Online Computing and Predicting Architectural Vulnerability Factor of Microprocessor Structures

Songjun Pan††  Yu Hu†  Xiaowei Li†*$
†Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China, 100190
‡Graduate University of Chinese Academy of Sciences, Beijing, P.R. China, 100049
{pansongjun, huyu, lxw}@ict.ac.cn

Abstract—Soft Errors have emerged as a key challenge to microprocessor design. Traditional soft error tolerance techniques (such as redundant multithreading and instruction duplication) can achieve high fault coverage but at the cost of significant performance degradation. Prior research reports that soft errors can be masked at the architecture level, and the degree of such masking, named as architecture vulnerability factor (AVF), can vary significantly across workloads and individual structures, hence strict redundant execution may not be necessary for soft error tolerance. In this work, we exploit the AVF varying feature to adaptively tune reliability and performance. We present an infrastructure to online compute and predict AVF for three microprocessor structures (IQ, ROB, and LSQ), guiding when the protection scheme should be activated to improve reliability. Experimental results show that our method can efficiently compute the AVF for different structures independent of hardware configurations. The average differences between our method and a prior offline AVF computing method are 0.10, 0.01, and 0.039 for IQ, ROB, and LSQ, respectively.

Keywords—soft error; architecture vulnerability factor; microarchitecture; reliability; performance

I. INTRODUCTION

With the continuous decreasing of CMOS feature size, future shipped microprocessors will be increasingly vulnerable to radiation-induced soft errors. Soft errors (also known as single event upsets or transient faults) are caused by energetic particles such as alpha particles from packaging materials and neutrons from cosmic rays, such errors may incur serious program crash by flipping bits in storage cells. Soft error rate (SER) of each generation microprocessor increases gradually due to the exponential growing number of transistors per chip [1, 2]. Soft errors have become a great challenge to microprocessor design.

For regular storage structures (such as memory and cache), researchers can simply utilize parity and error correction codes to deal with soft errors, but these methods are not suitable for microprocessor core due to the high area and power overhead. Recently, many soft error detection and recovery techniques at the architecture-level or the microarchitecture-level have been proposed based on temporal or spatial redundancy, such as redundant multithreaded used in simultaneous multithreaded processors or chip multiprocessors [4, 5]. These methods can provide high fault coverage but still with significant performance degradation. Mukherjee et al. [6] showed that redundant multithreading techniques brought more than 30% performance penalty.

Strict redundant execution provides full fault detection during a program’s lifetime, and the fault detection rate will be very high, however, not all soft errors result in program failures. Prior work reports about 85% soft errors can be masked at different levels of microprocessors [7]. Mukherjee et al. [8] first proposed Architecture Vulnerability Factor (AVF) to describe the soft error masking rate. AVF is the probability that a soft error in a structure causes an external observable program failure. The higher AVF, the more the structure is vulnerable to soft errors. AVF varies significantly across workloads and individual structures, which means a microprocessor is more or less vulnerable to soft errors during executing a program [3, 9, 22]. As we can activate heavy-weight protection schemes during the high vulnerable periods and use light-weight protection schemes during less vulnerable periods, strict redundant execution is not necessary. Partial redundancy techniques [10] also have been proposed to alleviate performance degradation by only redundantly executing during a cache miss or low IPC periods. However, without quantitative guidance like AVF, these schemes just opportunistically improve reliability.

To meet the system reliability budget with minimum performance overhead, we utilize AVF as guidance to make a trade-off between reliability and performance. We compute the AVF online for various structures and determine whether to activate protection schemes for reliability enhancement according to the computed AVF. For online AVF computing, three key questions need to be answered: (1) how to choose the length of the computing interval; (2) how to online compute AVF efficiently; (3) when to activate protection schemes.

In order to solve these problems, we propose an occupancy-based method to online compute and predict AVF for issue queue (IQ), reorder buffer (ROB), and load/store queue (LSQ). We modify a validated execution-driven simulator to verify our method. Experimental results demonstrate that our method can efficiently compute AVF for these three structures within and across different hardware configurations.

This paper makes the following contributions:
We propose an occupancy-based online AVF computing method. Our method can efficiently compute different structures’ AVF at fine-granularity. In addition, our method can compute AVF across different hardware configurations.

