural components in a processor. Finally, the techniques provide only a specific configuration of vulnerability estimation, which is limited to the ISA and core of the underlying simulator.

In this work, we present gemV-tool [20], a toolset for comprehensive vulnerability estimation based on gem5 [21] (a popular cycle-accurate simulator) [22]. Unlike other simulators, gem5 explicitly models all the microarchitectural components of an out-of-order processor, various ISAs (ARM, ALPHA, SPARC, etc.), and even many system calls. Some of the critical features of the gemV-tool that enable accurate vulnerability estimation are as follows: (i) fine-grained modeling of hardware components through the use of RTL abstraction inside the gem5 simulator, (ii) accurate modeling of the vulnerability of both committed and squashed instructions, and (iii) comprehensive vulnerability modeling for all the microarchitectural components of out-of-order processors. We also performed extensive fault injection campaigns and validated that the gemV-tool has 97% accuracy with a 90% confidence level.

2. Related Works

2.1. Vulnerability Estimation from Different Layers

Seifert et al. [23] presented the timing vulnerability factor (TVF) for a circuit environment in which sequential elements are typically placed. All particle strikes do not necessarily propagate to soft errors in the architecture due to electrical, logical, and latch-window masking effects. For example, if the latch accepts data only during half of the lifetime, the corrupted data at the circuit level will be masked by approximately 50%. Such circuit-level vulnerability factors are related to the raw device fault rate and, thus, it is difficult to consider the characteristics of each microarchitectural component. TVF also ignores the architectural masking effects because it is unaware of the behaviors of the microarchitectural components. In conclusion, the TVF is too conservative to express the vulnerability of each microarchitectural component.

Sridharan et al. [24] analyzed and proposed a software-level vulnerability evaluation method called the program vulnerability factor (PVF). PVF estimates the vulnerability of software resources, such as architectural registers based on the given assembly code. Compared to conventional vulnerability estimation metrics, the PVF can be calculated quickly, and it functions as a decent predictor for estimating the vulnerability of hardware components. However, PVF estimation does not use clock cycles: instead, it uses the order of instructions as a single-time quantum. This undermines the accuracy of PVF vulnerability estimation.

The instruction vulnerability factor (IVF) [25] evaluates how many faults in the instructions affect the final program output. IVF experiments inject faults into static instructions to estimate the vulnerability of the software to soft errors. However, it is challenging to mimic the effects of soft errors by modifying static instructions because soft errors occur at the architectural level rather than at the instruction level. Furthermore, both PVF and IVF are unaware of the hardware architectures because they are only based on software or assembly-level analysis.

Mukherjee et al. [15] presented the AVF (architectural vulnerability factor), which calculates the probability that a state change (soft error or transient bit flip) in the device leads to an architecturally visible error. It traces architecturally correct execution (ACE) bits or bits that induce system failures if they change, and the time ACE bits reside in microarchitectural components. Thus, AVF estimation is faster than the register–transfer level simulation [26] and more accurately reflects the effects of soft errors that occur at the microarchitectural level. This study also leverages the AVF concept to accurately estimate each microarchitectural component's reliability.

2.2. Limitations of Previous Vulnerability Estimation Schemes

To estimate the vulnerability of a system, previous works exploited cycle-accurate, system-level, and software-based simulators, as described in Table 1. Mukherjee et al. [15] proposed the architectural vulnerability factor (AVF) based on Asim [27], which simulates Itanium 2-like IA64 processors. Li et al. [18] proposed SoftArch, which models the error generation and propagation based on the probabilistic theory in the Turandot simulator [28]. Sim-SODA [19] was presented to estimate the vulnerability of microarchitectures based on the Sim-Alpha simulator [29]. However, these methods lack accuracy, comprehensiveness, and availability.

First, the existing techniques estimate vulnerability at a coarse-grained granularity. In Sim-SODA, several hardware structures in instruction fetch and issue logic are modeled as a combined single hardware structure called "instruction window". Hardware structures, such as pipeline queues, instruction queues, and load/store queues, are not modeled individually, and their vulnerabilities cannot be evaluated. Problems still exist even if the hardware structures are modeled individually. In AVF and SoftArch, microarchitectural components, such as the instruction queue, are modeled as bulk structures. However, not all bits of a component are vulnerable or non-vulnerable at the same time. For instance, the predicted next PC address is not vulnerable because it cannot affect the program output;

however, the current PC address is vulnerable because it can cause incorrect program flow. These coarse-grained calculations may underestimate or overestimate the vulnerability of the processor.

Table 1. Comparison between vulnerability estimation tools.

Tool	Accuracy	Comprehensiveness	Extensibility	Validation
AVF [15]	Inaccurate: 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 [27] simulator	No published results
SoftArch [18]	Inaccurate: 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 architecture based on proprietary Turandot [28] simulator	No published results
Sim-SODA [19]	Inaccurate: 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 the load store queue are modeled for vulnerability estimation	ALPHA architecture based on open-source Sim-Alpha [29] simulator	No published results
gemV-tool (Proposed tool)	Every structure is modeled based on fields that are actually used (Section 3.1) Squashed instructions are also considered for vulnerability modeling (Section 3.2)	Register file, instruction queue, reorder buffer, load store queue, pipeline queues, and renaming units are modeled for vulnerability estimation (Section 3.3)	ARM, ALPHA, POWER, X86, SPARC architectures with various configurations based on open-source gem5 [21] simulator (Section 3.4)	Validated through extensive fault injection (Section 4)

Second, previous works ignored squashed instructions in their vulnerability estimation. In an out-of-order processor, instructions can be squashed because of misspeculation. Because these instructions are not executed, it can be assumed that all bits in the instruction are non-vulnerable, but this is not the case. For instance, the rename map holds the index mapping between architectural and physical registers and uses a history buffer to maintain the previous mapping of an architectural register. When instructions are squashed due to branch misprediction, the processor state must be rolled back to the last committed instruction. Then, the previous mapping in the history buffer is vulnerable because the processor may roll back to the wrong register if the mapping is corrupted. However, previous vulnerability estimation tools consider all squashed instructions to be non-vulnerable and miss out on these vulnerabilities.

Third, previous tools do not provide comprehensive vulnerability modeling because they consider only a subset of the microarchitectural components rather than all the components in a processor. In [15,18], the authors did not model register files, memory hierarchy, or pipeline structures in their vulnerability estimation. Sim-SODA includes more microarchitectural components, but it still does not consider pipeline queues and renaming units in its vulnerability estimation.

Lastly, previous tools are inflexible, because of the limitations of the simulators they use. Vulnerability estimation techniques in [15,18] use proprietary and private simulators that model Intel's Itanium 2-processor and IBM's POWER ISAs, respectively. Sim-SODA estimates the vulnerability on a publicly available Sim-Alpha simulator. These techniques can only estimate the vulnerability of their designated processors. Moreover, the accuracy of their vulnerability estimation suffers due to the inaccuracies of the simulators.

Ref. [30] shows that the runtime of Sim-Alpha can reach up to 43% compared to real Alpha architectures.

3. gemV-tool: Fine-Grained and Comprehensive Vulnerability Estimation

The main goal of the gemV-tool is to quickly and accurately estimate the vulnerability of the system to soft errors based on microarchitectural behaviors. For this purpose, the gemV-tool relies on the gem5 simulator [21], which provides cycle-accurate microarchitecture-level simulations with several hardware configurations. Figure 1 illustrates a technical overview of the gemV-tool with its input and output. Given the hardware and software configurations, the gemV-tool traces the read/write behaviors of each hardware component based on the gem5 simulation. After the simulation, the gemV-tool produces quantitative component-wise vulnerabilities with runtime results. In summary, the gemV-tool can provide hardware and software design space exploration in terms of performance and reliability.

