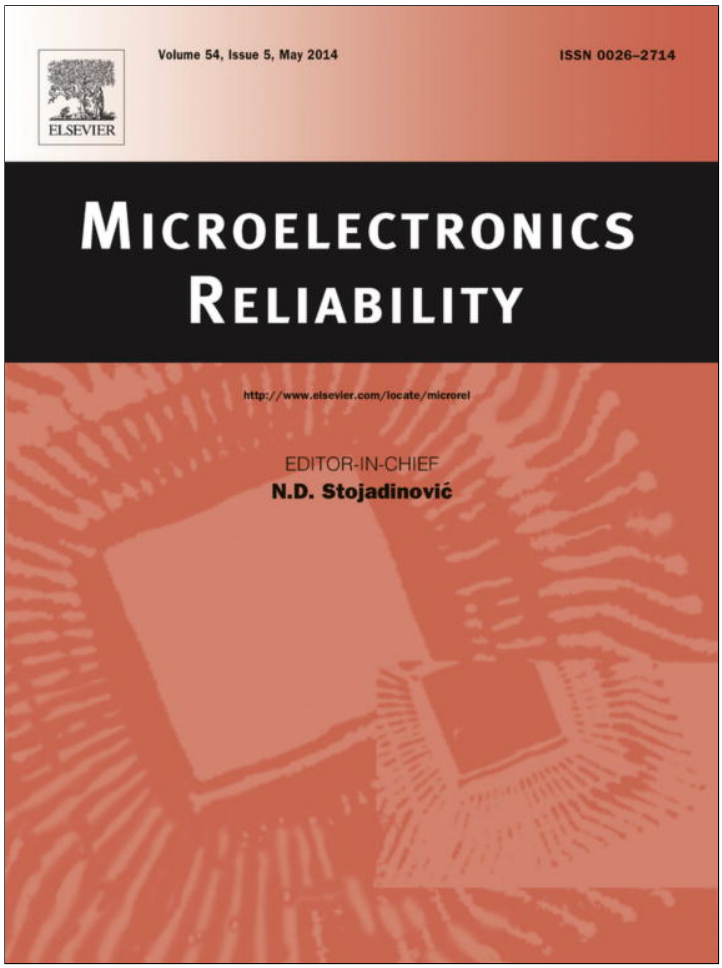

Microelectronics Reliability 54 (2014) 1009–1021

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

CEDAR: Modeling impact of component error derating and read frequency on system-level vulnerability in high-performance processors

Hossein Asadi [a,*], Alireza Haghdoost [a], Morteza Ramezani [a], Nima Elyasi [a], Amirali Baniasadi [b]

[a] Department of Computer Engineering, Sharif University of Technology, Islamic Republic of Iran
[b] Department of Electrical and Computer Engineering, University of Victoria, Canada

ARTICLE INFO

Article history:
Received 16 July 2013
Received in revised form 8 January 2014
Accepted 8 January 2014
Available online 5 February 2014

ABSTRACT

Reliability of the current microprocessor technology is seriously challenged by radiation-induced soft errors. Accurate *Vulnerability Factor* (VF) modeling of system components is crucial in designing cost-effective protection schemes in high-performance processors. Although *Statistical Fault Injection* (SFI) techniques can be used to provide relatively accurate VF estimations, they are often very time-consuming. Unlike SFI techniques, recently proposed analytical models can be used to compute VF in a timely fashion. However, VFs computed by such models are inaccurate as the system-level impact of soft errors is overlooked.

In this paper, we propose a system-level analytical technique, called *Component Error Derating And Read frequency* (CEDAR) vulnerability model, combining the advantages of previously presented analytical models and the SFI techniques. The key idea behind CEDAR is to take into account *component error derating* and *read frequency* for data-path blocks in high-performance processors. To further investigate the impact of read frequency and component error derating on the system-level VF, we use *Input-to-Output Derating* (IOD) factor of system components in the proposed analytical model. As a case study, we study system-level vulnerability for cache memory by providing IOD analysis for different processor core configurations. Our experimental results reveal that processor core IOD can significantly affect the system-level vulnerability of cache memories. The experimental results show that CEDAR improves the accuracy of previous analytical VF estimation techniques up to 91% and 5% for write-through and write-back cache memories, respectively, while it speeds up estimation time up to 10× as compared to SFI techniques.

1. Introduction

Over the past few years, radiation-induced transient errors have been a severe threat to data integrity in high-end and mainstream microprocessors. These errors, also referred to as *soft errors*, are initiated by alpha particles from packaging, high-energy neutrons, and low-energy neutrons interacted with the isotope Boron-10 (10B) [1]. A field study over several thousands of systems indicates that in the current processor technology, the majority f system reboots are initiated by *Single Event Upsets* (SEUs) occurring in data-path components such as cache memory and register files [2]. Errors in such structures can easily propagate to the system outputs and significantly reduce the overall system reliability.

With continuous downscaling of CMOS technology into the *nanometer* era coupled with the increasing importance of process variation in the newer CMOS technology, the threat of radiation-induced soft errors is becoming more severe than previous CMOS generations [3]. At the same time, since the number of transistors per chip continues to move up, *Soft Error Rate* (SER) per chip is expected to increase for the next several years [4]. Accordingly, designers would need to incorporate aggressive protection techniques in future microprocessor designs. As a result, the issue of estimating and mitigating the impact of radiation-induced transient errors becomes more important.

A challenging aspect of designing a cost-effective protection technique is developing accurate soft error vulnerability models for individual components. Having accurate vulnerability models facilitates accurate evaluation of the SER contribution of each data-path component to the overall system SER. This helps designing cost-effective SER-aware protection schemes across data-path components and for target workloads. The right protection level for data-path structures can reduce data loss probability and hence increase the overall system reliability.

Among previous modeling techniques, *Statistical Fault Injection* (SFI) has been commonly used by designers to extract SERs or

* Corresponding author. Tel.: +98 2166166639.
E-mail addresses: asadi@sharif.edu (H. Asadi), haghdoost@ce.sharif.edu (A. Haghdoost), mramezani@ce.sharif.edu (M. Ramezani), elyasi@ce.sharif.edu (N. Elyasi), amirali@ece.uvic.ca (A. Baniasadi).

0026-2714/$ - see front matter © 2014 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.microrel.2014.01.007

Vulnerability Factors (VFs) of system components [5–9]. The concept of VF has been first introduced by Biswas et al. [10] to estimate processor vulnerability to soft errors in early design stages of processors. Although SFI techniques can provide relatively accurate VFs, they are often very time-consuming. In particular, extracting VF for system components is an order of magnitude more time-consuming than extracting VF of the overall system. Therefore, given the limited design time frame, the computed VFs for system components using SFI techniques may have large variances. As a result, SFI techniques are either time-consuming, due to the large number of SFI runs, or prone to inaccuracy, due to the limited number of addresses targeted.

Unlike SFI techniques, recently presented analytical models provide VFs in a timely fashion. Such models, however, do not take into account the system-level impact of soft errors. In previously presented analytical models, the VF of a data-path component such as cache memory or register file is computed based on the propagation probability of an error from an internal site of a data block to the block outputs [5–8]. This is done by measuring the time period in which an error occurring in a data block could potentially propagate in the system, also referred to as the *critical time*, to estimate vulnerability [11]. In these models, however, two important factors, i.e., *read frequency* and *component error derating* have been overlooked. As we show in this paper, these two factors can affect the system-level VF of a component significantly.

In this work, we first examine the shortcomings of previous analytical models and then propose a modeling technique to accurately compute system-level VF of data-path components. The proposed technique, called *Component Error Derating And Read frequency* (CEDAR) vulnerability model, improves the accuracy of previous estimation methods while preserving low estimation time. The accuracy is improved by incorporating the impact of read frequency and component error derating into earlier analytical models. In the CEDAR model, we define *Component-level Vulnerability Factor* (CVF) and *System-level Vulnerability Factor* (SVF) to account for read frequency and component error derating. CVF and SVF of a component are defined as the percentage of errors occurring within the component, propagating to the component outputs and the system outputs, respectively. Therefore, higher CVF and SVF is indicative of higher component and system failure rate, respectively.

We also demonstrate that higher CVF does not necessarily lead to higher SVF. This is explained by proposing another important parameter referred to as the *Input-to-Output Derating* (IOD) factor. The IOD of a component is the portion of errors not being masked when propagating erroneous values from primary inputs to primary outputs of the component. IOD can be computed by performing a very limited number of SFI experiments in a timely fashion. Our experimental results show that the IOD of a CPU core can vary from 23% up to 77%, depending on the application. Therefore, overlooking IOD in VF estimations can result in significant over-estimation of the system-level VF.

We also develop a reference model based on *Statistical Fault Injection* (SFI) and *Monte-Carlo* (MC) simulations and compare SVF estimations to the reference model for *SPEC2K* benchmarks as well as previously proposed analytical models. These comparisons have been provided for *Write-Through* (WT) and *Write-Back* (WB) cache memories. Our results show that the VFs obtained by CEDAR are more accurate compared to those provided by previous analytical models by 91% while CEDAR is 10× faster than the SFI technique.

A preliminary version of this work has been presented in [12,13]. This work extends our previous work by offering the following contributions:

- **Extending Access Patterns:** In this work, the accuracy has been improved by considering the impact of read frequency. To this end, we extend access patterns presented in [12,13] and compute the corresponding vulnerability factor.
- **SFI:** We develop an SFI reference model to validate the accuracy of the CEDAR model. The SFI reference model also reveals the inaccuracy of previously vulnerability models.
- **VF Accuracy Analysis in WT vs. WB Caches:** Our observations indicate that the IOD of CPU cores has higher impact on the vulnerability of WT caches than WB caches. This is due to the fact that WB caches contain dirty blocks whose vulnerability impact is often not affected by IOD. Our analysis also reveals that the previously proposed VF models have less accuracy in WT caches compared to WB caches. CEDAR improves the accuracy of vulnerability estimation for both WB and WT caches by taking the IOD of the CPU cores into account.
- **Proposed Algorithm for WT and WB Caches:** We present an algorithm to compute SVF for both WT and WB cache configurations. In the proposed algorithm, different access patterns are accurately modeled to compute the SVF of cache blocks.

The rest of the paper is organized as follows. In Section 2, we present reliability background. In Section 3, we review previously suggested vulnerability modeling techniques and their shortcomings. In Section 4, we investigate the inaccuracies existing in previous analytical techniques. In Section 5, we present the CEDAR model. In Section 6, we describe VF computation for different access patterns. In Section 7, we present the algorithm used to compute CVF and SVF for cache memory. In Section 8, we explain the experimental setup. In Section 9, we present our methodology and report results. Finally, in Section 10, we conclude the paper.