We then integrate a history-based algorithm with the online AVF computing method to predict the AVF, and use the prediction result to guide when a protection technique should be activated.

The remainder of the paper is organized as follows. Section 2 provides the background of AVF and related work. Section 3 describes our experimental methodology. Section 4 provides our online AVF computing technique and AVF prediction algorithm in detail. Section 5 presents the experimental results and Section 6 summaries our work.

II. BACKGROUND AND RELATED WORK

In this section, we first review the definition of the architectural vulnerability factor, and then introduce some related work.

A. Architecture Vulnerability Factor

To estimate a structure’s vulnerability to soft errors, the first thing is to analyze its raw error rate at circuit level. Raw error rate is the rate that a structure encounters soft errors. Due to micro-architecture and architecture level masking, not all soft errors lead to external visible program failures. For a specific structure, its mean time to failures (MTTF), or the number of failures in time (FIT) is determined by the following equation [11]:

$$MTTF = \frac{1}{FIT} = \frac{1}{\lambda \times AVF}$$

where $\lambda$ stands for raw error rate, AVF stands for the soft errors masking rate. Raw error rate is determined by process technology and circuit design and will not change greatly with the technology scaling, so AVF is the main factor to determine a structure’s MTTF.

Many AVF computing methods have been proposed [3, 7, 8, 9]. One method is realized by counting the residency time of architectural correct execution (ACE) bits in a structure [3]. The ACE bits are those bits that will lead to external visible output if been upset, other bits are un-ACE bits. A hardware structure’s AVF during an interval can be expressed as [8]:

$$AVF = \frac{\sum \text{ACE bits in an instruction}}{\text{Total bits in the structure}} \times \frac{\text{residency cycles}}{\text{interval cycles}}$$

Several studies employ the above equation to compute AVF for microprocessor structures with detail performance simulators [3, 8, 9]. An alternative method to compute AVF is through fault injection [7]. Faults are injected in a microprocessor’s circuit-level model. After injecting a fault, the microprocessor’s architecture state will be analyzed to check whether the injected fault results in a failure or not. The percentage of faults causing program failures will be considered as the AVF.

All of the above-mentioned methods are implemented offline. The purpose of offline AVF computing is to analyze different structures’ vulnerability to soft errors at early design stage, and then utilize the information to guide microprocessor design. With detailed analysis, offline AVF computing methods can get high accuracy results, but these heavy-weight methods are not suitable for runtime tuning performance and reliability. Online AVF computing methods are needed in this case.

B. Related Work

Prior work has presented several online AVF computing methods from various aspects.

Li et al. [12] proposed a technique to online estimate microprocessor structures’ vulnerability. They utilize a modified fault injection and propagation scheme to compute AVF. To make the results have statistical significance, they inject thousands of faults to complete a computation which needs millions of cycles. Hence the approach is not suitable for fine-granularity reliability tuning.

Walcott et al. [13] studied the relationship between AVF and microarchitecture metrics (such as IPC and cache miss rate), and established several equations to estimate AVF using linear regression analysis for different structures. As the parameters in equations are generated under a certain hardware configuration, the runtime calculated AVF will be inaccurate if the online hardware configuration is different from the original one.

Soundararajan et al. [14] employed AVF as a constraint to satisfy system reliability. They only estimate the AVF for ROB per-cycle by analyzing ACE bits. If ROB AVF exceeds the system reliability threshold, protection methods (such as instruction throttling) should be used to reduce the number of ACE bits in ROB and then satisfy system reliability goal.

In this work, we propose an occupancy-based method to compute AVF online. Through adding a little extra hardware, we can analyze the occupancy per-interval (hundreds or thousands of cycles) and estimate AVF for IQ, ROB, and LSQ during a program execution. Then the AVF is predicted for the next interval with its history information. Whether a protection scheme will be activated or not is based on the prediction results. Only when the predicted AVF exceeds the system reliability threshold, the fault tolerance scheme will be activated. Our online computing and predicting technique can make a good trade-off between performance and reliability.