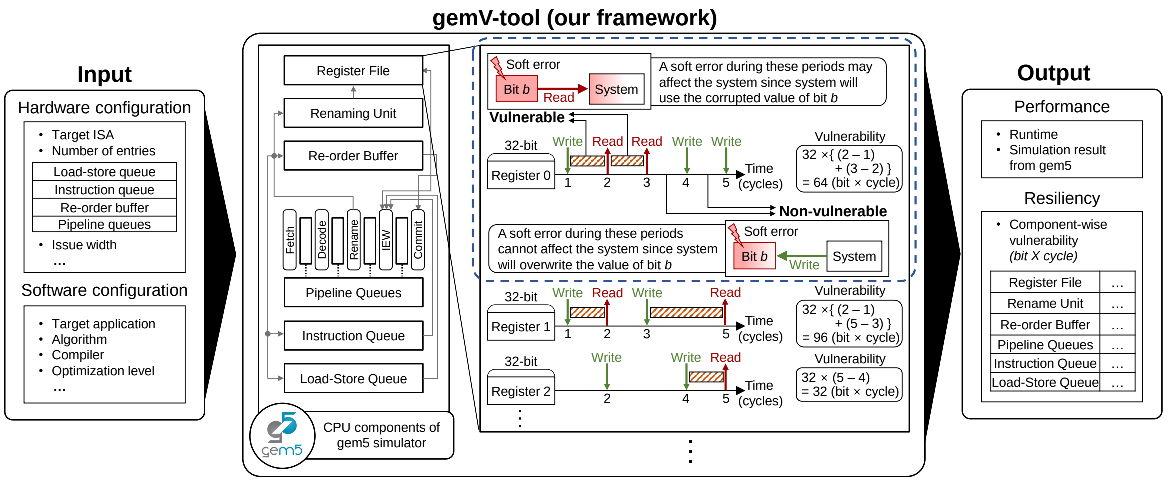

Figure 1. Comprehensive vulnerability estimation modeling on the gem5 simulator. (Green: Write, Red: Read).

To estimate the vulnerabilities of hardware components, the gemV-tool uses the architectural vulnerability factor (AVF) concept [15]. Vulnerability is a quantitative metric that represents the reliability of microarchitectural components and is measured in a *bit* × *cycles* to consider both the temporal and spatial domains. If a soft error occurs in a particular bit *b* during time *t* and results in a system failure, the specific bit *b* is considered vulnerable at the particular time *t*. Otherwise, bit *b* is considered non-vulnerable. Based on this classification, the gemV-tool defines the system vulnerability as the sum of these vulnerable bits in the microarchitectural components of a processor. For example, if two bits in a microarchitectural component are vulnerable for five cycles each, the vulnerability of the microarchitectural component is 10 bit × cycles (=2 bits × 5 cycles).

The dotted region in Figure 1 shows an example of the vulnerability estimation for register 0. In this example, the CPU writes a new value to register 0 at cycle 1 and reads the value of register 0 at cycle 2. If a soft error corrupts the value of register 0 between cycles 1 and 2, the CPU reads the corrupted value of register 0 at cycle 2, which may lead to system failures in the worst case. Therefore, register 0 is vulnerable during cycles 1–2. Similarly, cycles 2–3 are also vulnerable because of the read operation at cycle 3. On the other hand, even if a soft error corrupts register 0 between cycles 3 and 4, the corrupted value will be overwritten at cycle 4. Therefore, register 0 is not vulnerable during cycles 3–4. Similarly, register 0 is not vulnerable during cycles 4–5 due to the write behavior at cycle 5. Consequently, register 0 is vulnerable for two cycles (1–2 and 2–3). Because register 0 consists of 32 bits in this example, the quantitative vulnerability of register 0 is 64 bit × cycles.

Note that the gemV-tool can estimate the vulnerability in just a single simulation without additional simulations or preliminary experiments. This is because the gemV-tool can capture and analyze the microarchitectural behaviors of every hardware component with a single gem5 simulation.

3.1. Fine-Grained Vulnerability Modeling by Tracing Microarchitectural Behaviors

The gemV-tool implements fine-grained vulnerability modeling in its estimation in contrast to the previous vulnerability estimation models, which calculate vulnerability in large chunks. In the actual hardware, only specific bits that are read are vulnerable to soft errors, and even they are vulnerable only during the specific time that they are accessed. Therefore, the gemV-tool produces better results compared to previous works by estimating vulnerability using smaller units in both the spatial and temporal domains.

Consider Figure 2, which shows an example of fine-grained vulnerability estimation of the rename queue and IEW (issue, execute, and writeback) queue for add and store instructions. In the processor, an instruction usually goes through the fetch, decode, rename, IEW, and commit stages in a pipeline. Pipeline queues (note that our vulnerability modeling on pipelinequeues is based on the gem5 simulator. Given that the gem5 simulator represents pipeline registers differently from the RTL approach modeling, it can result in inaccurate vulnerability compared to the RTL modeling) transfer information about the instructions between stages. For example, the rename queue transfers useful data between the rename stage and IEW stage, such as the sequence number (SeqNum), flags, source register indexes, and the destination register index. However, different types of instructions require different amounts of data. For instance, an add instruction adds register values of the source registers and writes the result to the destination register. On the other hand, a store instruction updates the memory using data from the source registers, but does not require the destination register. This can lead to inaccuracies when estimating vulnerability in a coarse-grained manner.

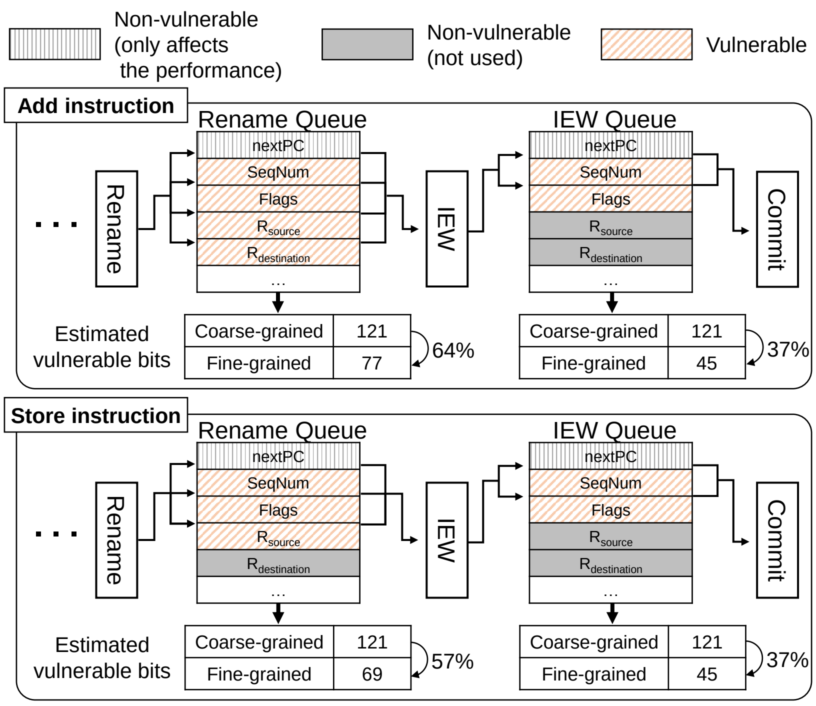

Figure 2. Fine-grained vulnerability tracking for pipeline queues for simple instructions.

In the spatial aspect, the fine-grained vulnerability estimation of the gemV-tool tracks only the accessed fields in the pipeline queues. For example, all pipeline queues in processors that use branch prediction hold the predicted next PC address for better performance. Even if this field is corrupted by a soft error, the system does not crash but only has a minor degradation in performance. Thus, the predicted next PC is not vulnerable, regardless of the instructions in the queue. In addition, the vulnerability of the fields differs depending on the instructions executed. An add instruction updates the destination register and, therefore, induces vulnerability to the destination register field of the rename queue ($R_{destination}$ in Figure 2). On the other hand, a store instruction does not update the destination register and, therefore, it does not have vulnerable periods in the destination register field.

In the temporal aspect, the gemV-tool tracks only the vulnerable duration of the accessed fields. For example, only the sequence number is vulnerable after the IEW stage because the other fields are not used at the commit stage. In memory operations (load and store), the memory addresses and data are not vulnerable between the fetch and rename stages because the memory reference is calculated by accessing physical registers after the rename stage. Thus, even if the bits in these fields become flawed, they are overwritten to correct values. In contrast, vulnerability estimation at a coarse-grained level would incorrectly define all the fields in the pipeline queues as vulnerable from the fetch to commit stages.

Considering these fields separately leads to a much more accurate estimation of the vulnerability. To do this, every hardware component in the gemV-tool instruments is modeled in the gem5 out-of-order processor with a *vulnerability tracker*. The vulnerability tracker is a data structure that records the read/write accesses of each field in the hardware and the type of instruction accessing the field. This information allows for instruction-specific vulnerability modeling in the finest-level granularity (bit level).

