Using Input-to-Output Masking for System-Level Vulnerability Estimation in High-Performance Processors

Alireza Haghdoost[1] Hossein Asadi[1] Amirali Baniasadi[2]
[1] *Department of Computer Engineering, Sharif University of Technology, Tehran, Iran*
[2] *Department of Electrical and Computer Engineering, University of Victoria, Canada*
haghdoost@ce.sharif.edu asadi@sharif.edu amirali@ece.uvic.ca

***Abstract*—In this paper, we enhance previously suggested vulnerability estimation techniques by presenting a detailed modeling technique based on Input-to-Output Masking (IOM). Moreover we use our model to compute the System-level Vulnerability Factor (SVF) for data-path components in a high-performance processor. As we show, recent suggested estimation techniques overlook the issue of error masking, mainly focusing on time periods in which an error could potentially propagate in the system. In this work we show that this is incomplete as it ignores the masking impact. Our results show that including the IOM factor can significantly affect the system-level vulnerability for data-path components. As a case study, we analyze the IOM factor for CPUs with different configurations. Our results show that the average variation of the IOM factor is less than 5%. Meantime, the IOM factor varies between 24% to 76% for the applications studied here. Accordingly we find the IOM factor to be less configuration dependent and mainly workload dependent.**

***Index Terms*—System Level-Vulnerability, Architectural Vulnerabulity Factor, High-Performance Processors, Fault Masking Factor.**

I. Introduction

The data integrity of high-end and mainstream processors is threatened by cosmic and terresial energetic particles such as neutrons and alpha particles from packaging materials. These energetic particles can change the state of storage elements such as flip-flops and SRAM cells within processors and cause a transient error. The radiation-induced transient errors, also called soft errors, occur more often than hard errors in the current VLSI technology [1], [2]. Recent research study has shown that soft errors could have significant impact on the data integrity of the current microprocessor technology [3].

As technology continues to scale down and the number of transistors per chip continues to move up, the soft error rate per chip is expected to increase for the next several years [1]. Accordingly, designers would need to incorporate aggressive protection techniques in future microprocessor designs. An important aspect of designing cost-effective protection techniques is developing accurate soft error vulnerability models for individual components. This will help understanding the extend of vulnerability for data-path components such as cache, register files, and load/store queues before developing protection techniques. Having an accurate model for such components would facilitate making informed decisions about the level of protection needed across data-path components and target workloads. The right protection level for data-path structures reduces data loss probability and therefore would increase system reliability.

Recent field study over several thousands of systems indicates that in the current processor technology, a majority of system reboots are initiated by single event upsets (or SEUs) occurring in data-path components such as cache and register files [3]. Errors in such structures can easily propagate to the system outputs and can significantly reduce the system reliability. In particular, cache reliability comes with high importance as errors occurring in the data cache can propagate to higher memory levels, and can easily lead to data integrity issues [4], [5]. While designing caches with low access time and miss rate is an important goal, maintaining low power dissipation and high reliability have also become necessary. This is particularly true for high-end and mainstream processors where reliability has always been a vital concern.

Previous studies have introduced analytical models to compute vulnerability of data-path components such as cache and register file to SEUs [6], [7], [8], [9]. Such models often provide fast estimation but suffer from inaccuracies as the system-level impact of soft errors are not taken into account in these models. More accurate measurements, i.e., fault injection (FI) strategies [10], [11], [12], [13], are both time-consuming, due to the large number of runs, and still prone to inaccuracy, due to the limited number of addresses targeted.

The goal of this study is to introduce a new vulnerability estimation technique to improve accuracy of previous estimation methods and maintain low estimation time. We do so by taking into account an important parameter ignored by earlier studies. Previous studies mainly rely on 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. While critical time is an important factor, it is not the only one.

In this work, we present a modeling technique based on the *Input-to-Output Masking* (IOM) factor. We define the IOM factor of a component as the percentage of errors masked when propagating erroneous values from the inputs to the outputs of the component. We present a technique to compute the IOM factor of components for a high-performance processor. Using the IOM factor, we also present a modeling technique to estimate the *Component-level Vulnerability Factor* (CVF) and the *System-level Vulnerability Factor* (SVF) of the data-path components of a high-performance processor. We define

CVF as the portion of errors occurring within a component and propagating to other components of the system. We also define SVF as the percentage of errors occurring within a component and propagating to the system outputs causing system failure.