III. EVALUATION METHODOLOGY

All of our experiments are conducted on the Sim-Alpha simulator [15]. Sim-Alpha is a validated execution-driven simulator for Alpha 21264 processor. Prior work [16, 17] has shown that the simulation accuracy for Sim-Alpha is very close to a real processor. In this work, Sim-Alpha was heavily modified to support online AVF computing and prediction techniques. We use all the twelve SPEC CPU2000 integer benchmarks to evaluate our method. Since the simulator cannot accurately simulate the float point pipeline [15], the float benchmarks are not included. All the integer benchmarks are compiled for the Alpha ISA. In order to reduce simulation time, we use the Simpoint tool [18] to pick the most representative simulation point for
these benchmarks. Each benchmark is fast-forwarded to its representative point before the detailed performance simulation takes place. Each benchmark is evaluated for 100 million instructions using the reference input set. Table 1 summarizes the base configuration of the simulator. We adjust the size of IQ, ROB, or LSQ in our experiments to reveal the independence of our method to various hardware configurations.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Stages</td>
<td>7</td>
</tr>
<tr>
<td>Fetch/Decode/Issue/Commit Width</td>
<td>4/4/4/11</td>
</tr>
<tr>
<td>Branch-Predictor Type</td>
<td>Hybrid, 4K global + 2-Level 1K + 4K choice</td>
</tr>
<tr>
<td>Issue Queue Size</td>
<td>20</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>80</td>
</tr>
<tr>
<td>Load/Store Queue Size</td>
<td>32/32</td>
</tr>
<tr>
<td>Integer ALUs</td>
<td>4, 1-cycle latency</td>
</tr>
<tr>
<td>Integer Multipliers/Dividers</td>
<td>4, 7-cycle latency</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64KB, 2-way, 64B line-size, 1-cycle latency</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64KB, 2-way, 64B line-size, 3-cycle latency</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>2MB, direct mapped, 64B line-size, 7-cycle latency</td>
</tr>
<tr>
<td>1-TLB/D-TLB</td>
<td>128-entry, fully-associative</td>
</tr>
</tbody>
</table>

IV. AVF COMPUTING AND PREDICTING ARCHITECTURE

In this section, we first give an overview of the proposed method and discuss how to choose the length of the computing interval, then we describe the AVF computing and predicting method in detail, finally we discuss the limitations of our method.

A. Overview

Figure 1 shows a typical out-of-order superscalar pipeline. The gray components (extra bits and the AVF computing and predicting unit) are added to support our AVF computing and predicting infrastructure. We mainly focus on three structures: IQ, ROB, and LSQ. These structures play an important role in modern microprocessors. If any of them is stroked by a soft error, the probability causing an external visible program failure is very high.

For our occupancy-based computing method, the AVF computing unit records the occupancy information of different structures every cycle. At the end of each interval, it calculates all three structure’s AVF for the present interval meanwhile gives the prediction AVF for the next interval. A protection scheme will be enabled if the predicted AVF exceeds the system reliability threshold. With the dynamically tuning, the proposed method makes a good trade-off between reliability and performance.

B. Interval length of AVF computing

The main purpose of online AVF computing is to find when a microprocessor is most vulnerable to soft errors during a program execution. A structure’s AVF changes dynamically due to the varying number of ACE bits and their residency time. Therefore, the first question we should answer is how to choose the length of a computing interval. An interval length can be as long as the whole execution time of a program, or as short as just one cycle. If the interval length is too long, some detail information during a program execution will be lost. If the interval length is too short, the AVF will be too sensitive to the behavior changing of a program and will frequently force the pipeline to stall, consequently the performance overhead will be unacceptable. Figure 2 shows the change of the IQ AVF during executing benchmark crafty with different interval lengths. The black curve represents the IQ AVF computed every 1000 cycles, the gray curve for 200 cycles and the dotted back curve for 5000 cycles. After trying various interval lengths, we choose an interval length of 1000 cycles to make a trade-off.

C. Online AVF Computing

The second question we should answer is how to online compute AVF efficiently, as online computation should bring little performance penalty to the microprocessor. According to Equation 2, counting ACE bits in each instruction is a key step during offline AVF computing. Accurately differentiating ACE bits and un-ACE bits takes a long time, which is a tough job even for an offline AVF analysis method, not to mention an online one.

In our occupancy-based computing method, we try to solve this problem from another aspect. We first assume all bits in each structure are ACE bits, so the upper bound AVF of a structure is equal to its occupancy during an interval. For example, we assume the IQ size is 32 entries. At a specific cycle, if 16 entries in IQ have been occupied by instructions, then the upper bound AVF of IQ is 50%. It means if a soft error happens at that cycle, the probability causing an external error is 50%. The above assumption gets
a conservative AVF as all un-ACE bits are taken as ACE bits during the computation.