3.2. Accurate Vulnerability Modeling of Committed and Squashed Instructions

The gemV-tool achieves accurate vulnerability estimation by considering a particular case of the squashed instructions. Because squashed instructions are not actually executed, soft errors in these instructions do not easily culminate in system failures. Therefore, previous studies did not calculate the vulnerability of squashed instructions. However, we found that certain bits in specific microarchitectural components are indeed vulnerable, and the gemV-tool updates the vulnerability even for squashed instructions.

The rename map holds the current and previous mappings between architectural and physical registers. The rename map uses a history buffer to maintain changes in these mappings. Figure 3 depicts this process for an exemplary instruction: *load r1, r2*. Assume that the architectural register r1 is first mapped to the physical register indexed as 11 and then is renamed (remapped) to the physical register indexed as 21. Then, for architectural register r1, the old physical register index in the history buffer is updated to 11, and the new physical register index is updated to 21. This information is needed to roll the system back to its original state when the instruction is squashed.

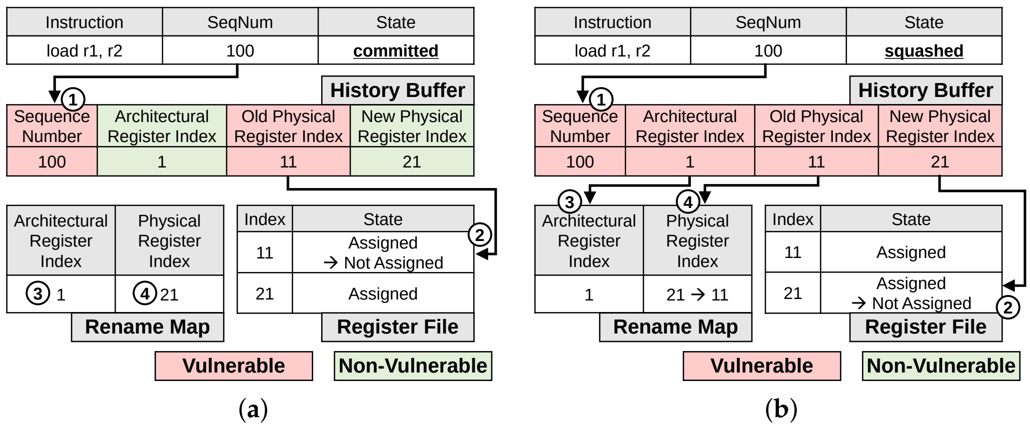

Figure 3. Accurate vulnerability estimation by considering both committed and squashed instructions. (**a**) If load instruction (load r1, r2) is committed, the history buffer does not have to restore the rename map. (**b**) If load instruction (load r1, r2) is squashed, the history buffer must restore the rename map.

Figure 3a shows the vulnerable and non-vulnerable parts of the history buffer when the instructions causing the renaming are committed. In this case, the architectural register index and new physical register index in the history buffer are not vulnerable, because the rename map holds the mapping between them (marked as ③, ④). On the other hand, the sequence number and the old physical register index are still vulnerable because these values need to be used to free the previously used physical register (marked as ①, ② in Figure 3a). If an error occurs in the sequence number (SeqNum), the entry cannot be accessed and the physical register is not released. If the old physical register index is corrupted, other registers currently in use may be released.

If the instruction causing the renaming is squashed, all four fields in the history buffer become vulnerable. As shown in Figure 3, the sequence number, architectural register index, and old physical register index are vulnerable because these values are used to undo the register renaming (marked as ①, ③, ④). The sequence number is used to index the entry in the history buffer, and the register indices are used to undo the mapping in the rename map. The new physical register index is also vulnerable because it is accessed to free the corresponding physical register (marked as ②). Interestingly, the sequence number and old physical register index in the history buffer are always vulnerable, regardless of whether the instruction is committed. Still, even for a simple benchmark, *matrix multiplication*, ignoring the vulnerability of squashed instructions, results in ignoring 35% of the total vulnerability of the history buffer.

For accurate vulnerability estimation, the gemV-tool adds a special data structure named *history* to gem5 and keeps track of recent accesses to every field of every entry in each microarchitectural component. The data structure *history* consists of *tick*, *operation*, and *sequence number*. *tick* holds the timing information of when access to a microarchitectural component takes place. *operation* holds the type of operation that took place, such as invalid, incoming, read, write, or eviction. *sequence number* holds the order of instructions executed and is used to trace whether an instruction is committed or squashed. With the help of this additional data structure, the gemV-tool correctly calculates the vulnerability of each instruction (both committed and squashed).

3.3. Comprehensive Vulnerability Modeling of All Microarchitectural Components

The gemV-tool provides comprehensive vulnerability modeling, including all microarchitectural components of an out-of-order processor. Other work in this area failed to include some microarchitectural components in their simulations. However, to break down vulnerabilities at the system level, comprehensive vulnerability modeling is required.

Consider the case in which the budget allows for the protection of only a few microarchitectural components. With the gemV-tool, we can find the hardware structures that contribute most to the overall system vulnerability. Figure 4 shows how vulnerability is distributed between microarchitectural components in the default configuration of the gem5 out-of-order processor running the *stringsearch* benchmark in ARM architecture (In this work, we omit the cache to compare the vulnerability of microarchitectural components. Because the size of the cache is much more significant than that of other components, the vulnerability of the cache is much larger than that of other components, and overshadows the differences between the vulnerabilities of other components). In this example, pipeline queues and renaming units contribute to more than half of the total system vulnerability and, therefore, should have a higher priority when applying protection techniques. These results could not have been derived from previous techniques that do not comprehensively model all microarchitectural components.

3.4. Versatile Vulnerability Modeling on Gem5 Simulator

Compared to previous tools, the gemV-tool excels in its ability to perform vulnerability modeling in a versatile environment. First, the gemV-tool supports multiple ISAs and, therefore, is able to estimate the vulnerability of a system independent of the underlying

ISA. It can be used to measure the vulnerability of the same application across different ISAs such as X86, ARM, SPARC, and ALPHA.

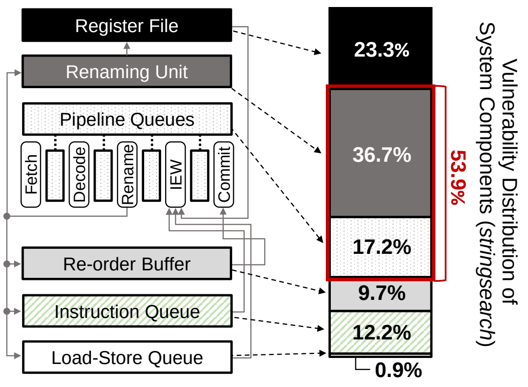

Figure 4. Comprehensive vulnerability modeling by considering all the microarchitectural components.

The gemV-tool also supports vulnerability modeling with various hardware configurations. Thus, it can be used to compare the vulnerabilities and performances of two different processors with the same ISA. For instance, both ARM Cortex-A5 and Cortex-A7 processors use the same ISA, ARM v7-A. However, Cortex-A7 provides better performance than Cortex-A5 because it supports the dual-issue superscalar pipeline unit [31]. The question of which processor guarantees better reliability against soft errors is more challenging. The gemV-tool can answer these difficult questions due to its ability to model different hardware configurations on a fixed ISA.

Further, the gemV-tool can provide accurate vulnerability modeling for various off-the-shelf commodity processors. Butko et al. [22] reported that gem5 can simulate the widely-used ARM Cortex-A9 architecture (one of the most commonly used embedded processors) with up to 99%. The accuracy of vulnerability modeling techniques relies on the accuracy of the baseline simulator used. If a microarchitectural component's behavior is not simulated correctly, the vulnerability cannot be estimated accurately. The gem5 simulator is also actively updated by both developers and engineers because it is based on an open-source infrastructure.

4. gemV-tool Validation

To validate the vulnerability estimations made by the gemV-tool, we performed extensive fault injection campaigns in all the microarchitectural components in gem5, as listed in Table 2. For each microarchitectural component, we inject a single bit-flip in a randomly chosen microarchitectural bit at a randomly selected cycle per each execution of a program in gem5. We inject 300 faults per component for each of the 10 benchmarks selected from MiBench [32] and SPEC CPU2006 [33]. Note that the gem5 simulator shares the same information on instructions among ROB, LSQ, and IQ; a bit flip into one component can affect the behavior of all three components. Thus, we modify gem5 by duplicating the fields so that a single-bit flip only impacts one particular component.