We report our VF estimations for SPEC'2K benchmarks and compare to previously suggested models. In particular, our results show that our proposed modeling technique improves the accuracy of the previously suggested models for write-through cache by more than 40%. We also analyze the IOM factor for CPUs with different configurations.

Our results show that the IOM factor of the processor core shows significant variation for different applications. Meantime average IOM variation is less than 5% for different processor configurations including speculative load execution, double sized load and store queue, and reorder buffer. The results confirm that the IOM factor of the CPU is less configuration dependent and is mainly workload dependent.

The rest of the paper is organized as follows. In Section II we review traditional error protection and previously suggested vulnerability modeling techniques and their limitations. In Section III we elaborate the inaccuracies existing in previous analytical techniques and describe our motivating observations. In Section IV we present our vulnerability modeling technique. In Section V we investigate our solution for cache as a case study. In Section VI we present the experimental setup. In Section VII we discuss our methodology and report results. Finally, in Section VIII we offer concluding remarks.

II. Related Work and Their Limitations

In this section, we first review the limitations of traditional protection techniques. Then, we discuss previous vulnerability factor modeling techniques and their drawbacks.

A. Limitations of Traditional Protection Techniques

Spatial redundancy techniques such as byte or block-based parity or *Single-bit Error Correction, Double-bit Error Detection* (SEC-DED) ECC, are commonly used to protect data in cache [4]. Such solutions, however, are rarely used for tag addresses as they impose several limitations. First, redundancy incurs area and power overhead, which increases proportionately with the cache size. Second, maintaining high throughput limits the redundancy checking hardware as it should not increase cache access time latency considerably. Unfortunately, techniques like SEC-DED codes could potentially add an extra clock cycle to the cache access time, which can severely degrade performance. Error detection techniques such as parity can be used to detect errors in tag arrays. However, the recovery of an erroneous tag array would be either impossible or very difficult [6] under such techniques.

Scrubbing is an alternative technique that could be used to improve cache reliability in conjunction with SEC-DED ECC [14]. 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 typically used 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 [4].

As it is difficult to provide guaranteed reliability for caches, an alternative approach is disabling the cache in safety-critical applications [10]. By disabling the cache, the area susceptible to SEUs is drastically reduced hence increasing processors dependability dramatically. 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.

B. Previous Vulnerability Factor Modeling Techniques

Many previously proposed cache reliability estimation methods rely on fault injection strategies [10], [11], [12], [13], [15], [16]. When using an FI strategy, a limited number of memory addresses are targeted. Several workloads are then run 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 [17]. 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 [18]. 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 [4]. They also studied tag array soft error susceptibility [4].

Mukherjee et al. [7], [19] 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 as a result of a raw error event in the structure [7], [19]. 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 [8]. The 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. Several experiments are performed on various data cache configurations. Accordingly, a flushing technique is proposed to enhance reliability.

Asadi et al. introduced a critical time model to estimate the reliability of an unprotected or partially protected cache [6]. The proposed model computes cache vulnerability using the residency time of *Critical Words* (CW) in the cache. A CW is defined as a word in the cache 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.

S. Wang et al. introduced *Temporal Vulnerability Factor* or TVF as a soft error characterization model [20] to capture the upper bound of the cache vulnerability factor. The proposed model extends the work presented in [6] by calculating the critical times at various granularity levels, e.g. a cache line, a word, or a byte [20].

Li et al. studied the limitations of the AVF modeling [21] and showed that the 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 bigger than the actual vulnerability factor.

N.J. Wang et al. reported that statistical fault injection (SFI) at register transfer level (RTL) can generate more accurate AVFs compared to ACE analysis [22]. 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 [23].

III. Motivation and Limitations of Previous Analytical Models

Previous vulnerability analytical models mainly focus on computing vulnerability at the component-level. This means the vulnerability factor is defined as the percentage of errors occurring within a component that propagate to the outputs of the component. Such studies [6], [24], [7], [20], 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.

Here we present an example on a general purpose register file (GPRF) , shown in Fig. 1, to provide better understanding regarding the impact of masking. Assume that the target byte within the GPRF is struck by an SEU changing one of its eight bits. The target byte is then sent to the ALU unit. Let's also assume that the byte is used as an operand in a *Logical AND* arithmetic operation. Although the incorrect value of the byte is transferred from the GPRF to the inputs of the ALU unit, it does not necessarily propagate from the ALU inputs to the outputs of the ALU unit.