We then refine the result by exploiting the characteristic of different instructions. According to a prior study [8], the executed instructions in any program can be grouped to three categories: ACE instructions, dynamic dead instructions, and NOP instructions.

ACE instructions are those if been corrupted will affect the program final output, and all bits in ACE instructions are ACE bits. Dynamic dead instructions are those whose results will not be used by other instructions, or those whose results will only be used by dynamic dead instructions. Dynamic dead instructions have no effect to program final output. For online AVF computing method, it is hard to differentiate which instructions are dynamic dead instructions, as those instructions are dependent on the future use of their results.

NOP instructions are mainly used to reduce the memory access time. They do not make any useful operations and will not affect the architecture state. Only the opcode in NOP instructions are ACE-bits. Fahs et al. [20] reported there were 11% NOP instructions in SPEC CPU2000 integer benchmarks using Alpha instruction set, while Choi et al. [21] reported 27% NOP instructions using an Intel Itanium processor. As NOP instructions make little contribution to AVF computing, we can get a more accurate AVF result if we exclude NOP instructions when calculating a structure’s occupancy.

For each entry in IQ and ROB, we add an extra bit to indicate whether an instruction in it is NOP or not. An instruction’s opcode can be obtained at the decode stage, if it is a NOP instruction, the corresponding extra bit is set to logic value one, and the instruction will not be counted during the occupancy computation. The AVF computing unit records the occupancy information of IQ and ROB every cycle and calculates the AVF for them at the end of each interval.

For LSQ, only the load/store instructions will be issued to it. Therefore, if a soft error affects one entry in LSQ, the probability to change a memory address or an instruction result is very high, which may cause an un-recoverable failure. In our computation, all instructions stored in LSQ will be considered as ACE instructions, so the LSQ AVF is equal to its occupancy conservatively.

With the occupancy-based method, we can easily compute different structures’ AVF online. The main advantage of our method is that we do not need a complex mechanism to analyze ACE bits and un-ACE bits, and our method only has small area and performance overhead. In order to guide when to activate protection schemes, we integrate a prediction algorithm in our computing method.

D. AVF Predicting

To improve the system reliability at low performance overhead, we should compute the AVF for the present interval and predict it for the future interval. Hence, after computing the present interval’s AVF, we need a prediction algorithm to estimate the AVF for the next interval.

Program usually exhibits strong locality [19], thus if the present interval’s AVF has been computed, its next interval’s AVF will approximate to the value with high probability. We can predict the next interval’s AVF based on the latest interval’s AVF. Following we introduce two history-based prediction algorithms that can be integrated with our online computing method.

The first algorithm is a last-value based algorithm. It assumes the program executes in a stable phase and the present interval’s behavior will repeat. The predicted AVF for the next interval is simply the last measured value.

The last-value based algorithm just uses one interval’s information. If we utilize several intervals’ AVF, the predicted result will be more accuracy. Our second algorithm records the latest three intervals’ AVF, and use the average of the three intervals’ AVF to indicate the AVF for the next interval.

Figure 3 shows the IQ AVF and ROB AVF for benchmark crafty. The computing interval length is 1000 cycles. The gray and black curves indicate the AVF results given by the first and the second prediction algorithm, respectively. The dotted curve indicates the AVF results computed by an offline computing method Sim-SODA [3]. We can see the predicted AVF given by the second prediction algorithm is closer to the offline result than the first algorithm, hence the second prediction algorithm provides a more accurate value to guide fault tolerance. Therefore, we combine this algorithm with our online computing method in later experiments.

The last question we need to answer is when to activate the protection scheme to improve system reliability. We exploit the AVF to make an appropriate trade-off between reliability and performance. Protection schemes can be
activated either when one structure’s AVF exceeds the system reliability threshold or when all three structures’ AVF exceed the system reliability threshold. Prior research has presented several protection techniques, such as instruction throttling and selective redundancy [14]. We can utilize any of them to improve system reliability. While combining these techniques with our method is not within the scope of this paper, it is important to know that AVF guided protection methods can compensate the shortcomings of the conventional partial redundancy techniques.