2. Background

An energetic particle striking a CMOS transistor induces a localized ionization capable to change the state of a memory cell, logic gate, latch, or flip-flop causing a soft error [4]. In the past two decades, researchers have discovered three major sources resulting in soft errors in semiconductor devices at terrestrial altitudes. These sources are (a) alpha particles, (b) high-energy neutrons, and (c) low-energy neutrons interacted with the isotope Boron-10 (^{10}B) [1].

Soft Error Rate (SER) is defined as the system failure rate due to soft errors. *Failures-in-Time* (FIT) is another commonly used error rate metric. The FIT of a component is inversely proportional to the *Mean-Time-To-Failure* (MTTF) of the component. This is shown in Eq. (1). One FIT is equal to one failure in a billion hours of system operation.

$$FIT_{rate} = \frac{10^9}{MTTF \times 24_{hours} \times 365_{days}} \tag{1}$$

The overall FIT of a chip is calculated by adding the effective FIT rates of all the individual components as follows [11]:

$$FIT_{chip} = \sum_i FIT_{Component(i)} \tag{2}$$

The FIT of each component in a chip is the product of its *raw FIT rate*, associated *Architectural Vulnerability Factor* (AVF), and *Timing Vulnerability Factor* (TVF) as follows:

$$FIT = AVF \times TVF \times RawFIT_{rate} \tag{3}$$

AVF expresses the probability that a transient fault in a storage cell (such as SRAM) results in a user visible error [14]. For example, a bit-flip in a branch predictor may cause a mis-prediction,

however, it will never result in a user-visible error. As a result, the branch predictor's $AVF = 0\%$. In contrast, a bit-flip in a program counter register will most likely crash the instruction execution sequence and produce control flow error. Therefore, a program counter's AVF is about 100%. Computing the AVF of other components such as cache memory is more complicated because an erroneous value in such components can be masked by the CPU [15].

TVF is the fraction of time for which the circuit is vulnerable to transient faults. As an example, a simple latch is vulnerable against radiation-induced faults during 50% of its clock cycle time [15]. SRAM cells (used in cache memory) are always susceptible to these faults. Therefore, the associated TVF for SRAMs is 100%.

Finally, $RawFIT_{rate}$ is the circuit-level soft error rate of a device from radiation-induced faults. $RawFIT_{rate}$ of a storage element depends on the device characteristics and the flux that comes across the device [15].

3. Related works and their limitations

In this section, we first review the limitations of conventional protection techniques. We follow with discussing previous vulnerability factor modeling techniques and their drawbacks.

3.1. Limitations of traditional protection techniques

Spatial redundancy techniques (e.g., byte or block-based parity or error correction code) are commonly used to protect data in cache [16]. Such protection techniques, however, are rarely used for tag addresses as they impose several limitations. First, redundancy incurs area and power overhead. This overhead increases proportionately with the cache size. Second, such techniques can degrade performance by increasing cache access time.

Error detection techniques such as parity can be used to detect errors in tag arrays. However, the recovery of an erroneous tag array is either impossible or very difficult [11] under such techniques. Scrubbing is an alternative technique that could be used to improve cache reliability in conjunction with *Error Correction Codes* (ECCs) [17]. Scrubbing involves reading values from cache memory, correcting any single-bit errors, and writing the bits back to cache memory. While scrubbing has proven to be effective for very large memory systems, it is not recommended for L1 and L2 caches as it could interfere with processor accesses and reduce the effective L1/L2 bandwidth. Moreover, scrubbing requires dedicated hardware, which could significantly increase design complexity and system cost [16].

As it is difficult to provide guaranteed reliability for caches, an alternative approach is disabling the cache in safety–critical applications [5]. By disabling the cache, the area susceptible to SEUs is drastically reduced, increasing processors reliability considerably. This, however, can come with significant performance loss which may not be tolerable for many applications. It is due to limitations listed above that designing a reliable cache memory continues to serve as a serious challenge for microprocessor designers.

3.2. Previous vulnerability factor modeling techniques

Many previously proposed cache reliability estimation methods rely on *Fault Injection* (FI) strategies [5–8,18,19]. When using an FI strategy, a limited number of memory addresses are targeted. Several workloads are then executed to measure the number of detected failures. Consequently, FI studies are both time-consuming (due to the large number of runs), and prone to inaccuracy (due to the limited number of addresses targeted).

Li et al. introduced SoftArch as a model (and a tool) to enable soft error analysis at the architecture level [20]. SoftArch uses a probabilistic error generation and propagation process model in the processor. This tool, however, does not consider device or circuit-level details and does not support application-level masking. Somani et al. presented a cache error propagation modeling technique [21]. The proposed model uses software fault injection to determine the cache vulnerability to soft errors. Kim et al. used the same model to measure data cache access reliability [16].

Mukherjee et al. [14,22] introduced *Architectural Vulnerability Factor* (AVF) to analyze and quantify the architectural masking of soft errors in different processor structures using the processor performance model. The AVF of a structure is the likelihood of a failure occurring due to a raw error event in the structure [14,22]. To measure the AVF of a structure, the bits that affect the final program outcome are identified on a cycle-by-cycle basis. These bits are referred to as *Architecturally Correct Execution* (or simply ACE) bits. All other bits are termed un-ACE. Examples of un-ACE bits are the operand bits of an NOP instruction or opcode bits in a killed instruction. All bits are assumed as ACE bits unless proven un-ACE.

Biswas et al. extended the AVF model to cover caches and other address-based structures [10]. Their proposed model extends AVF measurement to data and tag arrays but does not cover status bits. The model determines the vulnerability factor of a cache based on the ACE lifetime of cache words. They performed several experiments on various data cache configurations. Accordingly, they suggested a flushing technique to enhance reliability.

Asadi et al. introduced a critical time model to estimate the reliability of an unprotected or partially protected cache [11]. The proposed model computes cache vulnerability using the residency time of *Critical Words* (CW) in the cache. A CW is defined as a cache word that is guaranteed to propagate to other locations in the memory system or to the CPU. Using the proposed model, Asadi et al. developed a simulation model and measured the reliability of L1 caches.

Wang et al. introduced *Temporal Vulnerability Factor* (TVF) as a soft error characterization model [23] to capture the upper bound of the cache vulnerability factor. Their proposed model extends the work presented in [11] by calculating the critical times at various granularity levels, e.g., cache line, word, or byte.

More recently, Tang et al. extends the proposed model in [23] for the private L1 data cache in the context of *Chip-MultiProcessors* (CMPs) [24]. This model includes vulnerable cache lines in the MESI coherence protocol and then characterizes the L1 data cache vulnerability and proposed early cache line invalidation technique to reduce the total vulnerability factor.

Online measurement of the AVF at run-time is extremely expensive and requires specialized hardware and dedicated computation power. Therefore, Sridharan et al. classified the AVF into *Program Vulnerability Factor* (PVF) [25] and *Hardware Vulnerability Factor* (HVF) [26] to speed up AVF computation at run-time. It allows software developers to influence the run-time AVF estimates by providing architecture-independent PVF. Duan et al. further extend online AVF measurement for the multithreaded applications [27]. They proposed a two-level prediction scheme which predicts PVF with contention-free assumption at first level and then incorporates threading contention in the second prediction level.

Li et al. studied the limitations of AVF modeling [28] and showed that AVF estimations can result in large discrepancies where the raw error rates of individual components are very large. As an example, they showed that in space applications the calculated vulnerability factor using the AVF technique is twice greater than the actual vulnerability factor.

Wang et al. reported that SFI at *Register Transfer Level* (RTL) can generate more accurate AVFs compared to ACE analysis [29]. In response, Biswas et al. demonstrated that ACE analysis accuracy can be improved up to 40% by adding more detail to the processor performance model [30].

There have been also numerous circuit-level sensitivity analysis techniques investigating circuit-level masking factors such as electrical, logical, and timing derating factors [1]. These techniques mainly examine the effect of *Single Event Transients* (SETs) or *Multiple Event Transients* (METs) as possible sources of failures that can occur in combinatorial logic. SETs/METs occurring in combinational logic can propagate into sequential elements, but they still may be masked similar to SEUs. The main shortcoming of circuit-level SER estimation techniques is that they are not scalable to large circuits since their execution time becomes intractable for circuits containing millions of gates and FFs. To alleviate this shortcoming, Costenaro et al. have introduced a method to estimate the SET sensitivity of complex combinational logic at the RTL level [31,32]. This is achieved by characterizing SET sensitivity at cell libraries and integrating them into coarse-grained block level models to speed up SET propagation in very large circuits. Further discussion of circuit-level SER estimation techniques is beyond the scope of this work.

4. Motivation and limitations of previous analytical models

Previous vulnerability analytical models mainly focus on computing vulnerability at the component-level. Such studies often define the vulnerability factor as the percentage of errors occurring within a component that propagate to the outputs of the component [10,11,14,23]. These studies assume that any error occurring within the component and propagating to the component outputs leads to a system failure. As we show in this work, many errors could potentially be masked by different processor components.

To provide better understanding, in Fig. 1(a) we present an example using L1 data cache. We assume that the target byte within the *L1 data cache* (*DL*1) is struck by an SEU changing one of its bits. The target byte is then sent to the ALU unit. We also assume that the byte is used as an operand in a Logical AND arithmetic operation. This example shows that even though the incorrect value of the byte is transferred from the DL1 to the inputs of the ALU unit, it does not propagate from the ALU inputs to the outputs of the ALU unit.

To elaborate this in more detail, as illustrated in Fig. 1(a), we assume that the target byte value is initially equal to $0 \times FA$ (denoted in hexadecimal format) and that an SEU event inverts the most significant bit of this byte, changing the byte from $0 \times FA$ to $0 \times 7A$. If this byte is used as an operand of a logical AND operation with a second operand equal to 0×32, the erroneous value would not propagate to the outputs of the ALU unit. This is because the logical AND masks the error bit when producing the final outcome, i.e., $(0 \times FA \wedge 0 \times 32) = (0 \times 7A \wedge 0 \times 32) = 0 \times 32$.