To further investigate the masking factor in this example, assume that the target byte's initial value is 0x05 (denoted in hexadecimal format). Let's also assume that an SEU event inverts the most significant bit of this byte, changing the byte from 0x05 to 0x85. If this byte is used as an operand of a logical *AND* operation with a second operand equal to 0x36 the incorrect value of the byte would not propagate to the outputs of the ALU unit. This is because the logical AND will mask the error bit when producing the final outcome, i.e., $(0x05\ AND\ 0x36) = (0x85\ AND\ 0x36) = 0x04$.

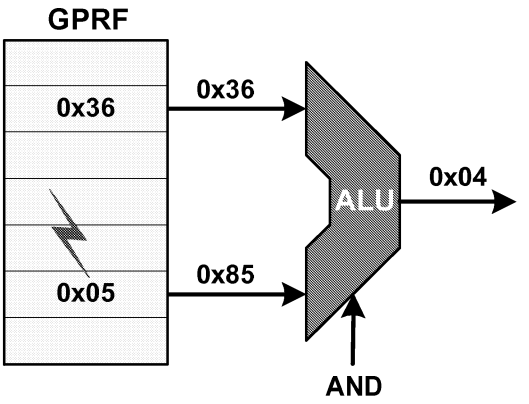

Fig. 1. Error masking example in the CPU

Error masking can occur in both data-path components and the control logic. 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 this work.

IV. Computing SVF Using Input-to-Output Masking Factor

In order to analyze the impact of error masking, we define the IOM factor of a component as the percentage of errors being masked when propagating erroneous values from the inputs to the outputs of the component. Previous work has introduced the *Derating Factor* (DF) to quantify the masking behavior at the circuit level [25]. DF is defined as the probability of propagating an incorrect value along logical paths of a combinational logic. In particular, DF [25], [26] is expressed as the probability of propagating an erroneous value from a logic gate to the circuit outputs. In this paper, we extend this concept from circuit-level to component-level in order to compute the error propagation probability from inputs to the outputs of a component. We refer to this probability as the *Input-to-Output Derating* of a component or $IOD_{component}$. As an example, IOD_{cpu} is defined as the probability of an erroneous value propagating from the inputs of the CPU to the outputs of the CPU. We refer to IOD's complement as IOM , i.e. $IOM = 1 - IOD$.

In our proposed modeling technique, we use the IOM factor to accurately compute the *System-level Vulnerability Factor* (SVF) of data-path components of a high-performance processor. We define CVF as the portion of errors occurring within a component and propagating to other components of the system. We also define SVF as the percentage of errors occurring within a component that propagate to the system outputs and cause a system failure.

To illustrate how IOM is used to compute system-level vulnerability factor of different components, consider the block digram of a typical high-performance processor (e.g., Xeon) shown in Fig. 2. Here we compute SVF for the *Instruction*

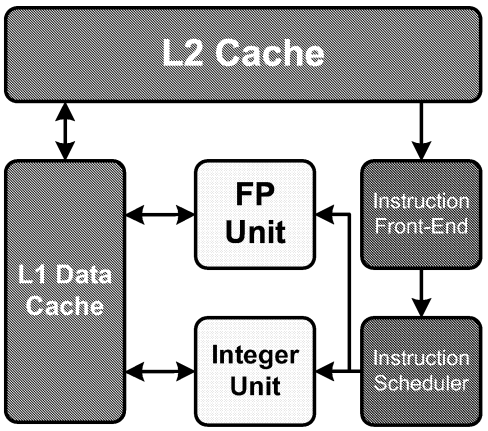

Fig. 2. Components in typical high-performance processor

Scheduler (IS), i.e., SVF_{IS}. CVF_{IS} can be obtained by computing the percentage of errors occurring within IS and propagating to the inputs of either the Integer Unit (IU) or the FP Unit. For the sake of simplicity, we assume that the processor is running integer benchmarks and ignore the FP unit. An error propagating to the inputs of the integer unit propagates to the L1 data cache if it is not masked by the integer unit. Therefore, SVF_{IS} can be computed as $CVF_{IS} \times (1 - IOM_{IU})$.

Now consider computing the SVF of the *Instruction Front-End* (IFE) for integer benchmarks. First, we need to compute CVF_{IFE} by taking into account the portion of errors occurring within the instruction front-end and propagating to the inputs of the instruction scheduler. An error propagating to the inputs of the instruction scheduler can propagate to the L1 data cache. Therefore, SVF_{IFE} can be computed as $CVF_{IFE} \times (1 - IOM_{IS}) \times (1 - IOM_{IU})$.

