Online Estimation of Architectural Vulnerability Factor for Soft Errors [*]

Xiaodong Li[†], Sarita V. Adve[†], Pradip Bose[‡], Jude A. Rivers[‡]
[†]University of Illinois at Urbana-Champaign [‡]IBM T.J. Watson Research Center
{xli3,sadve}@uiuc.edu {pbose,jarivers}@us.ibm.com

Abstract

As CMOS technology scales and more transistors are packed on to the same chip, soft error reliability has become an increasingly important design issue for processors. Prior research has shown that there is significant architecture-level masking, and many soft error solutions take advantage of this effect. Prior work has also shown that the degree of such masking can vary significantly across workloads and between individual workload phases, motivating dynamic adaptation of reliability solutions for optimal cost and benefit. For such adaptation, it is important to be able to accurately estimate the amount of masking or the architecture vulnerability factor (AVF) online, while the program is running. Unfortunately, existing solutions for estimating AVF are often based on offline simulators and hard to implement in real processors.

This paper proposes a novel way of estimating AVF online, using simple modifications to the processor. The estimation method applies to both logic and storage structures on the processor. Compared to previous methods for estimating AVF, our method does not require any offline simulation or calibration for different workloads. We tested our method with a widely used simulator from industry, for four processor structures and for 100 to 200 intervals of each of eleven SPEC benchmarks. The results show that our method provides acceptably accurate AVF estimates at runtime. The absolute error rarely exceeds 0.08 across all application intervals for all structures, and the mean absolute error for a given application and structure combination is always within 0.05.

1 Introduction

Soft errors (or single event upsets or transient errors) caused by alpha particles from packaging materials or high energy particle strikes from cosmic rays can flip bits in storage cells or cause logic elements to generate the wrong result. Such errors have become a major challenge to processor design. Previous research shows that soft error rate (SER) per chip is growing substantially, mainly due to the fast growth of the number of transistors on the chip [3, 11, 15].

If a particle strike causes a bit to flip or a piece of logic to generate the wrong result, we refer to it as a *raw* soft error event. Fortunately, not all raw soft errors cause the processor to fail. In a given cycle, only a fraction of the bits in a processor storage structure and some of the logic structures will affect the final program output. A raw error event that does not affect these critical bits or logic structures has no adverse effect on the program outcome and is said to be *masked*. For example, a soft error in the branch prediction unit or in an idle functional unit will not cause the program to fail. Research has shown that there is a large masking effect at the architecture level [1, 4, 10, 18, 20].

Mukherjee et al. [10] introduced the term architectural vulnerability factor (AVF) to quantify the architectural masking of raw soft errors in a processor structure. The AVF of a structure is effectively the probability that a visible error (failure) will occur, given a raw error event in the structure [10]. The AVF can be calculated as the percentage of time the structure contains *Architecturally Correct Execution* (ACE) bits (i.e., the bits that affect the final program output). Thus, for a storage cell, the AVF is the percentage of cycles that this cell contains ACE bits. For a logic structure, the AVF is the percentage of cycles that it processes ACE bits or instructions.

For a large class of systems and workloads, including those studied here, the AVF of a structure directly determines its mean time to failure (MTTF) [5] – the smaller the AVF, the larger the MTTF and vice versa. It is therefore important to accurately estimate the AVF in the design stage to meet the reliability goal of the system. Many soft er-

[*]This work is supported in part by an IBM faculty partnership award, the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grant NSF CCF 05-41383, an OpenSPARC Center of Excellence at the University of Illinois at Urbana-Champaign supported by Sun Microsystems, and an equipment donation from AMD.

1063-6897/08 $25.00 © 2008 IEEE
DOI 10.1109/ISCA.2008.9

ror protection schemes have significant space, performance, and/or energy overheads; e.g., ECC, redundant units, etc. Designing a processor without accurate knowledge of the AVF risks over- or under-design. An AVF-oblivious design must consider the worst case, and so could incur unnecessary overhead. Conversely, a design that under-estimates the AVF would not meet the desired reliability goal.