As another example illustrated in Fig. 1(b), assume that the target tag address is $0 \times 2B$, and an SEU event inverts the 6th bit of this byte, changing the tag address from $0 \times 2B$ to $0 \times 0B$. This would cause a wrong data ($0 \times 8E$) to be selected as an operand of logical AND operation with a second operand equal to 0×32. Please note that here the erroneous data would not propagate to the output of the ALU unit since the logical AND masks the erroneous data when producing the final outcome, i.e., $(0 \times CE \wedge 0 \times 32) = (0 \times 8E \wedge 0 \times 32) = 0 \times 02$. An alternative masking scenario is when a fault occurs in the cache but the erroneous value is never used or the fault changes an unused address of the cache. Finally, another instance of masking scenario is when a fault changes a valid data block in the cache, but the block is overwritten and the erroneous value gets vanished.

Error masking can occur in both data- and control-path components. In this work, we investigate the impact of error masking in data-path components on the overall system-level vulnerability. Analysis of error masking in the control logic is beyond the scope of this work.

5. The CEDAR model: Computing SVF using input-to-output derating factor

The main objective in CEDAR is to accurately compute *Component-level Vulnerability Factor* (CVF) and *System-level Vulnerability Factor* (SVF) of data-path components in a high-performance processor. CVF is defined as the percentage of errors occurring within a component and propagating to *other components* of the system whereas SVF is defined as the portion of errors occurring within a component that propagate to the *system outputs* and cause a system failure.

In order to accurately compute CVF and SVF, we define the IOD of a component as the probability of propagating an erroneous value from the component inputs to the component outputs. As an example, if 85% of erroneous input values are masked when

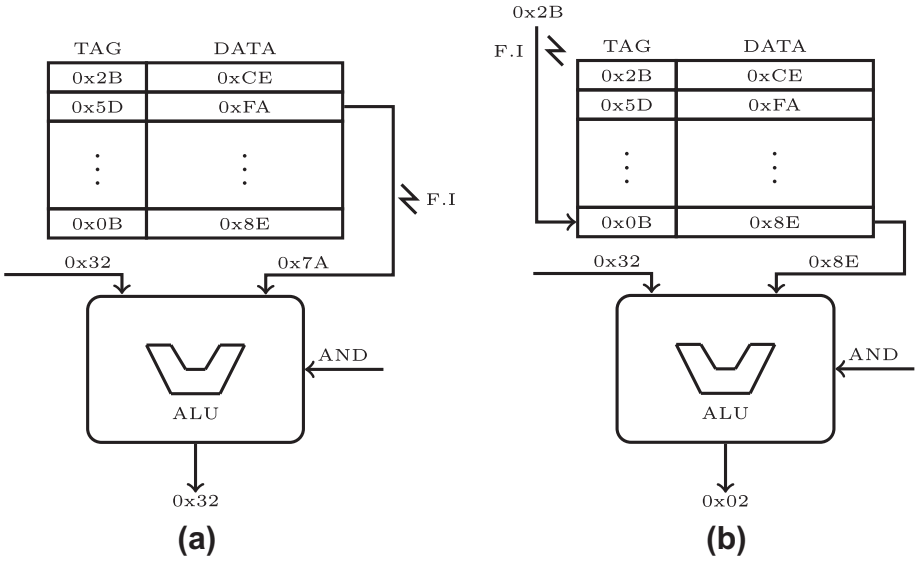

Fig. 1. Examples of error masking in a CPU.

propagating to the components outputs, the IOD factor would equal to 0.15. We refer to the complement of IOD as *Input-to-Output Masking* (IOM). Hence, IOM is defined as the percentage of errors being masked when propagating erroneous values from the component inputs to the component outputs (i.e., $IOM = 1 - IOD$).

Fig. 2 illustrates an example of how SVF of data-path components of a processor can be computed. According to the CVF definition, CVF_{IS} can be obtained by computing the portion of errors occurring within *Instruction Scheduler (IS)* and propagating to the inputs of either the *Integer Unit (IU)* or the *Floating Point (FP Unit)*. Therefore, considering running integer benchmarks, the SVF of IS (SVF_{IS}) can be computed as $CVF_{IS} \times IOD_{IU}$. Using the same assumption, the SVF of *Instruction Front End* (SVF_{IFE}) can be computed as $CVF_{IFE} \times IOD_{IS} \times IOD_{IU}$. This expression means that an erroneous value within IFE will propagate to the CPU outputs if it is neither masked by IS nor IU.

It is notable to mention that here we have assumed errors propagating to the L1 data cache will cause a data integrity issue and result in a system failure. In order to further increase the accuracy of the computed SVFs, we should take into account the input-to-output masking factor of the L1 data cache ($DL1$). As an example, SVF_{IFE} can be rewritten as $CVF_{IFE} \times IOD_{IS} \times IOD_{IU} \times IOD_{DL1}$. In this analysis, we ignore IOD_{DL1} since we believe the masking factor of the L1 data cache is negligible, i.e., erroneous values propagating to the DL1 cache are very unlikely to be overwritten by the CPU core.

6. Case study: Using CEDAR model to compute SVF and CVF of cache memory

The CVF and the SVF of data-path components can be accurately measured using IOD factor. The proposed technique can be used for computing the vulnerability of any data-path component. However, for the sake of clarity, we apply the CEDAR model to measure the vulnerability of cache memory in this section. In the next subsections, we first elaborate why the vulnerability of cache memory in the presence of ECC codes is still an issue. Then, we present a simple example to elaborate how the CEDAR model can be used to compute the SVF of a cache block. Finally, general cases and the corresponding formulations to compute the SVF of cache blocks are presented next.

6.1. Motivation for this case study

One may argue that with ECC protection schemes, computing vulnerability of cache memory to soft errors is meaningless.

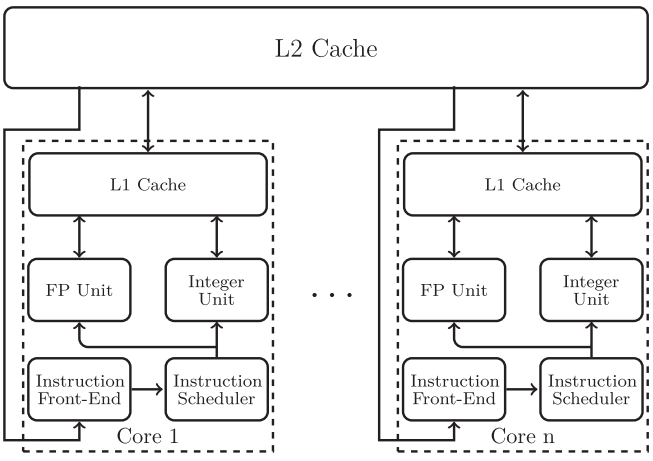

Fig. 2. Components of a typical multi-core processor.

Note ECC schemes can effectively detect and correct SEUs and *Single Event Multiple Upsets* (SEMUs) in the main memory [33] but such schemes in cache memory have less efficiency in the nanometer era as compared to the main memory. Unlike the main memory where the bits of a memory word are interleaved across multiple memory banks, the bits of a cache block are physically located nearby in the device layout. A SEMU in the main memory is translated to multiple single errors in multiple banks, which can be effectively detected and corrected by ECC codes such as *Single Error Correction-Double Error Detection* (SEC-DED).

Employing ECC and interleaving in cache memories, however, comes with significant power and performance penalties. While SEC-DED is commonly employed in L2 and L3 caches in high-end processors such as Itanium 9500 [34,35] or Power 7 [36], it is rarely used in L1 caches due to the significant performance penalty imposed to the system. Protecting DL1 cache with SEC-DED scheme increases cache access latency, which may require one or more extra clock cycles to access a cache block. This, in turn, can significantly affect the overall system performance.

In enterprise applications, even ECC schemes such as SEC-DED are not accountable for high rate of SEMUs in recent nanoscale technologies (40 nm and beyond). It has been demonstrated in [33] that the rate of SEMUs simultaneously affecting three cells or more is getting more pronounced in 40 nm technology and beyond. Thus, employing SEC-DED in a cache memory is not able to detect or correct SEMUs affecting more than two bits in a cache memory not equipped with interleaving. As an example, a particle strike affecting four adjacent cells cannot be detected by the SEC-DED scheme. Employing more aggressive protection techniques such as *Double Error Correction-Triple Error Detection* (DEC-TED) will impose significant power and performance overhead to the cache memory. According to numerous studies [37,38], interleaving is not employed in cache memories since interleaved cache memory imposes significant power penalty. Therefore, there is still a high probability of soft error susceptibility in cache memory even in the presence of ECC codes.

To summarize, ECC schemes can be employed in L1 caches with significant performance penalty. Such schemes, however, are still susceptible to SEMUs in 45 nm nodes and beyond. Other error correction schemes such as two-dimensional ECC (and Parity) [39] can be employed to achieve higher level of dependability [39–42]. Such protection schemes, however, typically come with significant cost and limited applicability [37–39,43]. Therefore, with accurate VF modeling using CEDAR, microprocessor designers could accurately estimate the FIT rate of an entire processor in design cycle and choose appropriate error detection or correction schemes to offer a cost-effective fault-tolerant design [14]. As a future work, one can investigate the effect of the IOD factor in other processor structures such as register file.

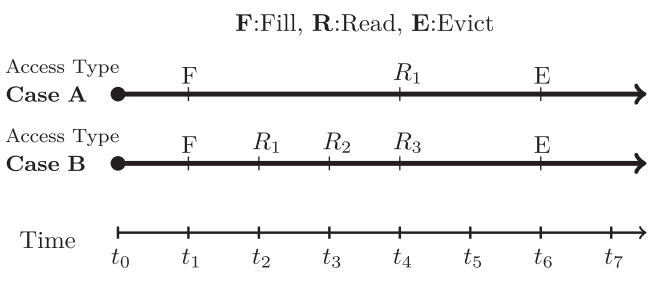

Fig. 3. An example with two different access patterns to demonstrate the limitation of previous methods to compute vulnerability of cache memory.

6.2. SVF of a cache block: Basic concept

The concept of critical byte or critical word presented in [11] is used in our proposed vulnerability modeling technique. A *Critical Word* (CW) or *Critical Byte* (CB) refers to a byte or a word of cache that definitely propagates an erroneous data from cache memory to other data-path components. *Critical Time* (CT) is the residence time of a CW or a CB in the cache. In this work, we extend this concept to more accurately estimate the impact of cache vulnerability factor on the overall system reliability.