In the above analysis, we assume that errors propagating to the L1 data cache cause a data integrity issue and will result in a system failure. We can further increase the accuracy of the computed SVFs by including the input-to-output masking factor of the L1 data cache ($DL1$). For example, the SVF of the IFE unit can be rewritten as $CVF_{IFE} \times (1 - IOM_{IS}) \times (1 - IOM_{IU}) \times (1 - IOM_{DL1})$. In this study, we ignore IOM_{DL1} as we expect this number to be very close to zero. That is errors propagating to L1 data cache is very unlikely to be masked by the DL1 cache.

V. Case Study: SVF and CVF of Write-Through Cache Memory

In this section, we use our proposed model to accurately compute the CVF and the SVF of a write-through cache. Here, we use IOM_{cpu} to model CPUs masking behavior. A previous study has defined a *Critical Word* (CW) or *Critical Byte* (CB) as a word or a byte guaranteed to propagate from the cache to other locations in the memory system or to the CPU [6]. The residence time of the CB (or CW) in the cache is referred to as the *Critical Time* (CT). Here, we extend this concept to better estimate the impact of cache vulnerability factor on the overall system reliability.

We define *System-level Vulnerability* (SV) of a byte within a cache memory as the sum of the time periods where an incorrect value in those time periods will lead to system failure. The following formula calculates SV for the access pattern to a byte within a write-through cache presented in Fig. 3.

$$\begin{aligned} SV &= (t_1 - t_0).(1 - IOM_{cpu}) + (t_2 - t_1).(1 - IOM_{cpu}) \\ &+ (t_3 - t_2).(1 - IOM_{cpu}) \\ &+ (t_1 - t_0).IOM_{cpu}.(1 - IOM_{cpu}) \\ &+ (t_2 - t_1).IOM_{cpu}.(1 - IOM_{cpu}) \\ &+ (t_1 - t_0).IOM_{cpu}^2.(1 - IOM_{cpu}) \end{aligned} \quad (1)$$

In the above equations, the first three terms, $(t_1 - t_0) \times (1 - IOM_{cpu})$, $(t_2 - t_1) \times (1 - IOM_{cpu})$ and $(t_3 - t_2) \times (1 - IOM_{cpu})$, represent the likelihood of an error occurring in $t_1 - t_0$, $t_2 - t_1$ and $t_3 - t_2$ being propagated to the system outputs by the read accesses immediately following (here R_1, R_2, and R_3 respectively). The next two terms model errors being masked by the CPU after the earlier read but propagated by a subsequent read access. For example, $(t_1 - t_0) \times IOM_{cpu} \times (1 - IOM_{cpu})$ represents the likelihood of an error masked by the CPU after being read by R_1, impacting the CPU output later by being read (and not being masked) by R_2. The last term, $(t_1 - t_0) \times IOM_{cpu}^2 \times (1 - IOM_{cpu})$, represents an error occurring during $t_1 - t_0$, masked by the CPU after both R_1 and R_2, but propagated by to the system output after R_3. Note we can also represent SV according to Equation 2.

$$SV = (1 - IOM_{cpu}).CV \quad (2)$$

We refer to the second section of SV in the Equation 2 , i.e. CV, as *Component-level Vulnerability*. In the example of Fig. 3, CV is computed according to Equation 3.

$$\begin{aligned} CV &= (t_1 - t_0) + (t_2 - t_1) + (t_3 - t_2) \\ &+ (t_1 - t_0).IOM_{cpu} + (t_2 - t_1).IOM_{cpu} \\ &+ (t_1 - t_0).IOM_{cpu}^2 \end{aligned} \quad (3)$$

SV analysis formulation could vary based on different cache access patterns. In general, write-through cache has only three types of valid DL1 cache accesses on a byte (fill, read and evict). This is shown as a general example in Fig. 4

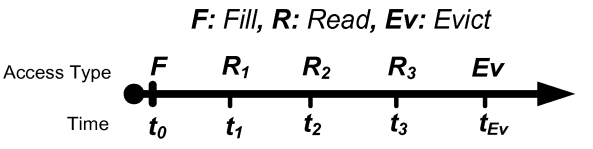

Fig. 3. An example of an access pattern to write-through cache to demonstate how SVF and CVF can be computed using the IOM factor

We use Equation 4 to calculate SVF and CVF of the byte in a write-through cache. We assume that the byte is accessed n times before eviction. In this equation, $CV(F : R_1)$ computes CV from the fill time to the first read operation (R_1). $CV(F : R_2)$ computes CV from the fill time to the second read operation (R_2), and finally, $CV(F : R_3)$ computes CV from the fill time to the third read operation (R_3). Note that in Equation 4 through Equation 6 we assume that $t_{Fill} = t0 = 0$.