Recently, there has been significant work motivating the need to estimate AVF at runtime as well. Studies have shown significant variation in the AVF value across different applications and within different phases of the same application [6, 17]. Thus, depending on the workload, the processor may be more or less vulnerable at different times. This observation creates new opportunities to reduce the soft error protection overhead while meeting the MTTF goal. If we are able to estimate AVF in real-time accurately, we can adjust the protection scheme based on the current AVF value. We can have more protection during highly vulnerable periods and less protection during less vulnerable periods, minimizing performance and/or energy overhead. For example, Soundararajan et al. [16] propose to use the AVF input to control instruction throttling and selective redundancy schemes. In this case, a real-time online AVF estimation is a must since a slow offline method will not be able to give timely input to the control logic.

There is some previous work that provides online AVF estimation; however, it is either dependent on extensive offline workload analysis [17] or targeted to a single structure [16] (see Section 2). In general, estimating AVF online is a challenging task since the complex computation used in offline analysis is not feasible in real-time. The AVF for many structures depends on many factors that are hard to measure and observe. For example, the AVF of the floating point unit depends not only on its utilization, but also on variables such as the percentage of dead values and speculative instructions. For storage structures, AVF estimation is even more difficult. It may be intuitive to think that the number of reads/writes to a storage structure may be correlated with the AVF of this structure. However, it is easy to construct two read/write sequences that have the same number of reads and writes, but very different AVF values.

In this paper, we describe a general online method to estimate AVF for a variety of structures (including logic and storage structures) without need for extensive offline workload analysis. Our approach is motivated by offline (complex) AVF estimation approaches. Specifically, a common method for offline estimation is to inject an error in a low-level simulator and determine whether it results in program failure. Many such injections are performed, and the AVF is calculated as the fraction of such injections that lead to failure. Our online estimation method effectively seeks to perform error injection while the program is running in production mode and uses the program execution to determine whether the error will result in failure. Of course, we cannot actually inject an error into a production run. We therefore introduce some additional *error bits* through the processor pipeline that can emulate the generation and propagation of an error.

To estimate the AVF of a structure, we emulate the injection of an error in the structure by setting its error bit to 1. An instruction touching this structure then propagates the injected error to its destination and so on. In this way, an error is propagated by the executing program. Our algorithm waits a fixed number of cycles to determine if the error could (potentially) result in program failure. Multiple such injections are done and as with offline error injection, the fraction that are determined to potentially result in failure provide an estimate of the AVF of the structure.

Our method depends on two key parameters to get an accurate estimate of AVF: (1) how many times to inject an error, and (2) after injecting an error, how long to wait to see if the error will cause a program failure. Setting these parameters too high can result in an estimation procedure that lasts too long and does not adequately track the AVF changes in the program. Setting them too low can result in incorrect estimates. We use analytical and experimental methods to determine these two parameters.

To evaluate our method, we implement it in a simulator and perform experiments to estimate the AVF for both logic and storage structures (integer ALU, FPU, instruction queue, and register file) for 100 to 200 intervals in each of eleven SPEC benchmarks. In order to validate our results, we compare against SoftArch, which is a more detailed (but complex) offline AVF estimation method (see Section 2). Our results show that our method generates very similar results to SoftArch. The absolute difference in AVF estimated by the two methods rarely exceeds 0.08 across all application intervals and structures studied. The mean absolute difference is less than 0.05 for any given application and structure. Further, we also compare an intuitive and simple AVF estimation method that uses the utilization of logic structures as a proxy for their AVF (an analogous extension of such a method for storage structures is not clear). We show that compared to our method, this simple method shows significant inaccuracies relative to SoftArch, providing evidence for the need for the hardware support required by our method. Overall, our results show that our novel method for online estimation of AVF is both accurate and robust in a variety of situations.