In the CEDAR model, we have defined *System-level Vulnerability* (SV) factor that is the sum of time periods in which an error occurring in a byte of cache in these times produces a system failure. Using IOD and SV definitions, the SV of the bytes in Case *A* and Case *B*, illustrated in Fig. 3, can be computed using Eqs. (4) and (5), respectively. SV_{caseA} is the propagation probability of an error occurring in time interval $(t_4 - t_1)$ to the system output by the first read access, i.e., R1.

$$SV_{caseA} = (t_4 - t_1) \cdot IOD_{cpu} \tag{4}$$

$$\begin{aligned} SV_{caseB} = {} & (t_2 - t_1) \cdot IOD_{cpu} + (t_3 - t_2) \cdot IOD_{cpu} + (t_4 - t_3) \\ & \cdot IOD_{cpu} + (t_2 - t_1) \cdot IOM_{cpu} \cdot IOD_{cpu} + (t_3 - t_2) \\ & \cdot IOM_{cpu} \cdot IOD_{cpu} + (t_2 - t_1) \cdot IOM_{cpu}^2 \cdot IOD_{cpu} \end{aligned} \tag{5}$$

The first three terms in Eq. (5) are the probability that an error occurring in time intervals $(t1, t2), (t2, t3)$, and $(t3, t4)$ propagates to the system output via R1, R2, and R3 read accesses, respectively. The next two terms are also the probability that an error occurring in time interval $(t1, t2)/(t2, t3)$ is masked by the processor after the first read, i.e., R1/R2 but is propagated by the second read access, i.e., R2/R3. For example, the term $(t_3 - t_2) \times IOM_{cpu} \times IOD_{cpu}$ expresses the probability that an error is masked after being read by R2 but is propagated to the processor output by R3. Finally, the last term, i.e., $(t_2 - t_1) \times IOM_{cpu}^2 \times IOD_{cpu}$ is the probability that an error is masked after being read by both R1 and R2 but is propagated to the processor output by R3.

SV can be also represented by Eq. (6), where the second term, i.e., CV_{caseB} is referred as component vulnerability. In other words, the system-level vulnerability is the product of the *Component Vulnerability* (CV) and the probability of not being masked by the CPU. CV_{caseB} in example shown in Fig. 3 is computed using Eq. (7).

$$SV_{caseB} = IOD_{cpu} \cdot CV_{caseB} \tag{6}$$

$$\begin{aligned} CV_{caseB} = {} & (t_2 - t_1) + (t_3 - t_2) + (t_4 - t_3) + (t_2 - t_1) \cdot IOM_{cpu} \\ & + (t_3 - t_2) \cdot IOM_{cpu} + (t_2 - t_1) \cdot IOM_{cpu}^2 \end{aligned} \tag{7}$$

It should be noted that when a byte within a dirty block is written back to the main memory, the masking factor of the CPU would not have any impact on the byte vulnerability. In fact, in this case, the erroneous value that is written back to the main memory can eventually result in a program failure. Therefore, *SV* would be equal to *CV* in this case.

For the sake of clarity, the SVF and CVF for Case A and Case B have been reported for different *IOD* values in Fig. 4. In addition, we have reported AVF to provide a comparison. In this figure, we have assumed that $t0 = 0$, $t_1 = 1$, $t_2 = 2$, $t_3 = 3$, $t_4 = 4$, $t_5 = 5$, and $t_6 = 6$. Two observations can be concluded from the simple example given in Fig. 4. First, vulnerability of a cache word is highly dependent on the IOD of CPU. Second, vulnerabilities computed by previous modeling techniques such as AVF [10] and CT [11] can be inaccurate for smaller values of IODs (or larger values of IOMs).

Note there are cases in which errors propagating to the main memory can be masked at the application level. In this work, however, we have assumed that an erroneous data propagating to higher memories levels (L2 cache or main memory) would result in a failure. This is one of the main limitations of the CEDAR model which does not account for application-level masking.

6.3. General cases to compute SVF of a cache block

In general, there are different SV formulations for different cache access patterns. Here, we compute SV for different possible scenarios. We explain five possible scenarios in detail as follows:

- *Case 1: A Clean Block with Multiple Read Accesses*
  As mentioned, in the first scenario, the accessed block does not change during the critical time. In other words, we have only read accesses during the critical time of the block and the block eviction occurs before the cool-down phase (Case 1 in Fig. 5). Eqs. (8)–(10) show the component vulnerability factors of case 1 for the *R*1, *R*2, and *R*3 read accesses, respectively. $CV(F : R_1)$ measures the CV of the byte from the fill time to the first read access, i.e., *R*1. Similarly, $CV(F : R_2)$ and $CV(F : R_3)$ are the CV of the byte from the fill time to the second and third read accesses. It should be noted that in Eqs. (8)–(10), we have assumed that $t_{Fill} = t0 = 0$.

$$CV(F : R_1) = t_1 \tag{8}$$

$$\begin{aligned} CV(F : R_2) &= t_1 + (t_2 - t_1) + t_1 \cdot IOM_{cpu} = t_2 + t_1 \cdot IOM_{cpu} \\ &= t_2 + CV(F : R_1) \cdot IOM_{cpu} \end{aligned} \tag{9}$$

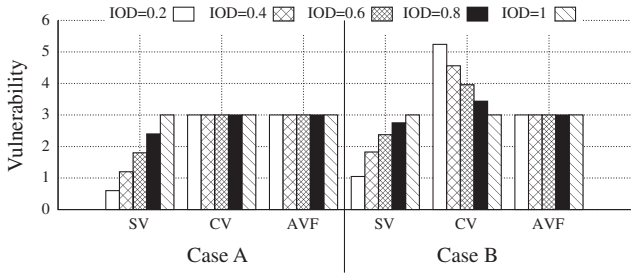

Fig. 4. Comparison SV and AVF for the example given in Fig. 3.

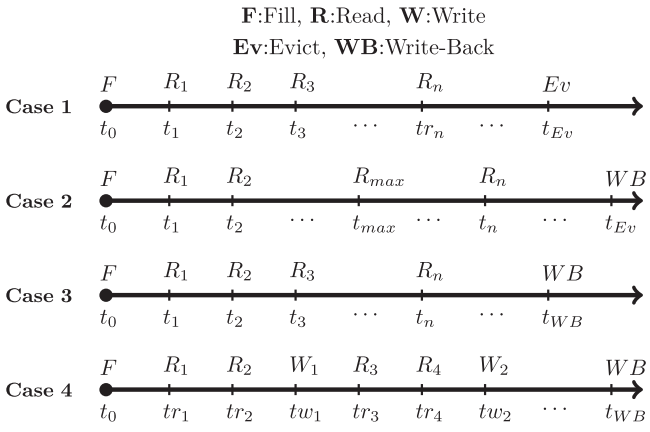

Fig. 5. Read/write patterns to compute vulnerability.

$$CV(F:R_3) = t_1 + (t_2 - t_1) + (t_3 - t_2) + t_1 \cdot IOM_{cpu} + t_1 \cdot IOM_{cpu}^2 + (t_2 - t_1) \cdot IOM_{cpu} = t_3 + t_1 \cdot IOM_{cpu} + t_1 \cdot IOM_{cpu}^2 + (t_2 - t_1) \cdot IOM_{cpu} = t_3 + CV(F:R_2) \cdot IOM_{cpu} \quad (10)$$

In general, Eq. (11) computes the CV of the byte from the fill time to the last read access for n consecutive read accesses.

$$CV(F:R_n) = tn + CV(F:R_{n-1}) \cdot IOM_{cpu} \quad (11)$$

Once $CV(F:R_n)$ is computed, we can compute SV using Eq. (12).

$$SV(F:R_n) = IOD_{cpu}.CV(F:R_n) \quad (12)$$

In case that the block eviction occurs in the cool-down phase, for all read accesses before the start of cool-down phase, SVs and CVs are computed using Eqs. (11) and (12). The cool-down read accesses are not considered in our computations.

- *Case 2: A Clean Block with Numerous Read Accesses*
  The main shortcoming of the equations presented in case 1 is that the computation can become time-consuming for larger number of read accesses (e.g., for $n > 100$). On the other hand, the difference between $SV(F:R_n)$ and $SV(F:R_{n-1})$ becomes negligible for n greater than 20 (especially for $IOD > 0.4$). This has been shown in Fig. 6. To this end, we set an upper bound for the number of read accesses in case that a block remains clean before cool-down phase. We refer to the maximum number of read accesses used in CV computations as R_{max}. Based on this assumption, we use Eq. (13) to compute CV and then use Eq. (12) to calculate SV.

$$CV(F:R_n) = CV(F:R_{max}) + t_n - t_{max} \quad (13)$$

If R_{max} and the block eviction occurs within the cool-down phase, we use Eq. (14) to calculate CV. Note that t_n and t_{max} have been shown in Fig. 5.

$$CV(F:R_n) = CV(F:R_{max}) \quad (14)$$

- *Case 3: A Dirty Block with no Write Operation on the Target Byte*
  In this scenario, the target byte (i.e., the byte whose vulnerability is under study here) is accessed just by read operation, but the other bytes in the corresponding block (the block that the target byte belongs to) are accessed by both read and write operations. In other words, the block is dirty while the target byte is clean. Since the block will be written back to the higher memory hierarchy, all bytes of the block are vulnerable from the filling time to the eviction time. Consequently, the SV and CV of the target byte in this case are computed by Eqs. (15) and (16), respectively.

$$CV(F:R_n) = t_{evict} \quad (15)$$

$$SV(F:R_n) = t_{evict} \quad (16)$$

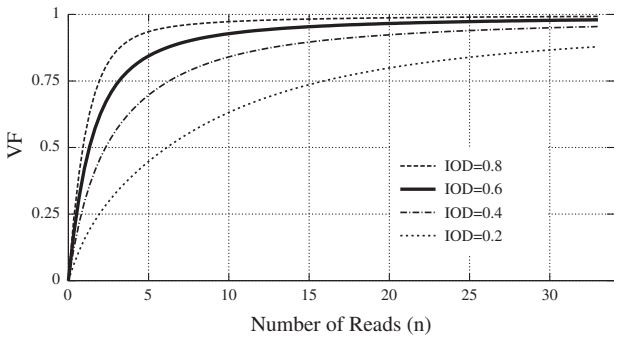

Fig. 6. Impact of the number of reads on the vulnerability factor.