We ran 300 simulations per microarchitectural component for each benchmark in our fault injection campaigns. We have also experimentally validated that 300 runs provide stable results for all components such as register file, rename map, history buffer, instruction queue, reorder buffer, load-store queue, fetch queue, decode queue, rename queue, I2E (IEW to Execute) queue, and IEW queue on the gem5 simulator with ARM syscall emulation mode. For each component in a benchmark, we ran 2000 experiments, each consisting of 1 to 2000 fault injection simulations on ARM architecture. The results varied largely between experiments with fewer than 300 runs. However, the results became stable for the experiments with 300 or more runs, differing by less than 2% among all the experiments with over 300 runs. Thus, fault injection campaigns with 300 runs per microarchitectural component are sufficient to validate the accuracy of the gemV-tool.

Table 2. gemV-tool validation by fault injection campaigns; 300 faults were injected per component for each of the following 10 benchmarks: *hello world*, *stringsearch*, *perlbench*, *gsm*, *qsort*, *jpeg*, *matrix multiplication*, *bitcount*, *fft*, and *basicmath*.

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

Note that we only consider single-bit soft errors in our experiments. With technology scaling, the rate of multi-bit soft errors is increasing as well as soft error rates in general. However, the rate of multi-bit errors is still much lower than that of single-bit errors. For instance, the rate of double-bit soft errors is just 1/100 compared to that of single-bit soft errors [34]. Thus, we do not consider multiple-bit soft errors in this work.

We classify the 3000 simulations for each microarchitectural component (300 injections $\times$ 10 benchmarks) as matched or mismatched cases. For example, Table 2 shows 2899 matched and 101 mismatched results for the register file. A simulation is classified as a match if the results of the injection and gemV-tool agree: if the injected fault causes a failure and gemV-tool returns that the selected bit is vulnerable at the injected cycle, or if the fault does not result in failure and gemV-tool returns non-vulnerable. Otherwise, the simulation was classified as a mismatch. The fault injection experiment is declared as a failure if the system crashes, halts, or produces an incorrect output. For example, if the gemV-tool predicts that a bit is vulnerable at a specific cycle, then the corresponding fault injection experiment should result in an incorrect output or program failure to be classified as a match. The vulnerability is estimated by the gemV-tool, as described in Section 3. The accuracy of the gemV-tool with fault injection simulations for each component was defined as $\frac{The Number\, of\, Matched\, Cases}{Total\, Number\, of\, Simulations}$. For the register file, we observed that 2899 out of 3000 simulations matched, as shown in Table 2, resulting in an accuracy of 96.63%.

Further, we performed additional experiments for one benchmark matmul to validate the accuracy of each hardware component in a single application. We randomly injected 3000 faults for 11 hardware components: register file, rename map, history buffer, instruction queue, reorder buffer, load-store queue, fetch queue, decode queue, rename queue, I2E (IEW to Execute) queue, and IEW queue (a total of 33,000 fault injections for the single benchmark). And the accuracy is 96% on average. Thus, our gemV-tool is validated for applications and hardware components.

Note that we need to adjust our results for individual components to calculate the overall accuracy of the gemV-tool for the entire processor. The soft error rate of each component is proportional to the size of each component; therefore, the simulation results of each component must be scaled to its size. For instance, if the sizes of components A and B are 99 and 1, respectively, the soft error rate of A is 99 times larger than that of B. Then, if the accuracies of vulnerability estimations in A and B are 50% and 10%, respectively, the overall accuracy of A and B should be calculated as $\frac{0.5\times 99+0.1\times 1}{99+1} = 49.6\%$, not $\frac{0.5+0.1}{2} = 30\%$. Thus, the overall accuracy of the gemV-tool should be calculated considering the size of each component: $\frac{\sum_{Component=k}^{All\, Components} Size_k \times Accuracy_k}{Total\, Size}$. Table 2 lists the results of the fault injection experiments for each microarchitectural component. The results show that the estimated vulnerability of each component using the gemV-tool is approximately 97% accurate.

Vulnerability estimation of the gemV-tool seems highly accurate for all the microarchitectural components in the processor. The main reason for the inaccuracies was software-level masking. Because the gemV-tool works at the architectural level, masking behaviors at the software level are invisible to gemV-tool. Even though benchmarks with minimal software-level masking effects were used in the experiments, 3% of the gemV-tool results disagree with those of the fault injection. One source of software masking is dynamically dead instructions. If the result of an instruction is no longer used, the instruction is considered dynamically dead [15], and naturally, the instruction does not affect the program output. It is possible that the gemV-tool concludes that specific bits are vulnerable because they are used by an instruction, but in fact, the instruction is dynamically dead and the bits are not vulnerable.

Another reason for the discrepancy is the masking effects of logical instructions [15,35]. Assume that the result of a logical AND of two input registers is stored in the destination register. If the value of one input register is zero, then the result of the AND operation is zero regardless of the other input. In this case, the gemV-tool would consider the bits in the other input vulnerable, whereas the injection experiment would report non-failure. Similarly, OR operations can also mask the injected faults to input if the other operand is 1.

Incorrect program flow can also result in a mismatched case because it is considered vulnerable in the gemV-tool, whereas it still produces the correct output in the fault injection trial. For example, soft errors on the PC address or branch target address can induce incorrect program flow, but in some cases still produce the correct output [36]. Our analyses reveal that most mismatched cases fit into one of the stated categories. In all three cases, the gemV-tool is considered a bit vulnerable, when in fact, it is not vulnerable. Even with mismatched cases, the gemV-tool overestimates the vulnerability of a system and avoids the worst case of classifying a vulnerable bit as non-vulnerable. We believe that the accuracy of the gemV-tool can be improved by considering software-level masking effects.

5. gemV-tool for Fast and Early Design Space Exploration

The gemV-tool is able to calculate the vulnerabilities of the microarchitectural components for several benchmarks, as shown in Figure 5. Because the metric of vulnerability is *bit* × *cycle*, the vulnerability tends to be higher for time-consuming benchmarks. To compare the vulnerabilities fairly, we also calculated the architectural vulnerability factor (AVF) of each benchmark using Equation (1). The AVFs of the tested benchmarks varied from 7% (benchmark *patricia*) to 16% (benchmark *qsort*). For instance, 10% of soft errors may cause system failures in *stringsearch* according to the AVF estimation.

$$AVF = \frac{Vulnerability\ (bit\ \times\ cycles)}{Total\ Size\ (bits)\ \times\ Execution\ Time\ (cycles)} \tag{1}$$

The gemV-tool allows for fast design space exploration at the early design stage. Other techniques, such as neutron beam testing, require developers to build an entire working prototype before evaluating its reliability. Even register–transfer level fault injection requires developers to bring down the design to a synthesizable form before reliability can be quantified. In contrast to these methods, the gemV-tool allows hardware architects, software engineers, and system designers to evaluate the reliability at a very early high-level design stage before implementing a physical prototype. For instance, we performed our gemV-tool on an Intel Xeon processor. For the microarchitecture design experiment in Section 5.1, we have run more than 13 k different runs with different hardware configurations, and it takes less than 1 h when we use 40 cores.

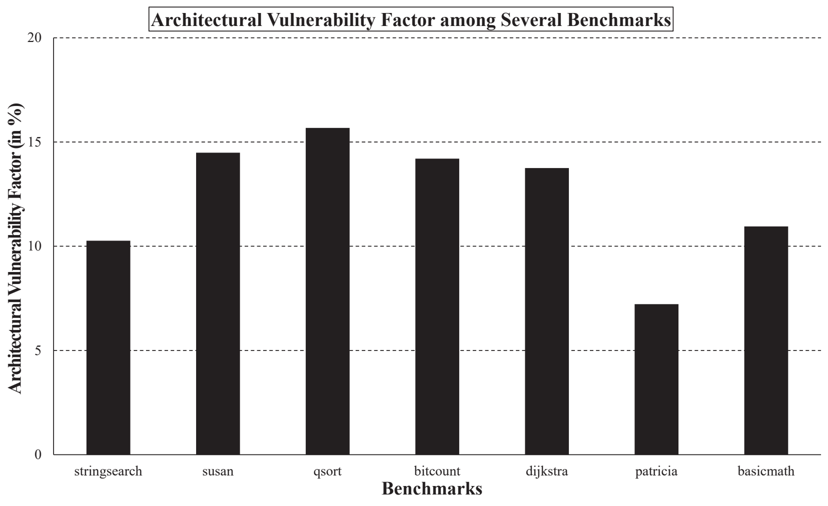

Figure 5. Architectural vulnerability factors of several benchmarks.