2 Related work

There have been several studies on estimating the AVF [10, 6, 19, 17, 16]. However, most studies estimate AVF using offline analysis with complex simulators [10, 6, 19]. This offline estimation is a complex process, requiring many resources to track values and instructions as they travel through a processor. Normally only a limited number of instructions can be analyzed in a reasonable amount of time. These methods are therefore not suitable for online real-time AVF estimation.

There has been some work on estimating the AVF in real time [16, 17]. Walcott et al. [17] apply statistical analysis using a detailed simulator to analyze the AVF behavior. Then they use regression to explore the relationship between AVF and various micro-architecture level variables such as structure occupancy, number of instructions executed, etc. After running the regression offline for certain workloads, the correlation coefficients between AVF and each micro-architecture variable are established. Since the micro-architecture variables are observable, the AVF value can be estimated through them. This method can potentially be implemented to estimate the AVF online; however, it requires heavy offline simulation and calibration for different workloads. It is not clear that the parameters calibrated for one set of workloads will give accurate estimation for another set.

Soundararajan et al. [16] propose a method to estimate AVF for the reorder buffer (ROB) in the processor. This method determines the AVF by estimating the occupancy of the instruction queue. The occupancy of the instruction queue is in turn estimated by counting the number of instructions that are dispatched or retired. This method can be implemented online, but is limited to a single structure. For example, it is hard to extend the same method to estimate the AVF for the register file.

There have been at least two major works that use bits similar to our error bits, but for different purposes. First, Weaver et al. [20] proposed the π bit to address false detected errors. Every instruction and register entry is associated with a single π bit. When an error is detected (e.g., via parity), the affected instruction's π bit is set by the instruction queue and the instruction is allowed to progress down the pipeline. When the affected instruction reaches the commit point, if it is determined to contribute to correct program outcome, a machine check error is raised; otherwise the set π bit is ignored. Second, the poison bit [12] and the analogous NaT bit of the Itanium architecture [14] are used to track deferred speculative exceptions. Our use of the error bits is different – we use them to estimate the AVF due to soft errors. Nevertheless, the hardware support required for all of these techniques is likely to be similar.

Finally, we use the offline estimation technique of SoftArch to provide the reference AVF used to validate our online estimator [6]. SoftArch uses simple probability theory to model the error generation and propagation process in a processor. It keeps track of the error probability of the different values in the processor. Integrated with a program timing simulator, SoftArch also identifies which values could affect program outcome and when that might happen. Using the error probability and the timing information, SoftArch is able to determine the MTTF and the AVF value for a workload. Our method differs from SoftArch in that we estimate the AVF online in hardware while the program is running. Instead of using complex probability calculations and detailed error tracking that are suited for offline analysis, we use a simple mechanism that can be efficiently realized in hardware. Whereas SoftArch (probabilistically) tracks every raw error that could be generated in the processor, our method takes an online statistical fault injection based approach to estimate the AVF in real-time.

3 AVF Estimation Algorithm

This section describes our online AVF estimation algorithm. We first give an overview of the algorithm and then describe the details, including the hardware support needed, overhead, and limitations.

3.1 Overview of the Algorithm

The main idea of the algorithm is to associate error bits with structures, inject an error by setting an error bit to 1, use the program execution to propagate the error, determine if the error (potentially) causes failure, and repeat another injection. The percentage of injections that cause failure is the estimated AVF. We first illustrate the working of the algorithm with an example small program segment below.

```
1.    r1 + r2  = r3
2.    r1 - r2  = r4
3.    r2 + r4  = r3
      ...
4.    r3 + r4  = r5
5.    store r5 to address r4
      ...
6.    load r5 from address r4
      ...
7.    r5 + r6  = r7
8.    Branch if r7 = 0
```

First, let us assume that we want to measure the AVF of the register file. Suppose at some cycle after completing line 1 but before executing line 3, we inject an error in register `r3` by setting its error bit to 1. When line 3 is executed, the value of `r3` is overwritten. Thus, its error bit is overwritten as well by an "or" of `r2`'s and `r4`'s error bits. Since neither of those source registers has an error, `r3` no longer has an error, and so the injected error bit has disappeared. After waiting for a pre-determined number of cycles, say M, we see no processor failure. This example in particular shows how our scheme correctly handles dead values.