- *Case 4: Dirty Block with Write Operation on the Target Byte*
  In contrary to the previous scenario, here the target byte is accessed by both read and write operations. This is shown as Case 4 in Fig. 5. Assume that there are n read and k write operations on the target byte. While tw_j refers to the jth write operation, tr_i refers to the ith read operation. In this scenario, we compute SV and CV of the byte between all two consecutive write operations, i.e., tw_{j-1} to tw_j, using Eqs. (11) and (12), respectively. The CV and SV of two consecutive write operations are shown as $CV(tw_{j-1}, tw_j)$ and $SV(tw_{j-1}, tw_j)$, respectively. Finally, the overall SV and CV are computed by Eqs. (17) and (18), respectively. As the block is dirty and eventually will be written back to the main memory, the time interval between the last write operation and the time of eviction is added to the these two equations.

$$CV = CV(tw_1) + CV(tw_2) + \cdots + CV(tw_k) + (t_{evict} - tw_k) \quad (17)$$

$$SV = SV(tw_1) + SV(tw_2) + \cdots + SV(tw_k) + (t_{evict} - tw_k) \quad (18)$$

- *Case 5: Other Situations*
  We use conventional life time analysis proposed in [11,23] for computing vulnerability in other situations. The lifetime model distinguishes among nine lifetime phases for each byte according to the previous activity and the current status, and further categorizes them into two groups, vulnerable and non-vulnerable phases. Five lifetime phases, WRP (i.e., time between the first read and the last read after a write), RR (i.e., time between the first read and the last read of a clean data item), WR (i.e., time between the last write and the first read), WPL (i.e., time between the last write and the replacement without any read in between) and WRPL (i.e., time between the last read and the replacement of a dirty data item) are vulnerable and we will consider their vulnerability time. The other lifetime phases, RPL (i.e., time between the last read and the replacement of a clean data item), Invalid (i.e., time in the invalid state), RW (i.e., time between the last read and the first write), and WW (i.e., time between the first write and the last write without any read in between) are non-vulnerable. Note that since we propose fine grained (per byte) life-time analysis on data items, RW and WW are unconditionally non-vulnerable.

6.4. Overall cache SV/CV calculation

Once the SVs of all bytes are calculated, we can compute the CV, SV, CVF, and SVF of the cache memory as follows:

$$CV_{Cache} = \sum_{i=1}^{N} CV_i \quad (19)$$

$$SV_{Cache} = \sum_{i=1}^{N} SV_i \quad (20)$$

$$CVF_{Cache} = \frac{CV_{Cache}}{TT \times M} \quad (21)$$

$$SVF_{Cache} = \frac{SV_{Cache}}{TT \times M} \quad (22)$$

In these equations, TT is the total execution time, M is the size of the cache in bytes, and N is the number of bytes for which CV and SV have been calculated.

7. Algorithm

Algorithm 1. CVF and SVF computation

```
1  b_i: Byte i^th in a cache block
2  B_j: Block j^th in a set
3  CV_{b_i}: Component-level Vulnerability of Byte b_i
4  SV_{b_i}: System-level Vulnerability of Byte b_i
5  CV_{b_i,lw}: CV of Byte b_i from Last Write Operation to now
6  SV_{b_i,lw}: SV of Byte b_i from Last Write Operation to now
7  TSC: Total Simulation Cycles
8  NCB: Number of Bytes within Cache Memory
9  begin
10   if WRITE HIT then
11     CV_{b_i,lw} = SV_{b_i,lw}=0
12   end
13   if READ HIT then
14     if B_j not dirty then
15       CV_{b_i} = (now - FillTime) + CV_{b_i} * IOM_cpu
16       SV_{b_i} = CV_{b_i} * IOD_cpu
17     end
18     else if b_i dirty then
19       CV_{b_i} = CV_{b_i} − CV_{b_i,lw}
20       SV_{b_i} = SV_{b_i} − SV_{b_i,lw}
21       CV_{b_i,lw} = (now - LastWriteTime) +
         CV_{b_i,lw} × IOM_cpu
22       SV_{b_i,lw} = CV_{b_i,lw} * IOD_cpu
23       CV_{b_i} = CV_{b_i} + CV_{b_i,lw}
24       SV_{b_i} = SV_{b_i} + SV_{b_i,lw}
25     end
26   end
27   if Evict OR Flush B_j (WriteBack) then
28     if B_j is dirty then
29       for each b_i in B_j do
30         if b_i is dirty then
31           CV_{b_i} = CV_{b_i} + now - LastWriteTime[b_i]
32           SV_{b_i} = SV_{b_i} + now - LastWriteTime[b_i]
33         end
34         else
35           CV_{b_i} = (now - FillTime[b_i])
36           SV_{b_i} = (now - FillTime[b_i])
37         end
38       end
39     end
40   end
41   if Evict OR Flush (WriteThru) then
42     No action needed
43   end
44   if End of simulation then
45     Start cool-down process
46     CV_cache = add all CV_{b_i}'s
47     SV_cache = add all SV_{b_i}'s
48     CVF_cache = CV_cache / (TSC × NCB)
49     SVF_cache = SV_cache / (TSC × NCB)
50   end
51 end
```

Algorithm 1 shows how CV and SV of target bytes are computed. In particular, in lines 10 through 12 as well as lines 18 through 25, CV and SV of a dirty byte within a dirty block are calculated. We use $CV_{b_i,lw}$ and $SV_{b_i,lw}$ to compute $CV(tw_j)$ and $SV(tw_j)$ based on Eqs. (17) and (18), respectively. In lines 31 and 32, the vulnerability of target byte between the last write operation and eviction time is added to the overall vulnerability based on Eqs. (17) and (18). CV and SV of a clean byte within a clean block are computed in lines 14 through 17 according to Eqs. (11) and (12). CV and SV of a clean byte within a dirty block are computed in lines 34 through 37 according to Eqs. (15) and (16). Finally, in lines 44 through 50, CVs and SVs of data blocks are summed up to calculate the overall CV and SV of cache memory according to Eqs. (19) and (20). In addition, we calculate CVF and SVF using Eqs. (21) and (22), respectively. Algorithm 1 can be used to compute CV and SV of both WB and WT caches.

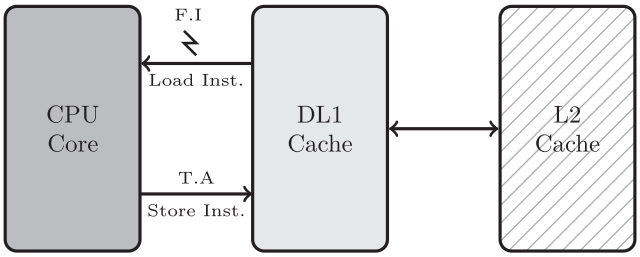

Fig. 7. Estimating *IOD* of the CPU core.

Table 1
Default configuration parameters used in our simulations.

Configuration parameter	Value
Processor	
Functional units	4 integer ALUs, 4 integer multiplier/divider 1 FP ALUs, 1 FP multiplier/divider
LSQ Size/RUU Size	32 Instructions/32 Instructions
Fetch/Slot/Map/Issue/ Commit Width	4/4/4/4/11 instructions/cycle
Integer/FP issue queue size	20/15 instructions
Reorder buffer size	80 instructions
Register file	40 FP/40 Integer entry
Return address stack	32-entry
Victim buffer	8 entries, 1-cycle hit latency
MSHR entries	8/cache
Prefetch MSHR	entries 2/cache
Cycle time	1 ns
TLB and Cache memory hierarchy	
TLB	128-entry ITLB/128-entry DTLB, fully-associative
L1 instruction cache	64 KB, 2-way, 64 byte lines
(IL1)	1 cycle latency
L1 data cache	64 KB, 4-way, 64 byte lines
(DL1)	3 cycle latency
L2	2 MB unified, direct-mapped 64 byte lines, 7 cycle latency
Memory	100 cycle latency
Branch logic	
Predictor	Hybrid, 4 K global two-level 1 KB local, 4 K choice
Branch miss-prediction penalty	7 cycles
BTB	512 entry, 4-way
Mis-prediction penalty	7 cycles

8. Experimental setup

For our experimental system setup, we have used the *sim-alpha*, a cycle accurate *Alpha 21264* processor simulator [44]. In order to implement CEDAR, we have modified the source code of the sim-alpha. We have extended sim-alpha to include WT caches. In addition, for the sake of comparison, we have employed AVF estimation method implemented in *sim-soda* [45] in our simulation framework.

We have used *SPEC2K* benchmark suite [46] compiled for Alpha ISA [47] (with different inputs for some benchmarks) as our workload. We run 100 M instructions and 100 M cool-down for SVF, CVF and AVF estimation as well as SFI experiments. When running the workload, we have skipped the first 100 M instructions. The default system parameters including cache size and associativity are reported in Table 1 in detail. We extend the simulator to perform two types of FI experiments as well as VF estimations. Further detail of FI experiments is described in the following sections.

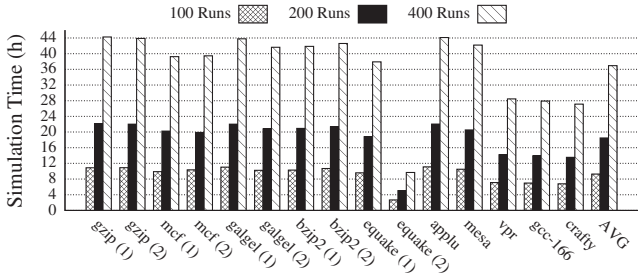

Fig. 8. Fault injection simulation time to estimate the IOD factor of the CPU core.

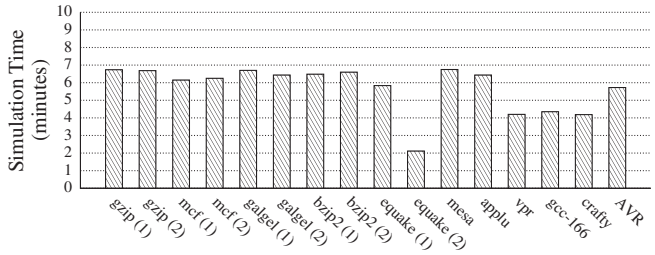

Fig. 9. SVF simulation time to estimate the vulnerability factor of the CPU core.