E. Limitations

The major limitation of our method is that the computation accuracy is not as high as the offline computing method. As a time-consuming analysis is not allowed during program executing, it is hard to accurately identify all the ACE bits online. Though our method adds some extra hardware, including 32 flag bits for IQ, 80 flag bits for ROB, and the AVF computing unit, the area overhead is negligible.

V. EXPERIMENTAL RESULTS

Following we present our experimental results. Figure 4 shows the average AVF for IQ, ROB, and LSQ computed by our online computing method and an offline scheme SIMSODA [3] during executing all twelve integer SPEC CPU2000 benchmarks. As can be seen from Figure 4, the vulnerability of different microprocessor structures to soft errors varies across individual structures and benchmarks. IQ AVF is higher than other two structures for most benchmarks. That is because in Sim-Alpha, IQ is separated into an integer IQ and a floating point IQ, which make integer IQ have higher occupancy than other two structures.

The absolute AVF differences between our online computing method and the offline method are 0.10, 0.01, and 0.039 for IQ, ROB, and LSQ, respectively. These discrepancies are caused by dynamic dead instructions since we have not differentiated them from ACE instructions. As the number of entries in IQ is less than the entries in ROB and LSQ, dynamic dead instructions make more contribution to the IQ AVF computing and little to ROB and LSQ, that’s why the absolute difference of IQ is larger than the absolute differences of ROB and LSQ.

For most benchmarks, online computed AVF is larger than offline method. There is an exception that online computed ROB AVF and LSQ AVF are a little smaller than that of the offline results during executing benchmark gzip. That reason is when executing benchmark gzip, the percentage of NOP instructions reaches up to 40%. NOP instructions are not considered in our online computing method, but the opcode in each NOP instruction still makes contribution for offline AVF computing, which makes the AVF results computed by our method a little smaller than that of the offline method.

We further analyze three structures’ AVF during executing different benchmarks with an interval length of 1000 cycles. Figure 5 shows the changing AVF of the three structures when executing benchmarks crafty and gap (we do not list all of the benchmarks as the space limitation). When executing benchmark crafty, three structures’ AVF change frequently at a wide range. IQ AVF is larger than ROB AVF and LSQ AVF, which is consistent with our previous analysis. While for benchmark gap, on the contrary, the AVF fluctuation is very small at most time. Different structures’ AVF vary from a large range during different intervals and heavily depend on the executing benchmarks, but there is no guarantee that any of the three structure’s AVF will be greater than other two structures’ all the time.

![Figure 4. AVF for online and offline computing.](image)

We also change the base configuration of our simulator. Table 2 lists several configurations for our simulator. Here config1 describes the base configuration. For other three configurations, we change one structure’s size each time to analyze the effect to AVF. Figure 6 shows the AVF results of different configurations. For config2, the ROB size decreases from 80 to 10 entries, ROB AVF increases sharply, while IQ AVF and LSQ AVF both decrease significantly. That is because ROB has become system bottleneck, and the number of instructions can be executed reducing sharply. For config3, IQ AVF is nearly half of the value of config1 (as shown in Figure 4), while ROB AVF and LSQ AVF are almost the same. For config4, LSQ AVF increases significantly. We can see a structure’s size has great impact to its AVF, and our online computing method still can compute AVF with high accuracy. Our method utilizes a structure’s occupancy to guide AVF computing, it is a generic method and is independent of the microprocessor’s configuration.

VI. CONCLUSIONS AND FUTURE WORK

This paper has presented a method to online compute and predict AVF at the interval granularity. Our method exploits the AVF varying feature to adaptively tune reliability and performance. The highlight of our method is that we do not need a complex mechanism to analyze ACE bits and un-ACE bits. We evaluate our method with an execution-driven simulator Sim-Alpha running SPEC CPU2000 integer benchmarks. Experimental results show that our method can efficiently compute the AVF for different structures under several hardware configurations. For IQ, ROB, and LSQ, the average absolute differences between our method and a previous offline AVF computing method are 0.10, 0.01, and 0.039, respectively. For the future work, we plan to compute AVF for other hardware structures and to combine our method with a protection technique.
ROB OFF IQ OFF eon
performance Characterization of a
LSQ OFF vpr
2.08
2.04
Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on March 28,2010 at 02:14:07 EDT from IEEE Xplore. Restrictions apply.