Next, assume that at some cycle before executing line 4, we inject an error into `r4`. This error bit will propagate to the result register `r5`. Next we see a store writing an erroneous value (`r5`). As discussed later, we assume errors in retiring stores can cause program failure; therefore, when that store retires, we update a failure counter. So far, we have injected two errors and one of them causes program failure. If we calculate AVF at this time, it would be 50%.

Next, let us examine how our scheme measures the AVF for a functional unit like the integer ALU. Suppose at the cycle when the load instruction at line 6 is executed, we inject an error into the ALU by setting its error bit. Since the ALU is not used during that cycle, the error bit will not propagate to other structures. Thus, the error is masked. Next, assume we inject an error into the ALU at the cycle when line 7 is executed. The ALU is used during the cycle to calculate `r7`. Thus, by our approach, the injected error propagates into `r7`. Now `r7` has its error bit set to 1 which later propagates to the branch instruction. When instruction 8 is executed, we note that it is an erroneous branch. As discussed next, we assume erroneous branches can potentially cause program failure and update a failure counter when this branch hits retirement.

Our algorithm tracks only one error at a time; injecting multiple errors simultaneously will make the algorithm too complex for at least two reasons. First, different errors could merge and this could obscure the true nature of the structure's vulnerability information. For example, when two values $x1$, $x2$ are added up together, two separate errors in $x1$ and in $x2$ could combine into one new error and the original error information is lost. Second, one error could propagate into several values and they might all lead to program failures. We should count them as just one failure since they are all caused by the same error source. Tracking such information requires complex hardware and logic. Thus, we only inject one error at a time and clear all current errors before injecting the next error.

The full algorithm is summarized as **Algorithm 1** and subsequent sections elaborate on the details. We first discuss the cases where we assume the injected error results in program error. Then we discuss the two predetermined variables N and M that are used to control how many times to inject errors and how long to wait after each error injection (to determine potential failure) respectively. We then discuss the hardware support required and the other overheads, and finally the limitations of our method.

3.2 Determining Potential Failure

In reality, an error causes program failure only if it propagates to the program output. Unfortunately, we cannot perform this ideal assessment of failure for two reasons. First, waiting for propagation to the output could take too long for our technique. That is, it would limit the number of error injections we could monitor in a reasonable amount of time. Second, since our method does not disturb the actual program execution, any changes that would occur in the control flow of the program due to the injected errors are not seen. For these reasons, we conservatively consider an error to potentially cause failure if it propagates to the following points: (1) an output or store instruction retires with an error bit set in the retirement buffer entry (propagating to output requires propagating to a store), (2) a load instruction retires with an error bit set (an erroneous load could cause a visible failure), and (3) a control flow instruction retires with an error bit set (it could lead to an unmodeled change in program flow causing failure that would not otherwise be detected). Note that waiting until retirement to flag a failure ensures that misspeculated instructions do not flag failures.

Algorithm 1 Algorithm to estimate AVF for a structure

```
Set the counters $injectionCount = 0$ and $failureCount = 0$
while $injectionCount < N$ ($N$ is a predetermined threshold) do
    Inject an error into the structure by setting its error bit to be 1. For a storage structure that contains many entries, randomly choose one to inject an error.
    For the next $M$ cycles ($M$ is predetermined), propagate the error bits according to the execution. If a bit propagates to certain predefined failure points, set the processor failure bit.
    If the processor failure bit is set, $failureCount = failureCount + 1$.
    Clear all error bits in the processor.
    $injectionCount = injectionCount + 1$.
end while
AVF = $\frac{failureCount}{injectionCount}$
```

3.3 Determining N – the number of error injection samples needed

In this section, we show that Algorithm 1 gives an unbiased estimation of the AVF and, more importantly, we derive an equation to determine the number of samples needed to get an accurate estimation.