8.1. IOD experimental setup

In order to measure the IOD of a processor core, we have performed fault injection experiments. In our fault injection experiments, a random load instruction is selected in each iteration and a random bit of the instruction operand is flipped. Afterwards, we trace the program to investigate if the injected error propagates to the CPU outputs. As illustrated in Fig. 7, in our fault injection experiments, faults are injected in load instruction operand (i.e., when a data block is fetched from D-cache and transferred to the CPU core) and then store instructions (when data is copied from the CPU core to the D-cache) are observed to investigate if the error affects the output of the CPU core. For each application, we have performed 100 fault injections and report the percentage of faults being masked by the processor core. For instance, if 20 out of 100 fault injection experiments do not propagate to the CPU core outputs, IOM_{cpu} would be $\frac{20}{100} = 0.20$, (i.e., $IOD = 0.8$). We have also conducted 200 and 400 fault injections and discussed about the accuracy of the experiment with 100 fault injections in Sections 9 and 9.1.

In our fault injection experiments, and in the interest of keeping simulation time low, we have used functional simulation mode of sim-alpha. Previous studies show that using functional simulation mode runs much faster than circuit-level or RTL fault injection approaches [5–8,48] while having negligible impact on accuracy. Note in this work, we do not have any contribution in running the functional simulation or other soft error injection techniques. We have used an existing high-level fault injection model to estimate the IOD of a single component in the processor data-path (e.g., Load/Store queue). Our proposed model can be employed at any abstraction level. To have a fair comparison between the proposed model and the reference model, we have implemented both models in the same abstraction level. In this work, we use a high-level fault injection method so that it can be used at early design stages where the micro-architecture low-level models (RTL, Flip-Flop) are not available yet. Note that relatively accurate estimations at early design stages can help designers improve system efficiency with minimum cost while reducing the total design turnaround time.

The run-time for IOD_{cpu} estimation of some programs for the configuration presented in Table 1 is reported in Fig. 8. We have illustrated different inputs of each benchmark within parenthesis. For instance, *gzip(1)* means *gzip* benchmark with input number 1.

In Fig. 9, we also report the simulation time to compute SVF for different application programs. Note unlike IOD or SFI experiments, the SVF computation is executed only once and its execution time is negligible as compared to SFI or IOD execution time. As shown in this figure, the time taken to compute SVF is less than 10 min which is orders of magnitude smaller than the SFI computation time.

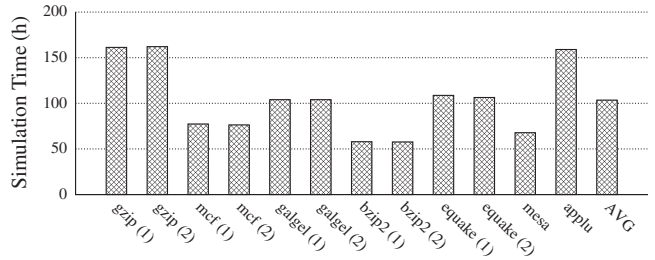

Fig. 10. Simulation time to estimate the vulnerability factor of the CPU core for 1000 experiments using the SFI method.

8.2. SFI experimental setup

In order to verify the results of the proposed SVF estimations, SFI experiments have been performed. In SFI experiments, in contrary to our approach in the IOD section, in each experiment at the beginning, a random time of simulation is determined, i.e., a random instruction number is selected. Then, a random valid address of D-cache is selected. When the specified time reaches, a random bit of a cache data block at the specified address is flipped. Afterwards, we trace the program to investigate if the injected error propagates to the CPU core output. In our fault injection experiments, faults are injected into data cache and then store instructions are observed to investigate if the error affects the output of the CPU core. For each application, we have performed 1000 fault injections and report the percentage of faults propagated to the output of the CPU core.

In order to examine the propagation of an injected fault to the output of the CPU core, we run several experiments without injecting any faults into D-cache. We refer to such experiments with no error injection as golden runs. A challenging issue is when the results of gold runs are not identical for each workload. Such phenomenon will occur when a workload is not deterministic. For this reason, only workloads with deterministic behavior have been included in the experiments. To increase the number of benchmarks, however, we have experimented deterministic workloads with various inputs. Moreover, unlike *IOD* experiments, we have used the timing simulation mode of *sim-alpha*. The run-time for the SFI estimation of experimented workloads have been reported in Fig. 10.

It is notable to mention that three sets of simulation times have been reported in this section, i.e., IOD, SVF, and SFI simulation times. These three sets of simulation times have been reported in Figs. 8–10, respectively. SVF computation is performed only once and as reported in Fig. 9, the SVF simulation time is less than 10 min for different benchmark programs. On the other hand, IOD is computed for 100 runs as reported in Fig. 8. Finally, the SFI technique which has been used as a reference model has been simulated for 1000 runs. By comparing the results reported in Figs. 8 and 10, it can be seen that 1000 runs in SFI, on average, is approximately 10× greater than its 100 IOD runs, as expected.

9. Results

Simulation results are reported and discussed in this section. In Section 9.1, we explain the IOD values as well as the impact of the number of runs and processor configuration on IOD. In the subsequent subsections, we report SVF, CVF, and AVF for the WB and WT caches. In Section 9.4, we discuss the accuracy of the proposed technique. Finally, we verify the results of CEDAR estimation by using SFI experiments.

9.1. IOD experiments

In Fig. 11, we report IODs of the CPU for the benchmarks used in this work. Based on this figure, on average, 44% of single bit errors are not masked by the CPU. Among the studied benchmarks, *gcc-166* and *applu* with IODs of 0.23 and 0.77 have the minimum and maximum IOD, respectively. As depicted in Fig. 11, the IOD of each program has been extracted for different number of fault injection experiments. The IODs extracted using 100 fault injections, on average, differs from the IODs extracted using 400 FI runs by less than 5%, as reported in Fig. 11.

In Fig. 12, we report the impact of *Speculative Load* instructions (SL) and disabling *Speculative Update of Branch and Line Predictor* (SUBLP) on the processor core IOD. In this figure, both SL and SUBLP are enabled in the *Base* configuration. SL and SPUBL also show the configuration in which speculative load instruction and speculative branch update are disabled. In case of disabling SL, *galgel(2)* and *applu* have the maximum and minimum impact on the IOD with 8.5% and 0.94% changes. In case of disabling SUBLP, the maximum and minimum changes in IOD are 58.7% in *galgel(1)* and 1% in *gzip(1)* programs. On average, by disabling of either SL or SUBLP, IOD changes 2.4% and 6.2%, respectively.

We have done a similar investigation on the effect of *Load and Store Queue* (LSQ) and *Re-Order Buffer* (ROB) size on the processor core IODs. As shown in Fig. 13, on average, the IOD of the processor core changes 1.9% and 12.4% while doubling the size of load store and reorder buffer queues, respectively.

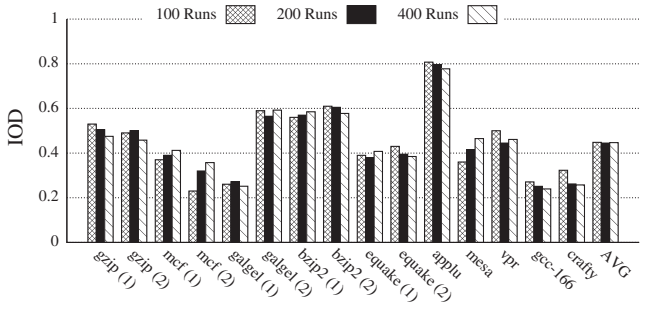

Fig. 11. IOD sensitivity to the total number of fault injections applied to load operations.

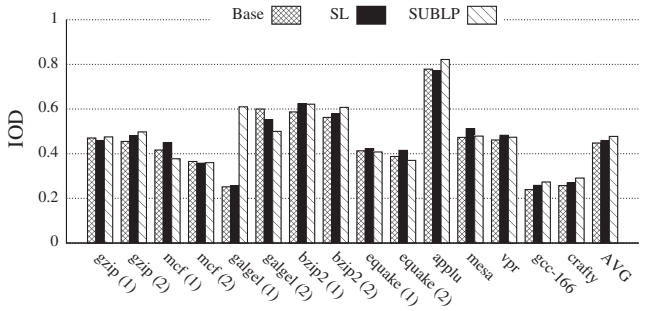

Fig. 12. IOD sensitivity to speculative load and speculative update of branch and line predictor.

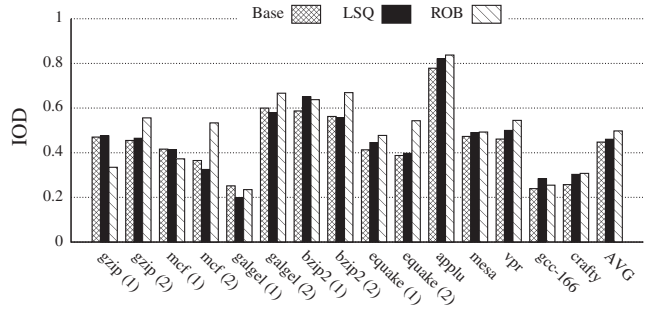

Fig. 13. IOD sensitivity to double size of LSQ and ROB.

We conclude from Figs. 12 and 13 that the processor core IOD has little sensitivity on the processor configuration while it is highly sensitive to the running application. In other words, IOD (or similarly IOM) is mainly workload dependent rather than configuration dependent.

9.2. Write-back cache analysis

Fig. 14 shows the SVF, CVF, and AVF of a DL1 cache for the WB cache configuration. As presented, there is little difference between these three VF models in WB configuration. This is mainly because in a WB cache, dirty blocks have to be written to a higher memory hierarchy. Therefore, an error occurring in such blocks could easily propagate to other components. In other words, the probability that an erroneous bit results in a system failure is very high in a WB cache. This probability is highly dependent on the percentage of dirty blocks. The higher the percentage of dirty blocks, the higher the probability of system failure, and as a result, the less difference between estimations in SVF, CVF, and AVF. Among the

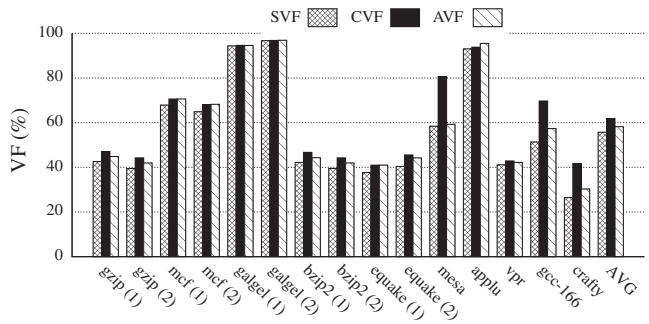

Fig. 14. Comparison of SVF, CVF, and AVF in a DL1 cache with write-back policy.