$$\begin{aligned} CV(F:R_1) &= t_1 \\ CV(F:R_2) &= t_1 + (t_2 - t_1) + t_1.IOM_{cpu} \\ &= t_2 + t_1.IOM_{cpu} \\ &= t_2 + CV(F:R_1).IOM_{cpu} \\ CV(F:R_3) &= t_1 + (t_2 - t_1) + (t_3 - t_2) + t_1.IOM_{cpu} \\ &+ t_1.IOM_{cpu}^2 + (t_2 - t_1).IOM_{cpu} \\ &= t_3 + t_1.IOM_{cpu} + t_1.IOM_{cpu}^2 \\ &+ (t_2 - t_1).IOM_{cpu} \\ &= t_3 + CV(F:R_2).IOM_{cpu} \end{aligned} \quad (4)$$

In general, for n successive read operations, we use Equation 5 to compute CV from the fill time to the last read operation.

$$CV(F:R_n) = tn + CV(F:R_{n-1}).IOM_{cpu} \quad (5)$$

Once we have computed $CV(F : R_n)$, SV can be computed according to Equation 6.

$$SV(F:R_n) = (1 - IOM_{cpu}).CV(F:R_n) \quad (6)$$

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

$$CV_{Cache} = \sum_{i=1}^{N} CV_i \quad (7)$$

$$SV_{Cache} = \sum_{i=1}^{N} SV_i \quad (8)$$

$$CVF_{Cache} = \frac{CV_{Cache}}{TT \times M} \quad (9)$$

$$SVF_{Cache} = \frac{SV_{Cache}}{TT \times M} \quad (10)$$

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

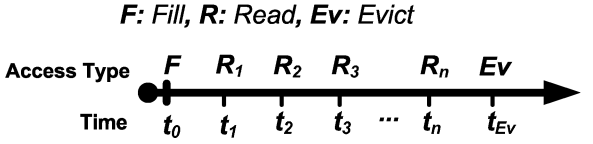

Fig. 4. General access patterns to compute vulnerability

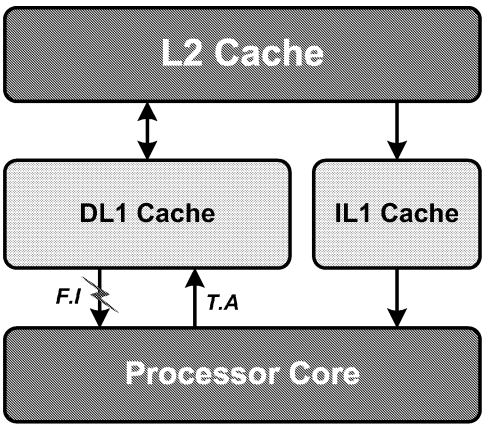

Fig. 5. Estimating IOM of the CPU core

VI. Experimental Setup

For these experiments, we used *sim-alpha*, which is a cycle accurate alpha 21264 processor simulator [27]. To evaluate the reliability of DL1 caches, we have extended the source code of sim-alpha to implement our vulnerability estimation method. We have also implemented the write-through cache policy in this toolset. We have incorporated AVF estimation algorithms from *sim-soda* [28] in our simulation framework.

Our evaluation uses the SPEC'2K benchmark suite [29] compiled for the Alpha ISA [30]. We run our benchmarks for 100M instructions and 100M cool-down after skipping the first 100M instructions for SVF, CVF, and AVF estimation.

The default system parameters (cache size, associativity, etc.) are detailed in Table I, and were chosen to be representative of modern state-of-the-art processors.

A. IOM Experimental Setup

To estimate how often the CPU masks an error occurring in the cache (here referred to as IOM) we modified sim-alpha for fault injection experiment. In our IOM estimation method we flipped one bit in a randomly selected load instruction operand and investigated whether the modified data reaches the CPU output. We inject a limited number of faults using a model depicted in Fig. 5. In this experimental setup, we perform fault injection only on load instructions (i.e., when data is transferred from the D-cache to the CPU core) and observe fault propagation on store instructions (i.e., when data is transferred from the CPU to the D-cache). We follow this procedure for 400 times per application and report the percentage of the time an error not being propagated to the CPU output (i.e., store instructions). As an example, when running *gzip*, if 100 times out of 400 number of fault injection experiments do not propagate to the CPU output, IOM_{cpu} when running *gzip* would be equal to $\frac{100}{400} = 0.25$.