Algorithm 1 gives an unbiased estimator.

An error injected in a structure is either masked or not masked with probability AVF and 1-AVF respectively. We introduce a random variable X to model this process: $X = 1$ if the error is not masked and $X = 0$ if the error is masked. X has the following probability mass function:

$$Pr(X = 1) = AVF, \qquad Pr(X = 0) = 1 - AVF$$

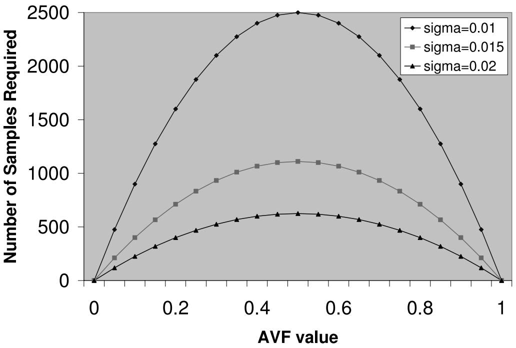


Figure 1. The number of samples N needed for different values of AVF and estimation precision of $\sigma_{\bar{X}}$.

Our algorithm seeks to estimate AVF which is the expectation of X or $E(X)$. It does this by determining the outcome of N error injections or by generating N samples of X, denoted X_1, X_2, ..., X_N. The algorithm estimates AVF as the mean of these samples denoted $\bar{X} = \frac{X_1+X_2+...+X_N}{N}$. If the N samples are independent and identically distributed (i.i.d.), then it can be shown using simple probability theory that $\bar{X}$ is an unbiased estimator for $E(X)$ since $E(\bar{X}) = E(X)$ [13].

Independence of the samples can be ensured using random sampling; i.e., by using a random number generator to determine the error injection time. Since a random number generator is complex to implement in hardware, in our experiments, we injected errors at fixed length intervals. Although we expect that small time-scale variations in the workload behavior will effectively introduce enough randomization, this is an approximation and potential source of inaccuracy in our estimation. In the following, we assume that the samples are identically distributed for simplicity, but relax this assumption at the end of the section.

Determining N for an accurate estimation.

To ensure that $\bar{X}$ is an accurate enough estimator of AVF, we analyze and bound the standard deviation of $\bar{X}$, denoted $\sigma_{\bar{X}}$, as follows. It is well-known that the standard deviation $\sigma_{\bar{X}} = \frac{\sigma_X}{\sqrt{N}}$ if all X_i are i.i.d. [13]. Thus, we can fix the number of samples, N, depending on the desired value of $\sigma_{\bar{X}}$ (i.e., the desired accuracy of the AVF estimate). Based on the above equation, we have

$$N = \frac{\sigma_X^2}{\sigma_{\bar{X}}^2} \qquad (1)$$

From the distribution of X, we know that $\sigma_X = \sqrt{AVF(1-AVF)}$, where AVF $\in [0,1]$. Thus, we can plot the desired value of N as a function of the AVF, given a desired precision (standard deviation) of the estimator. Figure 1 shows such plots for different values of $\sigma_{\bar{X}}$. In practice, the AVF value is unknown before the estimation, so we cannot directly use the plots to determine N. Instead, we note that the maximum possible value of σ_X is 0.5 corresponding to an AVF of 0.5. We substitute this value in equation 1 to derive a conservative upper bound for N. For example, for the estimation standard deviation to be less than 0.01, we need $N = 0.5^2/0.01^2 = 2500$ samples. Similarly, for $\sigma_{\bar{X}} < 0.02$, we need $0.5^2/0.02^2 = 625$ samples. In general, N can be chosen based on the needed precision. In this work, we choose $N = 1000$ since we empirically find it to be a good balance between the estimation precision and the simulation time.