5.1. gemV-tool for Microarchitecture Design

The gemV-tool can be utilized to address difficult performance-vulnerability trade-off questions. For example, how does the issue width of a processor affect runtime and vulnerability? On one hand, a broader issue width could reduce the runtime and, therefore, the vulnerability. On the other hand, a broader issue width requires more sequential components in the processor, which could increase the vulnerability. With gemV, we can easily simulate the effect of such parameter changes and quantitatively answer these difficult questions. For the benchmark *stringsearch*, we observe that vulnerability decreases when increasing the issue width from 1 to 3, as shown in Figure 6. The vulnerability and runtime (Note that our gemV-tool does not depend on any specific CPU cycle. For example, we have used a 500 MHz CPU as a base clock in the case of our experiment, but a gemV-tool user can configure it if needed.) were normalized to those of the underlying configuration (issue width = 8).

Interestingly, the vulnerability and the runtime both decrease as the issue width increases. Moreover, the issue width affects the vulnerability more than the runtime. For example, if the issue width is decreased from 8 to 1, the vulnerability increases to 240%, whereas the runtime only increases to 125%.

Another interesting question is how does the size of a component affect the overall runtime and vulnerability of the architecture? We answer this question with the gemV-tool by changing the size of the LSQ in the *stringsearch* benchmark. Figure 7 shows the results, where the vulnerability and runtime were normalized to the initial configuration (LSQ size = 64). When the size of the LSQ is increased from 4 to 256, the runtime decreases monotonically. On the other hand, the vulnerability decreases at first but starts to increase after the LSQ size reaches 16. Designers can utilize this fact to find the configuration that best fits their needs. It is also worth noting that in this case, the performance is more sensitive to the change in LSQ size than the vulnerability, in contrast to the results of the experiments with varying issue widths. Thus, designers should be aware of these sensitivities when selecting the optimal configuration for microarchitectural components.

These questions can be extended to explore larger design spaces. Given an existing processor configuration and performance leeway, how can we change configurations to minimize vulnerability? We answer this question for the benchmark *stringsearch* with the gemV-tool by plotting design points for runtime against the vulnerability. We assume that the number of physical registers in the register file is fixed at 256. We also assume that the number of entries in the rename map, history buffer, and the IEW queue are also fixed to 114, 86, and 8, respectively. We then considered the entry size for each component: 64, 128, 192, 156, 320, and 384 as the possible number of entries of ROB; 4, 8, 16, 32, 64, 128, and 256 as that for LSQ and IQ; and 1 through 8 for pipeline queues such as fetch, decode, rename, and I2E. We ran the experiments with randomly selected entry sizes for ROB, LSQ, IQ, and pipeline queues from the given ranges and plotted the results in a two-dimensional plane. The resulting points were divided into four quadrants: the first quadrant contained points with positive values for both runtime and vulnerability, the second had negative runtime and positive vulnerability, the third had negative runtime and vulnerability, and the fourth had positive runtime and negative vulnerability. A positive value represents an increase compared to the original configuration, whereas a negative value represents a decrease. For example, point (10, −5) represents a 10% increase in runtime and a 5% decrease in vulnerability as compared to the original configuration. Figure 8 shows the vulnerability and runtime for the configurations normalized to those of the initial configuration (192 entries for ROB, 64 entries for LSQ and IQ, and 8 entries for all the pipeline queues).

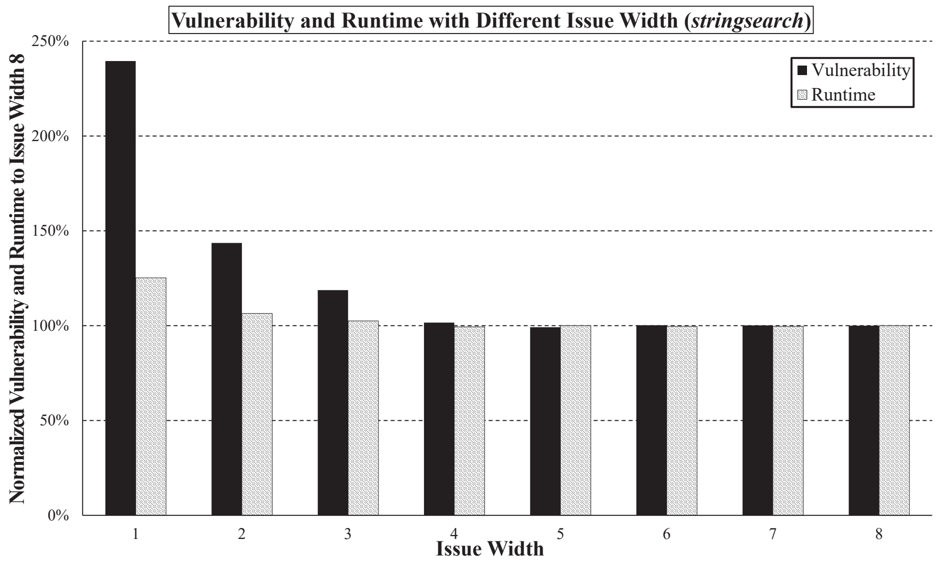

Figure 6. Normalized runtime and vulnerability depending on issue width.

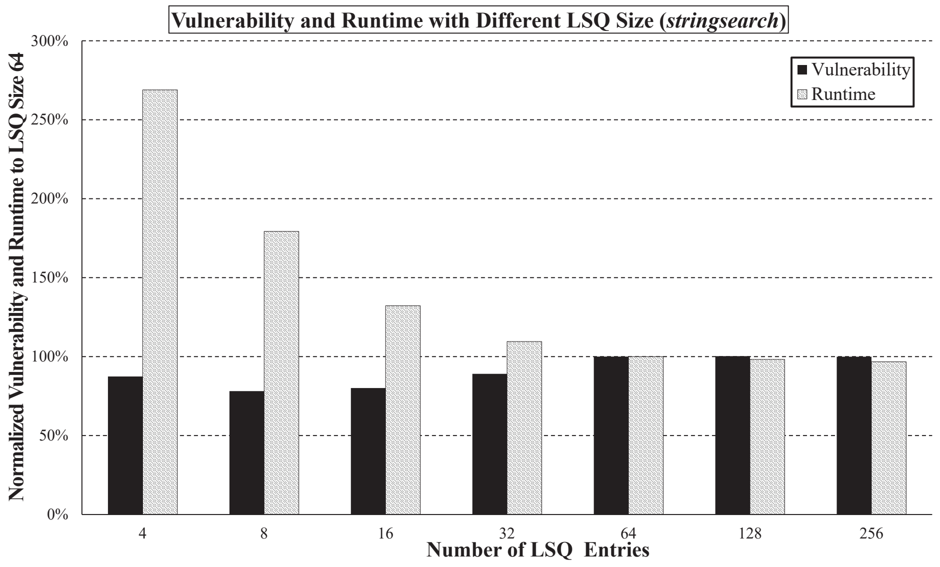

Figure 7. Normalized runtime and vulnerability, depending on the LSQ size.

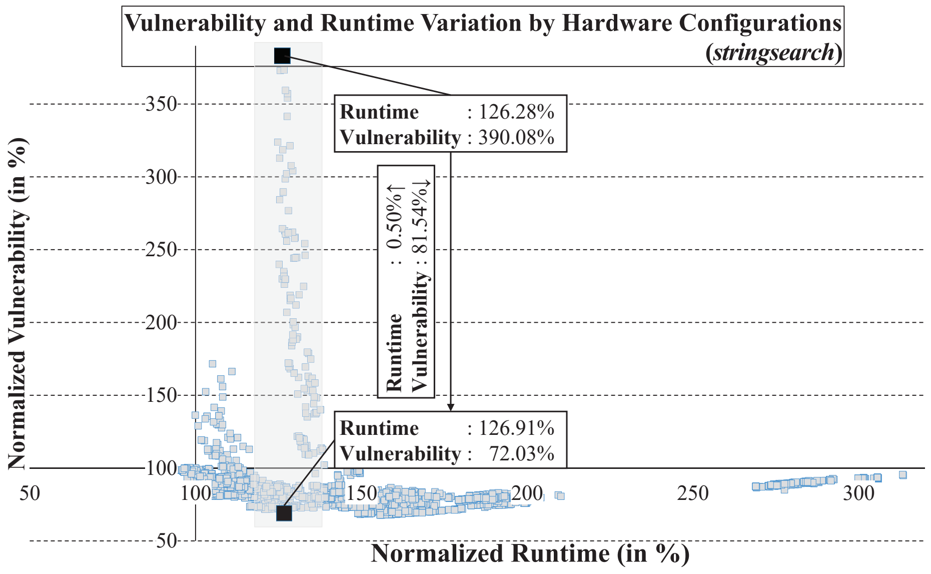

Figure 8. Normalized runtime and vulnerability for various hardware configurations, plotted as runtime and vulnerability.