We run all instructions of benchmarks in simulator using functional simulation mode for IOM fault injection estimation. Compared to previous fault-injection methods presented in [10], [11], [12], [13], since IOM is computed in functional simulation mode it is accomplished in a more timely fashion

with very negligible impact on accuracy. The simulation time to estimate IOM_{cpu} for some programs of SPEC'2K benchmark suite is reported in Fig. 6. We report IOM for the applications studied here and for the configuration presented in Table I.

VII. Results

In this section we report our estimation results. In subsection VII-A we report our estimated IOM and the IOM sensitivity to the number of runs and different processor configurations. In subsection VII-B we report SVF, CVF and AVF for the write-through cache. Finally in subsection VII-C we report accuracy improvement for our suggested model.

A. IOM Experiments

In Fig. 7, we report IOM_{CPU} for the applications studied here. Average IOM_{CPU} is 0.59. This means 59% of single bit errors in the data operands are masked by the CPU. *Galgel* shows maximum IOM of 0.76 while *applu* shows a minimum IOM of 0.24. In Fig. 7, we also report the IOM sensitivity to the number of iterations. The results confirm that relatively accurate IOMs can be obtained by using a small number of fault-injection experiments on load instructions. Our results show that even with 100 fault injection iterations, the computed IOM is as accurate as 400 times iterations.

In Fig. 8 we report IOM sensitivity for the CPU to executing *Speculative Load* instructions (SL) and disabling *Speculative Update of Branch and Line Predictor* (SUBLP). Among programs, *bzip* shows maximum IOM change of 8.9% while *galgel* shows a minimum IOM change of 0.67% when executing speculative load instructions. The average IOM sensitivity to executing speculative load instructions is 2.2%. When disabling SUBLP *applu* shows maximum IOM change of 19.6% while *galgel* shows a minimum IOM change of 0.22%. The average IOM sensitivity to SUBLP is 2.3%

In Fig. 9 we report IOM sensitivity to doubling the size of *Load and Store Queue* (LSQ) and *Re-Order Buffer* (ROB) compared to the base processor configuration reported in Table I. Average IOM sensitivity to doubling LSQ and ROB size is 0.43% and 4.5%, respectively.

The results shown in Fig. 8 and Fig. 9 confirm that IOM shows little sensitivity to processor configuration changes

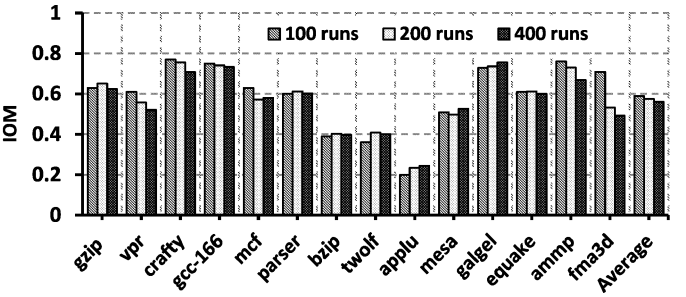

Fig. 7. IOM Sensitivity to the total number of fault injections applied to load operations

while showing significant variations from one application to another one. We conclude that IOM_{CPU} is less configuration dependent and is mainly workload dependent.

B. Write-Through Cache Memory Vulnerability Analysis

In Fig. 10, bars from left to right report SVF, CVF, and AVF, respectively for the write-through cache configuration detailed in Table I. Average SVF, CVF, and AVF are 8.4%, 18.4% and 10.3%, respectively. The maximum difference is reported for *crafty* where SVF, CVF, and AVF are 6.3%, 21.8% and 10%, respectively. In a write-through cache data blocks have a shorter critical time making an error propagating in system less likely.

C. Accuracy Improvement

In Fig. 11, we report accuracy improvement for our suggested model compared to AVF for the write-through cache. As reported accuracy improvement is about 40% for the write-through cache. The maximum accuracy improvement using our proposed modeling technique is about 100% which is achieved when running *applu*.