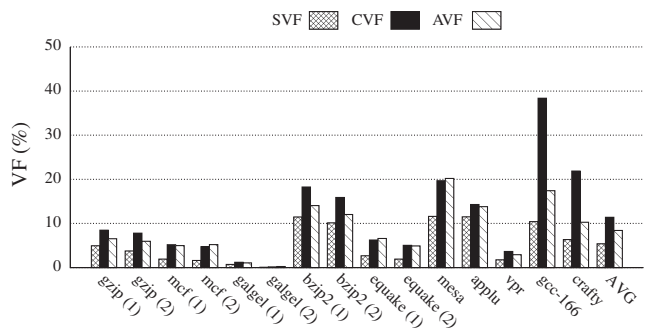

Fig. 15. Comparison of SVF, CVF, and AVF in a DL1 cache with write-through policy.

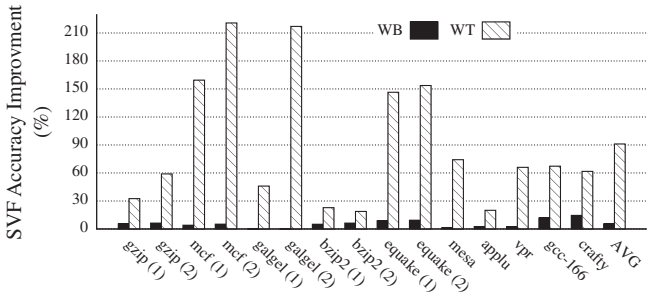

Fig. 16. Accuracy improvment of SVF model compared to AVF modeling technique for both write-back and write-through policies.

Table 2
Number of true, faulty, and failed runs and vulnerability factor in SFI experiments.

Benchmark	True runs	Failed runs	Faulty runs	VF_{FI}(%)
gzip(1)	934	0	66	6.6
gzip(2)	949	0	51	5.1
mcf(1)	984	1	15	1.6
mcf(2)	990	3	7	1
galgel(1)	995	0	5	0.5
galgel(2)	995	1	4	0.5
bzip2(1)	864	1	135	13.6
bzip2(2)	877	0	123	12.3
equake(1)	963	12	25	3.7
equake(2)	978	4	18	2.2
mesa	858	5	137	14.2
applu	864	0	136	13.6
Average	937.58	2.25	60.16	6.24

benchmark programs, *crafty* and *galgel(2)* show the maximum (14%) and minimum (0.17%) differences, respectively.

9.3. Write-through cache analysis

In Fig. 15, we show the SVF, CVF, and AVF of the DL1 cache for the WT configuration. Compared to the WB cache, the cache is less vulnerable in the WT configuration as the critical time of data blocks is shorter. In addition, the difference between estimations are higher than the difference in the WB cache since there is no dirty block in a WT cache.

9.4. Accuracy improvement

Fig. 16 shows the amount of accuracy improvement by CEDAR compared to AVF for both WT and WB caches. The results show an accuracy improvement of about 91% for the WT cache while having an accuracy improvement of about 5% for the WB cache. As presented, the maximum accuracy improvement is achieved in *mcf(2)* program in the WT cache up to 2×. In the WB cache, there is no improvement for benchmark programs, such as *galgel(1)* and *galgel(2)*. The accuracy improvement reported in Fig. 16 has been computed according to Eq. (23).

$$Accuracy\ Improvment = \left|\frac{AVF - SVF}{SVF}\right| \times 100 \tag{23}$$

9.5. SFI experiments

In this section, we discuss the results of SFI experiments. We classify our FI runs as follows:

- **True Run**: An FI experiment in which the erroneous value has been masked by the CPU core and does not propagate to the output of the CPU. This run is similar to the golden run.
- **Faulty Run**: An FI experiment in which the injected fault propagates to the CPU core and is written back to the DL1 cache.
- **Failed Run**: An FI experiment in which the simulation terminates before it reaches the maximum number of instructions specified (100 M in our experiments) due to propagation of erroneous value to the main memory. For example, a failed run may occur while reading data from an invalid address with no actual data in this address.

Based on the above descriptions, we have defined the vulnerability factor according to FI experiments as below:

$$VF_{FI} = \frac{Faulty\ Runs + Failed\ Runs}{True\ Runs + Faulty\ Runs + Failed\ Runs} \times 100 \tag{24}$$

The results of SFI experiments for WT configuration are reported in Table 2. As presented in this table, the vulnerability factor according to SFI experiments is, on average 6.2%. Among the benchmark programs, *mesa* and *galgel(2)* are most and least vulnerable to the injected fault, respectively.

In Fig. 17, we report the results of SFI experiments for the WT configuration. The vulnerability factor obtained according to SFI experiments have been calculated with 95% confidence level. The sampling error is equal to $Z_{1-\alpha/2}\sqrt{p(1-p)/n}$ [49]. In this equation, p and n are the vulnerability factor based on *SFI* experiments and the number of FI experiments, respectively [49]. The other parameter, $Z_{1-\alpha/2}$, is the confidence level coefficient in our study, $\alpha = 95\%$ and Z = 1.96).

For the sake of accuracy, we have performed 1000 fault injection experiments to allow n to be large enough, resulting in a sampling error of 0.015. To achieve smaller error intervals, one would require at least 10× more SFI experiments which need significantly more computation power in the experiments. In Fig. 17, we report FI_{min} and FI_{max} according to this sampling error. In this figure, all VFs have been normalized to the maximum VF for each application

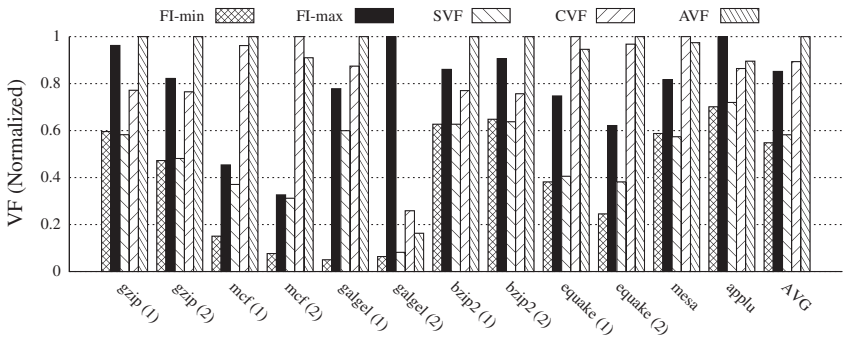

Fig. 17. Comparison of SFI (FI-min/FI-max), SVF, CVF, and AVF in a DL1 cache with write-through policy (normalized to the maximum VF for each application program).

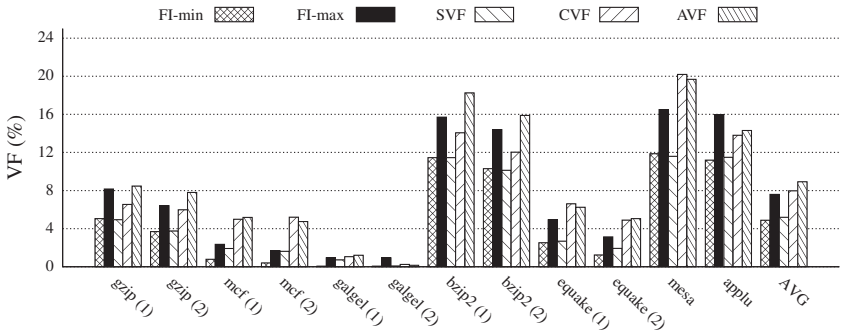

Fig. 18. Comparison of SFI (FI-min/FI-max), SVF, CVF, and AVF in a DL1 cache with write-through policy (non-normalized VFs).

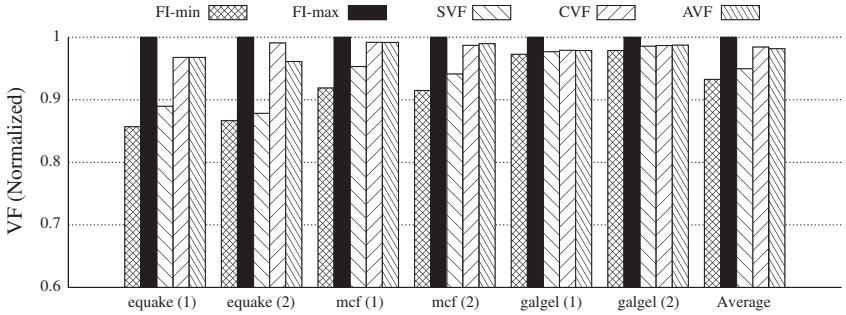

Fig. 19. Comparison of SFI (FI-min/FI-max), SVF, CVF, and AVF in a DL1 cache with write-back policy (normalized to the maximum VF for each application program).

program. FI_{min} and FI_{max} are equal to $VF_{FI} - Z_{1-\alpha/2}\sqrt{p(1-p)/n}$, and $VF_{FI} + Z_{1-\alpha/2}\sqrt{p(1-p)/n}$, respectively. Compared to the results of AVF, the SVF estimations computed by the CEDAR model mostly lies in $[FI_{MIN}, FI_{MAX}]$ interval, verifying our proposed SVF modeling results. As reported in this figure, the SVF results are within FI intervals for all benchmarks except gzip(1) and bzip (2). For these two benchmarks, both AVFs and SVFs are out of the FI interval. But as shown in Fig. 17, even in these two cases, SVFs are much closer to FI intervals as compared to AVFs. As illustrated in Fig. 17, the AVF results do not lie in the error estimation interval in most cases and also in the average case. We can conclude that by considering the IOD of the CPU core and read frequency in AVF technique we can enhance the accuracy of this model. Note as stated earlier, the results provided in Fig. 17 have been normalized to the maximum VF for each application program. The actual VFs (i.e., non-normalized numbers) are reported in Fig. 18.

In Fig. 19, we also report the results of SFI experiments for the WB configuration. Similar to the WT cache experiments, the vulnerability factor obtained according to SFI experiments have been calculated with 95% confidence level. The VFs reported in this figure have been normalized to the maximum VF for each application program. The results reported in Fig. 19 demonstrate that both AVF and the CEDAR model provide accurate VFs. This can be explained by the fact that replacement vulnerability contributes the most to the overall vulnerability in WB caches as studied in previous works [23,50]. Replacement vulnerability, classified as WPL and WRPL in the previous work (see Section 6), does not depend on the IOD factor. In other words, WB caches contain dirty blocks whose vulnerability is often not affected by IOD. As such, previous vulnerability modeling techniques such as AVF provide relatively accurate VFs for WB caches despite the fact that they overlook the IOD factor. The main contribution of the CEDAR model is to provide accurate VFs for WT caches, where overlooking the IOD factor can lead to significant inaccuracy. Note here we have used selective benchmark circuits whose program outputs are deterministic. Those benchmark programs that produce non-deterministic results cannot be verified by FI experiments and as such, they have been excluded from our experiments.