A hardware designer can use these plots to choose the required hardware configuration dictated by the runtime and vulnerability bounds. For instance, given a particular runtime target, the hardware designer can choose from the points in the grey band in Figure 8 to find the configuration with maximum vulnerability reduction. It is interesting to note that vulnerability can change significantly by simply changing the configurations without applying any protection. With a 1% runtime variation, the vulnerability can be reduced by up to 81%, as shown by the two black points in Figure 8.

We ran the same experiment for the *matrix multiplication* benchmark, running more than 1.2 million trials to explore all possible hardware configurations. The results are shown in Figure 9. In this experiment, we found two design points that differed by less than 1% in the runtime, but by 37% in vulnerability. This is surprising, considering that the vulnerability ranges from 91% to 148%. Thus, given any runtime or vulnerability overhead, it is now possible to find alternate design points with lower vulnerability or runtime. This feature of the gemV-tool can be very useful for designers to explore the design space at the early design stage.

Another interesting observation is that each hardware component differs in the extent of vulnerability reduction. We explore this observation by changing the size of each component independently, and present the following results. Changes in the number of ROB entries did not have significant impacts on vulnerability (14% maximum). Entry size changes for LSQ and IQ, however, can influence the vulnerability by up to 44% and 55%, respectively. In particular, the LSQ also affects the performance by up to 85%. In the pipeline queues, the variance in the entry size can cause up to a 50% increase in vulnerability and a 20% increase in runtime. This analysis can guide hardware designers in selecting the best configuration with a limited total number of sequential elements in the system. Figure 10 shows the maximal vulnerability reduction when the total number of entries is fixed. The x-axis represents the sum of the number of entries in each component, and the y-axis shows the maximal vulnerability reduction percentage. When the total number of entries is limited to less than 300, there exists a configuration with 42.7% less vulnerability compared to the configuration with the maximum vulnerability under the same conditions. The graph shows that the variation in vulnerability becomes more substantial as the total number of sequential elements increases. Therefore, determining the optimal configuration is critical in terms of vulnerability for more complex systems.

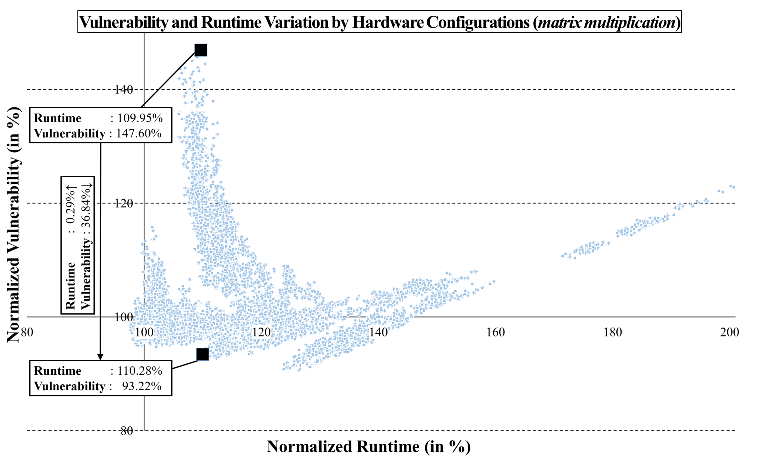

Figure 9. Vulnerability and runtime with different hardware configurations (*matrix multiplication*).

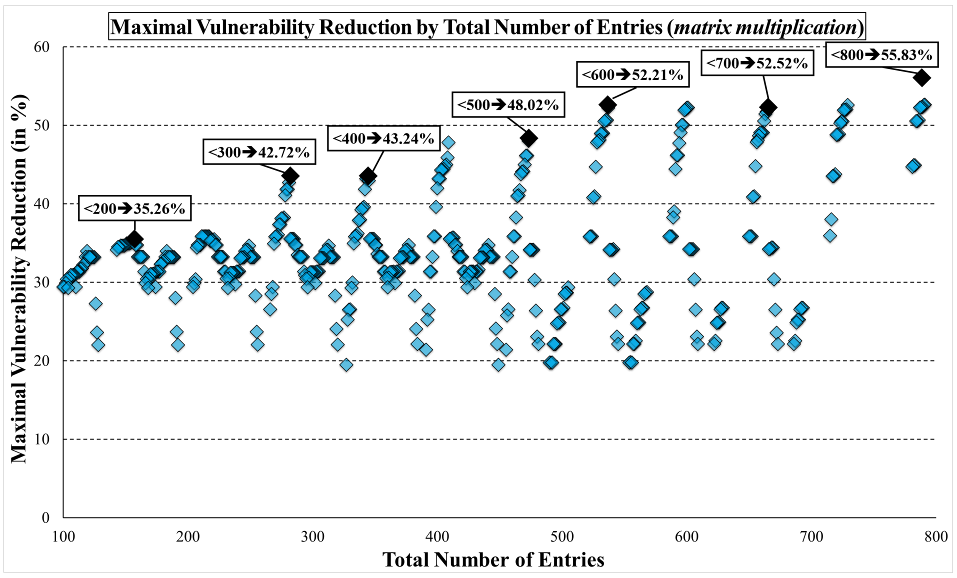

Figure 10. Vulnerability variation by fixing the number of sequential elements.

5.2. *gemV-Tool for Software Design*

The gemV-tool can also be used by software engineers to find alternate design points with lower vulnerability or runtime. Alternative design points can be realized with software changes in either the algorithm, compiler used, or optimization level. For example, given the choice of two sorting algorithms—quick sort and insertion sort—which would be the optimal choice in terms of runtime and vulnerability? To explore such changes, we first establish a baseline runtime and vulnerability for an insertion sort algorithm compiled with *gcc* at the highest (O3) level of optimization. Figure 11 presents the normalized runtime and vulnerability for various configurations of algorithms, compilers, and optimization levels. We consider an array sorting application with five sorting algorithms (bubble, quick, insertion, selection, and heap sort), two compilers (GCC and LLVM [37]), and four optimization levels (no optimization, O1, O2, and O3). Interestingly, just by changing the software configurations, the vulnerability can be reduced by up to 91% without additional runtime overhead. Specifically, switching from a selection sort algorithm at the O1 level of optimization to quick sort at the O3 level of optimization reduces runtime by 53% and vulnerability by 91%. A software engineer can use these experiments to explore the design space and choose optimal design points according to their demands.

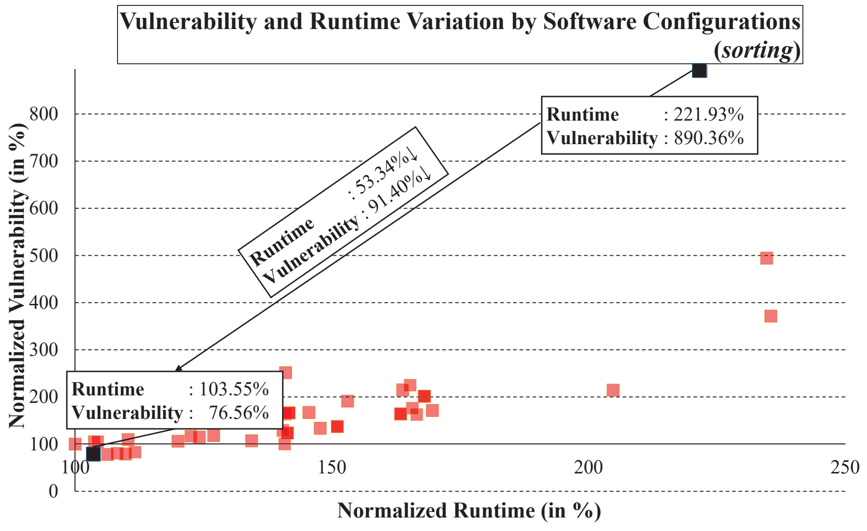

Figure 11. Normalized runtime and vulnerability depending on software configurations.