VIII. Conclusions

In this paper, we presented the *Input-to-Output Masking* (IOM) factor that can be used to accurately compute the *System-level Vulnerability Factor* (SVF) for data-path components of a high-performance processor. As a case study, we analyzed the IOM factor of cache with possible configurations

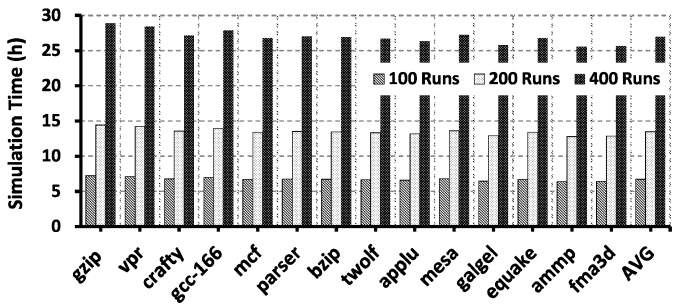

Fig. 6. Fault injection simulation time to estimate the IOM factor of the cpu core

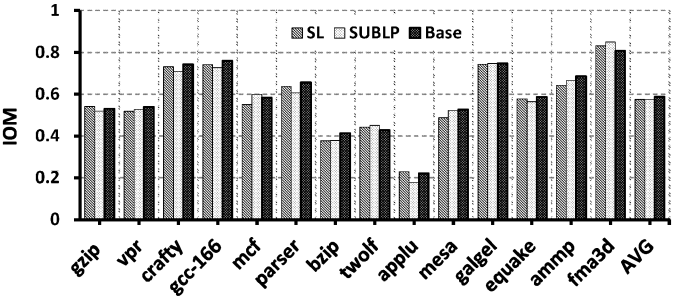

Fig. 8. IOM sensitivity to speculative load and speculative update of branch and line predictor

TABLE I
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 (IL1)	64KB, 2-way, 64 byte lines 1 cycle latency
L1 Data Cache (DL1)	64KB, 4-way, 64 byte lines 3 cycle latency
L2	2MB unified, direct-mapped 64 byte lines, 7 cycle latency
Memory	100 cycle latency
Branch Logic	
Predictor Branch miss-prediction penalty	Hybrid, 4K global two-level 1KB local, 4K choice 7 cycles
BTB	512 entry, 4-way
Mis-prediction Penalty	7 cycles

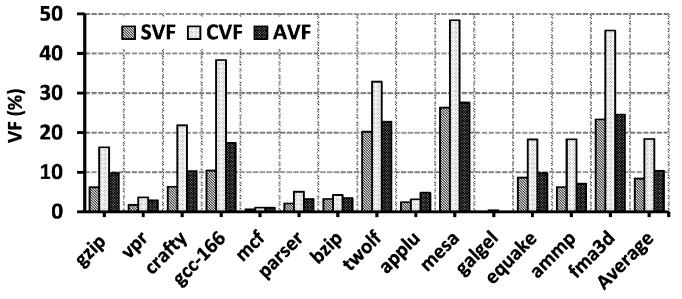

Fig. 10. Comparison of SVF, CVF, and AVF in a DL1 cache with write-through policy

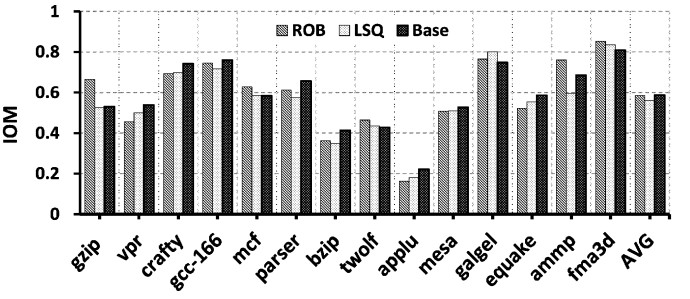

Fig. 9. IOM Sensitivity to double size of LSQ and ROB

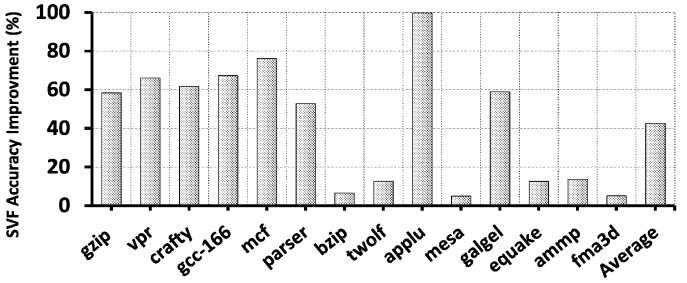

Fig. 11. Accuracy improvement of SVF model compared to AVF modeling technique for the write-through cache

for a high-performance processor. Our results showed that the IOM factor is less configuration dependent and is mainly workload dependent. We also reported our vulnerability estimations using the IOM factor for a write-through data cache and compared to previously suggested models. Our experimental results showed that the proposed modeling technique improves the accuracy of the previously suggested models by more than 40%.