Storage structures with multiple entries.

So far, our analysis of the AVF estimation of a component implicitly treats the component as a single entity. For a storage structure that contains many entries, we can view each entry as a (sub-)component and sample each entry. Assuming the structure has K entries, we define one random variable for each of the K entries and denote them as X^i, i in 1, 2,...K. The AVF of the structure is $\frac{\sum_{i=1}^{K} E(X^i)}{K}$.

Suppose we sample the entire structure N times. Ideally, for each sample, we would like to choose the entry to sample using a random number generator; however, that would be too expensive in hardware. As an approximation, we choose to sample the different entries in a round-robin fashion, resulting in N/K samples for each entry or each X^i. Our AVF estimator, $\bar{X}$, is the average of these N samples. This is an unbiased estimator for the AVF of the structure since $E(\bar{X})$ is the AVF.

Assuming the samples are independent and that all samples for an entry are i.i.d., we can show that [13] $\sigma_{\bar{X}} = \sqrt{\frac{\sigma^2_{X^1}+\sigma^2_{X^2}+..+\sigma^2_{X^K}}{N*K}}$.

In this formula, if we conservatively assume that all the σ_{X^i} are the maximum value of 0.5, it follows that $\sigma_{\bar{X}} < 0.5/\sqrt{N}$. Thus, even in this case, the bound for N is the same as for the single structure.

Relaxing the identical distribution assumption.

Above we also assume that all the samples are identically distributed. However, we know that workload behavior may change significantly over long intervals of time. If the estimation interval includes such large-scale changes, then we can think of the interval as consisting of multiple phases (each with its own AVF) and the AVF for the entire estimation interval to be the average AVF across all the phases.

Now the expectation of our estimation becomes $E(\bar{X}) = E(\frac{\sum_{i=1}^{N} X_i}{N}) = \frac{1}{N}\sum_{i=1}^{N} E(X_i)$, where $E(X_i)$ may be different for different i. If our samples are spread evenly over the entire estimation interval, then it follows that $E(\bar{X})$ is the AVF of the entire estimation interval. To achieve even sampling, we inject a new error every fixed time interval M over the entire estimation interval.

The standard deviation of the estimation now is $\sigma_{\bar{X}} = \frac{1}{N}\sqrt{\sigma^2_{X_1} + \sigma^2_{X_2} + .. + \sigma^2_{X_N}}$. σ_{X_i} may be different for different i. By conservatively assuming that σ_{X_i} takes its maximum value, $\sigma_{\bar{X}} < 0.5/\sqrt{N}$. This is exactly the same equation as with the i.i.d. assumption.

3.4 Determining M – the interval between successive error injections

Each time we inject an error, we need to wait to see if it can cause processor failure. The interval M that we need to wait is an important parameter in our algorithm. If we wait too long, it will take a long time for us to have a reasonable estimate for AVF. However, if the wait time is too short, a potentially unmasked error might not have propagated as a failure yet. Thus, we need to choose M so that it is large enough that most of the unmasked errors propagate as a failure (as defined above) during that period.