Table 3 summarizes the effects of different software configurations on vulnerability and runtime. In general, vulnerability is much more sensitive to software configuration than runtime. The essential software option is the sorting algorithm. The vulnerability can be increased to up to 10 × depending on the selected algorithm—from the selection sort compiled by GCC with the O1 option to quick sort compiled with the same options. The choice of the algorithm also leads to a maximum difference of up to approximately 114% in runtime. The factor that least affects vulnerability is the compiler, but it can still result in up to 314% variation in vulnerability and 52% in runtime. Depending on the compiler optimization level, the vulnerability can change by up to 739% and the runtime by up to 101%. Similar vulnerability-aware design space exploration in software can allow software designers to meet specific requirements in runtime, vulnerability, or both.

Table 3. 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×

5.3. gemV-Tool for System Design

A system designer can use the gemV-tool to make design choices. Consider the following questions: (i) Given a selection of processors running on different ISAs, which one offers the best trade-off in runtime and vulnerability? We explore this problem by changing the ISA within the gemV-tool while keeping the hardware size constant. Figure 12 shows the vulnerability and runtime under different ISAs (ARM, SPARC, x86, and ALPHA) for the *stringsearch* benchmark, with no change in hardware and software configurations. Baseline vulnerability and runtime were established on the ARM ISA. In this experiment, ALPHA was the least vulnerable, being 38% less vulnerable than SPARC, which was the most vulnerable. With this result, the system designer may choose ARM ISA for the minimum runtime or ALPHA for minimum vulnerability.

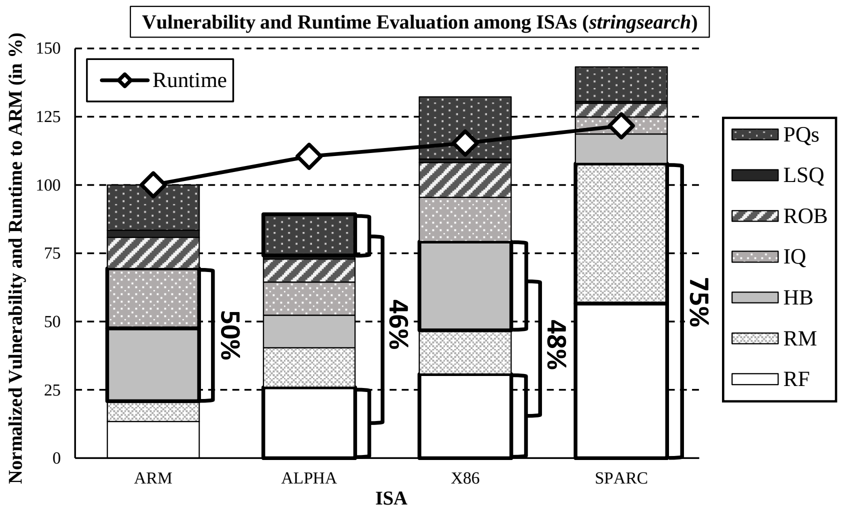

Figure 12. Variation in runtime and vulnerability for *stringsearch* under different ISAs. Bars show vulnerability, and diamond points indicate runtime.

(ii) Given an ISA to work on, which components are the most vulnerable? The system designer may also need to break down the vulnerability of individual hardware components. Each column in Figure 12 shows the detailed breakdown of each component in the processor, consisting of HB (history buffer), RM (rename map), LSQ, IQ, PQs (pipeline queues), RF (register file), and ROB. The figure shows that protecting just two components can lead to a significant reduction in vulnerability. For example, the history buffer and IQ account for 50% of the vulnerability in the ARM processor, and the rename map and register file account for 75% of SPARC. This information can be used to design protection techniques that target specific components. In the case of SPARC, a simple protection mechanism such as ECC applied to the register file would be extremely efficient and effective. However, the same protection would not be as useful on the ARM processor as the RF contributes only 21% to the system vulnerability.

6. Conclusions

Because reliability has become an important design concern in modern embedded systems, various protection techniques against soft errors have been presented. This also necessitated reliability quantification schemes to quantitatively study the effectiveness of such protection techniques. However, accurate reliability quantification schemes such as exhaustive fault injection campaigns and neutron beam testing are undesirable because they are too expensive and challenging to perform. Other techniques, including vulnerability estimation tools, are incomprehensive, inaccurate, and inflexible. In this paper, we present the gemV-tool—a comprehensive and accurate vulnerability estimation based on the cycle-accurate simulator gem5. We also performed several experiments to demonstrate how the gemV-tool can help engineers in design space exploration in the early design phase. We show the effects of microarchitectural changes on runtime and vulnerability, which are relevant to hardware designers. For software designers, we show the effects of the algorithm, compiler, and optimization level on runtime and vulnerability. We also demonstrated the usefulness of the gemV-tool to a system designer in designing component-specific or ISA-dependent soft error protection techniques.

The gemV-tool is useful for early and fast design space exploration of reliability against soft errors. It answers fundamental questions at several design space abstraction levels.

(i) **At the microarchitecture design level,** how does the issue width (single-issue or dual-issue) of a processor affects its vulnerability and performance? How can a microarchitect determine the optimal issue width for the processor?

(ii) **t the system software design level**, how can software system designers improve the reliability against soft errors? How does the algorithm or compiler's optimization level affect the runtime and vulnerability of an application?

(iii) **At the architectural design level**, architecture designers can choose a different ISA for better performance or power, but how can they ensure that protection against soft errors? How does the soft error vulnerability of an application depend on the ISA?

Such trade-off questions between runtime and vulnerability at all levels can now be answered quickly and accurately, using the gemV-tool. To demonstrate the capabilities of the gemV-tool, we performed a broad range of design space explorations and made the following observations:

- Vulnerability decreases when the issue width increases from 1 to 3. Beyond this, any increase in the issue width does not have a noticeable effect on vulnerability. We also find that vulnerability is also correlated with other architectural parameters, such as the number of entries in the reorder buffer (ROB), instruction queue (IQ), load/store queue (LSQ), or pipeline queues. Among the many configurations, there is an interesting design configuration with 82% less vulnerability while incurring a performance penalty of $< 1\%$.
- The vulnerability varies significantly depending on the algorithm implemented. For instance, our experimental results show that changing from a selection sort to a quick sort algorithm can affect the system vulnerability by 91%. This can help software engineers find the least vulnerable and fastest algorithm for an application.
- The distribution of system vulnerabilities among microarchitectural components is quite sensitive to ISA. For example, protecting only the register rename map and register file in SPARC architecture can lead to more than 75% vulnerability reduction, while it only lead to a 21% reduction in the ARM architecture. In the latter case, it is better to protect the history buffer and instruction queue (more than 50% vulnerability reduction).

Unlike the other vulnerability estimation tools, we verified the accuracy of the gemV-tool by comparing it against fault injection experiments on all the microarchitectural components. The validation, however, assumed that each microarchitectural component's soft error rate is proportional to its size. As this assumption cannot be verified in simulated environments, we plan to validate the accuracy of the gemV-tool through neutron beam testing in the future. In addition, the gemV-tool will also model and characterize software-level masking effects in the future. Since the incorrect microarchitectural state does not always result in system failures, Papadimitriou et al. [38] have exploited fault injection campaigns on the cycle-accurate simulator to track the software-level masking effects for silent data corruption analysis. After analyzing the software-level masking, we will no longer classify dynamically dead instructions or non-influential program flow changes as vulnerable, thereby improving the accuracy of the gemV-tool.