10. Conclusions

In this paper, we proposed an analytical vulnerability modeling technique, called CEDAR, to accurately estimate the *System-level Vulnerability Factor* of data-path components in an advanced microprocessor. In CEDAR, we took into account the number of read operations from the block during its life time in our computations. To do this, we proposed to use the input-to-output derating factor of the components in which the block is fed as the input. As a case study, we extracted the IOD of the DL1 cache for different configurations. Based on our investigations, IOD is mainly application dependent rather than configuration dependent. In addition, we extracted the vulnerability of the cache for WT and WB configurations using the CEDAR model as well as the previously well known proposed models. We have also performed *Statistical Fault Injection* experiment to verify the CEDAR model. Our experimental results through comparison showed that CEDAR is about 91% and 5% more accurate for WT and WB configurations, respectively as compared to the previously proposed analytical techniques.

References

[1] Baumann RC. Soft errors in advanced computer systems. IEEE Des Test Comput 2005;22(3):258–66.
[2] Shazli SZ, Abdul-Aziz M, Tahoori MB, Kaeli DR. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In: IEEE international test conference; October 2008. p. 1–10.
[3] Tahoori MB, Parulkar I, Alexandrescu D, Granlund K, Silburt A, Vinnakota B. Panel: reliability of data centers: hardware vs. software. In: IEEE/ACM international conference on design, automation and test in Europe conference (DATE); 2010. p. 1620.
[4] Baumann RC. Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans Device Mater Reliab 2005;5(3):305–16.

[5] Faure F, Velazco R, Violante M, Rebaudengo M, Sonza Reorda M. Impact of data cache memory on the single event upset-induced error rate of microprocessors. IEEE Trans Nucl Sci 2003;50(6):2101–6.
[6] Hwang SH, Choi GS. On-chip cache memory resilience. In: International symposium on high-assurance systems engineering; November 1998. p. 240–7.
[7] Kim S, Somani AK. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In: International conference on dependable systems and networks (DSN); 2002.
[8] Rebaudengo M, Reorda MS, Violante M. An accurate analysis of the effects of soft errors in the instruction and date caches of a pipelined microprocessor. In: IEEE/ACM international conference on design, automation and test in Europe (DATE); 2003. p. 602–7.
[9] Farazmand DKN, Ubal R, Kaeli D. Statistical fault injection-based AVF analysis of a GPU architecure. In: IEEE workshop on silicon errors in logic; 2012.
[10] Biswas A, Racunas P, Cheveresan R, Emer J, Mukherjee SS, Rangan R. Computing architectural vulnerability factors for address-based structures. In: International symposium on computer architecture (ISCA); 2005. p. 532–43.
[11] Asadi H, Sridharan V, Tahoori MB, Kaeli D. Balancing performance and reliability in the memory hierarchy. In: IEEE international symposium on performance analysis of systems and software (ISPASS); March 2005. p. 269–79.
[12] Haghdoost A, Asadi H, Baniasadi A. Using input-to-output masking for system-level vulnerability estimation in high-performance processors. In: The 15th CSI symposium on computer architecture and digital systems; September 2010. p. 91–8.
[13] Haghdoost A, Asadi H, Baniasadi A. System-level vulnerability estimation for data caches. In: IEEE 16th pacific rim international symposium on dependable computing (PRDC); 2010. p. 157–64.
[14] Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Annual IEEE/ACM international symposium on micro-architecture (MICRO); 2003. p. 29–40.
[15] Mukherjee S. Architecture design for soft errors. Morgan Kaufmann Publishers Inc.; 2008.
[16] Kim S, Somani AK. Area efficient architectures for information integrity in cache memories. In: International symposium on computer architecture (ISCA); May 1999. p. 246–55
[17] Saleh AM, Serrano JJ, Patel JH. Reliability of scrubbing recovery-techniques for memory systems. IEEE Trans Reliab 1990;39(1):114–22.
[18] Li Man-Lap, Ramachandran Pradeep, Karpuzcu Ulya R, Sastry Hari Siva Kumar, Adve Sarita V. Accurate microarchitecture-level fault modeling for studying hardware faults. In: International symposium on high-performance computer architecture (HPCA); February 2009. p. 105–16.
[19] Ramachandran P, Kudva P. Statistical fault injection. In: International conference on dependable systems and networks (DSN); 2008.
[20] Li X, Adve SV, Pradip B, Rivers JA. Softarch: an architecture-level tool for modeling and analyzing soft errors. In: International conference on dependable systems and networks (DSN); June–July 2005. p. 496–505.
[21] Somani AK, Trivedi KS. A cache error propagation model. In: Pacific rim international symposium on fault-tolerant systems (PRDC); 1997. p. 15–21.
[22] Weaver C, Emer J, Mukherjee SS, Reinhardt SK. Techniques to reduce the soft error rate of a high-performance microprocessor. In: International symposium on computer architecture (ISCA); June 2004. p. 264–75.
[23] Wang Shuai, Hu Jie, Ziavras SG. On the characterization and optimization of on-chip cache reliability against soft errors. IEEE Trans Comput 2009;58(9).
[24] Tang L, Wang S, Hu J, Hu XS. Characterizing the L1 data cache's vulnerability to transient errors in chip-multiprocessors. In: IEEE computer society annual symposium on VLSI (ISVLSI); 2011. p. 266–71.
[25] Sridharan V, Kaeli DR. Eliminating microarchitectural dependency from architectural vulnerability. In: High performance computer architecture (HPCA); 2009. p. 117–28.
[26] Sridharan V, Kaeli DR. Using hardware vulnerability factors to enhance AVF analysis. In: International symposium on computer architecture (ISCA); 2010.
[27] Duan L, Peng L, Li B. Predicting architectural vulnerability on multithreaded processors under resource contention and sharing. IEEE Trans Depend Secure Comput 2013;10(2):114–27.
[28] Li X, Adve SV. Architecture-level soft error analysis: examining the limits of common assumptions. In: International conference on dependable systems and networks (DSN); 2007.
[29] Wang Nicholas J, Mahesri Aqeel, Patel Sanjay J. Examining ace analysis reliability estimates using fault-injection. In: International symposium on computer architecture (ISCA); 2007. p. 460–9.
[30] Biswas A, Racunas P, Emer J, Mukherjee Shubhendu. Computing accurate AVFs using ACE analysis on performance models: a rebuttal. IEEE Comput Archit Lett 2007;7(2):21–4.
[31] Costenaro E, Evans A, Alexandrescu D, Chen L, Tahoori M, Nicolaidis M. Towards a hierarchical and scalable approach for modeling the effects of SETs. In: IEEE workshop on silicon errors in logic-system effects (SELSE); 2013.
[32] Evans A, Alexandrescu D, Costenaro E, Chen Liang. Hierarchical RTL-based combinatorial SER estimation. In: IEEE international on-line testing symposium (IOLTS); 2013. p. 139–44.
[33] Dixit A, Wood Alan. The impact of new technology on soft error rates. In: IEEE international reliability physics symposium (IRPS); 2011. p. 5B.4.1–5B.4.7.
[34] Bacha Anys, Teodorescu Radu. Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors. In: Proceedings of international symposium on computer architecture (ISCA). ACM; 2013. p. 297–307.
[35] Intel Itanium processor 9300 series and 9500 series. <http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/itanium-9300-9500-datasheet.pdf>.
[36] POWER7 system RAS key aspects of power systems reliability, availability, and serviceability. <http://www-07.ibm.com/tw/imc/seminar/download/2010/POWER7_RAS_Whitepaper.pdf>.
[37] Gold Brian T, Ferdman Michael, Falsafi Babak, Mai Ken. Mitigating multi-bit soft errors in L1 caches using last-store prediction; 2007.
[38] Szafaryn LG, Meyer BH, Skadron K. Evaluating overheads of multibit soft-error protection in the processor core. MICRO, IEEE 2013;33(4):56–65.
[39] Kim Jangwoo, Hardavellas N, Mai Ken, Falsafi B, Hoe JC. Multi-bit error tolerant caches using two-dimensional error coding. In: IEEE/ACM international symposium on microarchitecture, MICRO; 2007. p. 197–209.
[40] Slayman CW. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. IEEE Trans Device Mater Reliab 2005;5(3):397–404.
[41] Maiz J, Hareland S, Zhang K, Armstrong P. Characterization of multi-bit soft error events in advanced SRAMs. In: IEEE international electron devices meeting. IEDM Technical Digest; 2003. p. 21.4.1–21.4.4.
[42] Lee Ikhwan, Basoglu Mehmet, Sullivan Michael, Yoon Doe Hyun, Kaplan Larry, Erez Mattan. Survey of error and fault detection mechanisms. University of Texas at Austin; Tech. rep; 2011.
[43] Strukov D. The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories. In: Fortieth Asilomar conference on signals, systems and computers. ACSSC; 2006. p. 1183–7.
[44] Desikan R, Burger D, Keckler SW, Austin T. Sim-alpha: a validated, execution-driven alpha 21264 simulator; 2001.
[45] Fu X, Li T, Fortes J. Sim-soda: a unified framework for architectural level software reliability analysis. In: Workshop on modeling, benchmarking and simulation; 2006.
[46] Standard performance evaluation corporation. SPEC CPU2000 benchmarks; 2000. <http://www.specbench.org/cpu2000>.
[47] Kessler R. The alpha 21264 microprocessor. IEEE Micro 1999;19(2):24–36.
[48] Cho Hyungmin, Mirkhani S, Cher Chen-Yong, Abraham JA, Mitra S. Quantitative evaluation of soft error injection techniques for robust system design. In: Design automation conference (DAC); 2013. p. 1–10.
[49] Leveugle R, Calvez A, Maistri P, Vanhuawaert P. Statisctical fault injection: quantified error and confidence. In: IEEE/ACM international conference on design, automation and test in Europe conference (DATE); 2009. p. 502–6.
[50] Asadi Hossein, Sridharan Vilas, Tahoori Mehdi B, Kaeli David. Vulnerability analysis of L2 cache elements to single event upsets. In: IEEE/ACM international conference on design, automation and test in Europe conference (DATE); 2006. p. 1276–81.