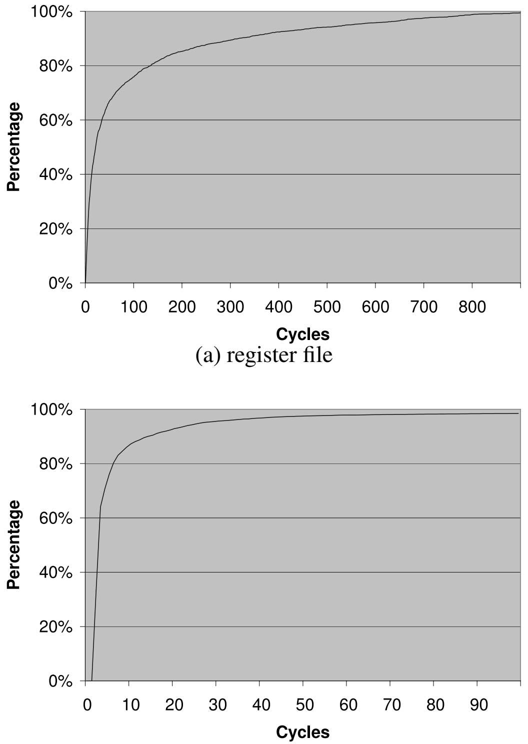
We empirically determine the appropriate injection interval length M using the error propagation time distribution in the processor. We inject errors into each structure of the processor and measure the time it takes for the errors to propagate to our predefined failure points. Figure 2 shows the cumulative distribution of these propagation times for the register file and FXU units for application *bzip2*.

Depending on the various latency parameters of the modeled processor and the workload characteristics, the distribution curves will change. For example, for the issue queue, an error injected into one of the entries may, in the case of a long latency cache miss, remain ”live” for a duration that is at least as long as the worst-case miss latency in the system. Different structures may also have different distribution curves. For example, we can see that the register file and the FXU have different distribution curves in Figure 2. Thus, the optimal choice of M depends on the structure, workload, and processor. Estimating the optimal M is therefore a complex process.

For our simulations, we choose M to be conservative so that the value covers all the workloads and the structures we study here, namely register file, instruction queue, FXU, and FPU. Based on the distributions observed for these structures, we choose M = 1000. We could have used a smaller M for some of our structures; however, even with M = 1000, we need only 1 million cycles to estimate the AVF (given N = 1000). Thus, for simplicity, we use $M = 1000$ for all the structures and workloads we study. Other structures may require larger values of M.

3.5 Hardware support and overhead

The processor contains storage and logic structures. For each storage entry such as a register in a register file or an issue queue entry, an error bit needs to be attached. For the bus, one extra line is needed to carry over the error bit when a value is transferred over the bus. For each logic structure like the FXU or FPU, an error bit will is required.



(a) register file

(b) FXU

Figure 2. The cumulative distribution for the time taken by an error to propagate to points of potential failure (defined in Section 3.2) for *bzip2*.

The scheme also needs the necessary hardware logic support to set and clear each error bit. We emulate the injection of an error to a given structure by setting its error bit to one. When the structure is used, its error bit needs to be propagated down the pipeline. For example, if the error bit of a storage cell is set to one, when the value in the cell is read, the error needs to propagate together with the value. If the value is overwritten, the error bit needs to be overwritten as well. If the error bit of a logic structure is set to one and this structure is active, the error bit will be attached to the output value. If the structure is idle, the error bit will not propagate further and is masked. If a logic structure takes more than one input, such as the ALU, “or” gates are needed to merge the error bits from each input.

Besides the error bits, the scheme also needs basic hardware counters to track the total number of errors injected and the number that (potentially) lead to processor failure.

The overhead of the scheme mainly comes from the setting and clearing of the error bits. The error bits require extra hardware. We need one bit for every 32-or 64-bit value; hence, the space overhead for storage entries is about 1-3%. For a logic structure however, we only need one bit for a given structure. We also need the necessary logic to keep track of how many failures have occurred and how many errors have been injected. This can be done using several basic counters. In addition, we need a counter to keep track of which storage entry or logic structure to inject next.

During program execution, the error bits propagate together with the values and should not cause any extra slowdown for the processor. Once in every $M * N$ instructions or so, the processor needs to do the accounting and calculate the AVF. Given that this is done typically once every (several) million instructions, the time overhead should be negligible.