Author Contributions: Methodology H.S.; Conceptualization Y.K.; writing J.J.; supervision, K.L.; project administration A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by "Regional Innovation Strategy (RIS)" through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (MOE) in 2023(2022RIS-005), National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2022-00165225), and National Science Foundation Grants No. CPS 1646235.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Baumann, R. Soft errors in advanced computer systems. *IEEE Des. Test Comput.* **2005**, *22*, 258–266. [CrossRef]
2. Dixit, A.; Wood, A. The impact of new technology on soft error rates. In Proceedings of the International Reliability Physics Symposium, Monterey, CA, USA, 10–14 April 2011; pp. 5B.4.1–5B.4.7
3. Kim, B.; Yang, H. Reliability Optimization of Real-Time Satellite Embedded System Under Temperature Variations. *IEEE Access* **2020**, *8*, 224549–224564. [CrossRef]
4. Yoshida, J. Toyota Case: Single Bit Flip That Killed. *EE Times*, 25 October 2013, p. 8.
5. Lee, I.; Basoglu, M.; Sullivan, M.; Yoon, D.H.; Kaplan, L.; Erez, M. *Survey of Error and Fault Detection Mechanisms*; Technical Report; University of Texas: Austin, TX, USA, 2011; Volume 11, p. 12.
6. Ko, Y.; Jeyapaul, R.; Kim, Y.; Lee, K.; Shrivastava, A. Guidelines to Design Parity Protected Write-back L1 Data Cache. In Proceedings of the Design Automation Conference, ACM, —DAC '15, San Francisco, CA, USA, 7 June 2015; pp. 24:1–24:6.
7. Jahinuzzaman, S.; Gill, B.; Ambrose, V.; Seifert, N. Correlating low energy neutron SER with broad beam neutron and 200 MeV proton SER for 22nm CMOS Tri-Gate devices. In Proceedings of the Reliability Physics Symposium (IRPS)—2013 IEEE International, Monterey, CA, USA, 14–18 April 2013; pp. 3D.1.1–3D.1.6. [CrossRef]
8. Bodmann, P.R.; Papadimitriou, G.; Junior, R.L.R.; Gizopoulos, D.; Rech, P. Soft error effects on arm microprocessors: Early estimations versus chip measurements. *IEEE Trans. Comput.* **2021**, *71*, 2358–2369. [CrossRef]
9. Entrena, L.; Garcia-Valderas, M.; Fernandez-Cardenal, R.; Lindoso, A.; Portela, M.; Lopez-Ongil, C. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *IEEE Trans. Comput.* **2012**, *61*, 313–322. [CrossRef]
10. Nguyen, H.T.; Yagil, Y. A systematic approach to SER estimation and solutions. In Proceedings of the International Reliability Physics Symposium, Dallas, TX, USA, 30 March–4 April 2003; pp. 60–70.
11. Alkhalifa, Z.; Nair, V.S.; Krishnamurthy, N.; Abraham, J.A. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.* **1999**, *10*, 627–641. [CrossRef]
12. Shrivastava, A.; Rhisheekesan, A.; Jeyapaul, R.; Wu, C.J. Quantitative Analysis of Control Flow Checking Mechanisms for Soft Errors. In Proceedings of the Design Automation Conference—ACM, DAC '14, San Francisco, CA, USA, 1–5 June 2014; pp. 13:1–13:6.
13. Cho, H.; Mirkhani, S.; Cher, C.Y.; Abraham, J.A.; Mitra, S. Quantitative evaluation of soft error injection techniques for robust system design. In Proceedings of the Design Automation Conference, Austin, TX, USA, 29 May–7 June 2013; pp. 1–10.
14. Mukherjee, S.S.; Emer, J.; Reinhardt, S.K. The soft error problem: An architectural perspective. In Proceedings of the International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA , 12–16 February 2005; pp. 243–247.
15. Mukherjee, S.S.; Weaver, C.; Emer, J.; Reinhardt, S.K.; Austin, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture—IEEE Computer Society—MICRO 36, San Diego, CA, USA, 5 December 2003; p. 29.
16. Biswas, A.; Racunas, P.; Emer, J.; Mukherjee, S.S. Computing accurate AVFs using ACE analysis on performance models: A rebuttal. *Comput. Archit. Lett.* **2008**, *7*, 21–24. [CrossRef]
17. Jeyapaul, R.; Shrivastava, A. Smart Cache Cleaning: Energy Efficient Vulnerability Reduction in Embedded Processors. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM—CASES'11, Taipei, Taiwan, 9–14 October 2011; pp. 105–114.
18. Li, X.; Adve, S.V.; Bose, P.; Rivers, J.A. SoftArch: an architecture-level tool for modeling and analyzing soft errors. In Proceedings of the International Conference on Dependable Systems and Networks, Yokohama, Japan, 28 June–1 July 2005; pp. 496–505.
19. Fu, X.; Li, T.; Fortes, J. Sim-SODA: A unified framework for architectural level software reliability analysis. In Proceedings of the Workshop on Modeling, Benchmarking and Simulation, 30 June 2006. Available online: http://www-mount.ece.umn.edu/~jjyi/MoBS/2006/program/2A-Fu.pdf (accessed on 19 December 2015).
20. gemV-tool. Available online: https://github.com/MPSLab-ASU/gemV (accessed on 19 December 2015).
21. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *ACM Sigarch Comput. Archit. News* **2011**, *39*, 1–7 [CrossRef]
22. Butko, A.; Garibotti, R.; Ost, L.; Sassatelli, G. Accuracy evaluation of GEM5 simulator system. In Proceedings of the International Workshop on Reconfigurable Communication-centric Systems-on-Chip, York, UK, 9–11 July 2012; pp. 1–7.
23. Seifert, N.; Tam, N. Timing vulnerability factors of sequentials. *IEEE Trans Device Mater. Reliab.* **2004**, *4*, 516–522. [CrossRef]
24. Sridharan, V.; Kaeli, D.R. Quantifying software vulnerability. In Proceedings of the WREFT, ACM, New York, NY, USA, 5–7 May 2008; pp. 323–328.
25. Borodin, D.; Juurlink, B.H. Protective redundancy overhead reduction using instruction vulnerability factor. In Proceedings of the Computing Frontiers—ACM, Bertinoro, Italy, 17–19 May 2010; pp. 319–326.
26. Seongwoo, K.; Somani, A.K. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In Proceedings of the International Conference on Dependable Systems and Networks, Washington, DC, USA, 23–26 June 2002; pp. 416–425.
27. Emer, J.; Ahuja, P.; Borch, E.; Klauser, A.; Luk, C.K.; Manne, S.; Mukherjee, S.S.; Patil, H.; Wallace, S.; Binkert, N.; et al. Asim: A performance model framework. *Computer* **2002**, *35*, 68–76. [CrossRef]

28. Moudgill, M.; Bose, P.; Moreno, J.H. Validation of Turandot, a fast processor model for microarchitecture exploration. In Proceedings of the International Performance, Computing and Communications Conference, Scottsdale, AZ, USA, 12 February 1999; pp. 451–457.
29. Desikan, R.; Burger, D.; Keckler, S.W.; Austin, T. *Sim-Alpha: A Validated, Execution-Driven Alpha 21264 Simulator*; Technical Report; University of Texas: Austin, TX, USA, 2001.
30. Desikan, R.; Burger, D.; Keckler, S.W. Measuring Experimental Error in Microprocessor Simulation. In Proceedings of the International Symposium on Computer Architecture, ACM—ISCA'01, New York, NY, USA, 30 June–4 July 2001; pp. 266–277.
31. ARM. *ARM Cortex-A Processor Comparison Table*; ARM: Cambridge, UK, 2020.
32. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the International Workshop on Workload Characterization, Austin, TX, USA, 2 December 2001; pp. 3–14.
33. Henning, J.L. SPEC CPU2006 benchmark descriptions. *Acm Sigarch Comput. Archit. News* **2006**, *34*, 1–7. [CrossRef]
34. Lee, K.; Shrivastava, A.; Kim, M.; Dutt, N.; Venkatasubramanian, N. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. In Proceedings of the ACM Multimedia, New York, NY, USA, 26–31 October 2008; Volume 8, pp. 319–328.
35. Ma, J.; Duan, Z.; Tang, L. A Methodology to Assess Output Vulnerability Factors for Detecting Silent Data Corruption. *IEEE Access* **2019**, *7*, 118135–118145. [CrossRef]
36. Wang, N.; Fertig, M.; Patel, S. Y-Branches: When You Come to a Fork in the Road, Take It. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques—PACT'03, New Orleans, LA, USA , 27 September–1 October 2003; p. 56.
37. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, IEEE Computer Society—CGO'04, San Jose, CA, USA, 20–24 March 2004; p. 75.
38. Papadimitriou, G.; Gizopoulos, D. Silent Data Corruptions: Microarchitectural Perspectives. *IEee Trans. Comput.* **2023**, *72*, 3072–3085. [CrossRef]