ACKNOWLEDGMENT

This work is supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program and by the Institute for Research in Fundamental Sciences (IPM) in Iran.

REFERENCES

[1] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005.

[2] M.J. Gadlage, J.R. Ahlbin, B. Narasimham, V. Ramachandran, C. A. Dinkins, N.D. Pate, B.L. Bhuva, R.D. Schrimpf, L.W. Massengill, R.L. Shuler, and D. McMorrow. Increased single-event transient pulsewidths in a 90-nm bulk cmos technology operating at elevated temperatures. *IEEE Transactions on Device and Materials Reliability*, 10(1):157–163, 2010.

[3] S.Z. Shazli, M. Abdul-Aziz, M.B. Tahoori, and D.R. Kaeli. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *IEEE International Test Conference*, pages 1–10, October 2008.

[4] S. Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 246–255, May 1999.

[5] W. Zhang, S. Gurumurthi, M. Kandemir, and A. Siavasubramaniam. Icr: In-cache replication for enhancing data cache reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 291–300, June 2003.

[6] H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli. Balancing performance and reliability in the memory hierarchy. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS05)*, pages 269–279, March 2005.

[7] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-36)*, pages 29–40, 2003.

[8] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 532–543, 2005.

[9] Pablo Montesinos, Wei Liu, and Josep Torrellas. Using register lifetime predictions to protect register files against soft errors. *Dependable Systems and Networks, International Conference on*, 0:286–296, 2007.

[10] F. Faure, R. Velazco, M. Violante, M. Rebaudengo, and M. Sonza Reorda. Impact of data cache memory on the single event upset-induced error rate of microprocessors. *IEEE Transactions on Nuclear Science*, 50(6):2101–2106, 2003.

[11] S. H. Hwang and G. S. Choi. On-chip cache memory resilience. In *Proceedings of the International Symposium on High-Assurance Systems Engineering*, pages 240–247, November 1998.

[12] S. Kim and A.K. Somani. Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2002.

[13] M. Rebaudengo, M. S. Reorda, and M. Violante. An accurate analysis of the effects of soft errors in the instruction and date caches of a pipelined microprocessor. In *Proceedings of the IEEE/ACM International Conference on Design, Automation and Test in Europe (DATE)*, pages 602–607, 2003.

[14] A. M. Saleh, J. J. Serrano, and J. H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Transactions on Reliability*, 39(1):114–122, April 1990.

[15] Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 105–116, February 2009.

[16] P. Ramachandran and P. Kudva. Statistical fault injection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[17] X. Li, S.V. Adve, B. Pradip, and J.A. Rivers. Softarch: an architecture-level tool for modeling and analyzing soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 496–505, June-July 2005.

[18] A. K. Somani and K. S. Trivedi. A cache error propagation model. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems (PRDC)*, pages 15–21, 1997.

[19] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*, pages 264–275, June 2004.

[20] Shuai Wang, Jie Hu, and Ziavras S. G. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Transactions on Computers*, 58(9), September 2009.

[21] X. Li and S.V. Adve. Architecture-level soft error analysis: Examining the limits of common assumptions. In *International Conference on Dependable Systems and Networks (DSN)*, June 2007.

[22] Nicholas J. Wang, Aqeel Mahesri, and Sanjay J. Patel. Examining ace analysis reliability estimates using fault-injection. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 460–469, New York, NY, USA, 2007. ACM.

[23] Joel Emer Shubhendu Mukherjee Arijit Biswas, Paul Racunas. Computing accurate avfs using ace analysis on performance models: A rebuttal. *IEEE Computer Architecture Letters*, 7(2):21–24, 2007.

[24] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 532–543, 2005.

[25] H.T. Nguyen and Y. Yagil. A systematic approach to ser estimation and solutions. In *Proceedings of the 41st Annual International Reliability Physical Symposium (IRPS)*, pages 60–70, 2003.

[26] H. Asadi and M. B. Tahoori. Soft error derating computation in sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD)*, November 2006.

[27] R. Desikan, D. Burger, S. W. Keckler, and T. Austin. Sim-alpha: A Validated, Execution-Driven Alpha 21264 Simulator, 2001.

[28] X Fu, T Li, and J Fortes. Sim-soda: A unified framework for architectural level software reliability analysis. In *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[29] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks, http://www.specbench.org/cpu2000, 2000.

[30] R. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.