3.6 Limitations

Our method also has several limitations. A major assumption of our method is that an error in the processor will propagate and cause program failure in a short period of time, currently less than several thousand instructions. Otherwise, the time it takes to estimate AVF will be much longer since M will need to be set to be a large number. Since we conservatively assume that values stored in memory are observable externally and thus can cause program failure, this assumption appears satisfactory for the structures we study. However, if we were to set the output instructions as failure monitoring points, then we may need to wait for longer periods, meaning that we may not be able to sample enough points. The downside of this is that we have to be very conservative in estimating when an error leads to failure.

Also, our method only depends on one run of the program and we are not able to simulate and track execution along incorrect paths invoked due to an error. Without this ability, we are left to defining the points of failure very conservatively.

Under the current scheme, we attach one bit for each value or instruction in the processor. Thus, our error injection granularity is limited to the full value or instruction. This means that we cannot distinguish between errors in different fields of a structure and cannot track which part of the instruction has error. This could be addressed by supporting multiple error bits per value or instruction, allowing errors to be injected at a finer granularity. Similarly, since we do not differentiate between bits constituting a given value, we conservatively assume that the value is wrong once any of its bits has an error. This prevents us from modeling detailed masking effects like logical masking.

Finally, the goal of this work is to develop an online AVF estimation algorithm. Our algorithm estimates the AVF for the past interval. Many processor adaptive control algorithms need the AVF for the future interval as the input. In order for our approach to be useful for controlling any processor adaptation, we need to integrate our method with an interval or phase prediction method. There has been much work on phase prediction. Our work can simply be combined with any phase prediction algorithm. For example, we could use a simple predictor which always predicts the next interval's AVF to be the same as the past interval.

Technology Parameters	
Process technology	90nm
Processor frequency	2.0 GHz
Processor Parameters	
Fetch rate	8 per cycle
Retirement rate	1 dispatch-group (=5, max) per cycle
Functional units	2 Int, 2 FP, 2 Load-Store, 1 Branch
Issue queue entries	FPU = 20, Load/Store/Integer = 36 Branch = 12
Integer FU latencies	1/4/35 add/multiply/divide (pipelined)
FP FU latencies	5 default, 28 div. (pipelined)
Register file size	80 integer, 72 FP
iTLB/dTLB entries	128/128
Instruction buffer entries	64
Memory Hierarchy Parameters	
L1 Dcache	32KB, 2-way, 128-byte line
L1 Icache	64KB, 1-way, 128-byte line
L2 (Unified)	1MB, 4-way, 128-byte line
Contentionless Memory Latencies	
L1/L2/Memory Latency	1 /20 /165 cycles

Table 1. Parameters for the simulated processor.

4 Experimental Methodology

To evaluate the accuracy of our AVF estimation method, we use the Turandot simulator [9]. Turandot was developed at IBM's T.J. Watson Research Center and is a trace-driven performance simulator that models the timing of the various pipeline stages of a modern out-of-order superscalar processor in detail [8]. As described in [8], Turandot was calibrated against a pre-RTL, detailed, latch-accurate processor model. Table 1 summarizes the parameters for the processor we simulate; these were chosen to roughly correspond to the POWER4 microarchitecture [7].

We implemented our AVF estimation algorithm in Turandot as described in Section 3 to estimate the AVF of the instruction queue (IQ), register file (REG), integer or fixed point functional units (FXU), and floating point units (FPU).[1]

[1] We were not able to collect data for TLBs since a reasonable M value required for effectively exercising them is close to 1 million cycles. Thus, to generate one AVF estimation requires a billion cycles of simulation, which made it difficult to collect a full set of results.

We evaluated our algorithm with eleven SPEC CPU2000 benchmarks. We used traces from the trace repository generated using the Aria trace facility in the MET toolkit [8], using the full reference input set. Sampling was used to limit the trace length to 100-200 million instructions per program. The sampled traces have been validated with the original full traces for accuracy and correct representation [2].

The value of the parameters M and N depend on the processor and compiler and should be carefully chosen. In our experiments, as we have mentioned in previous sections, we choose $M = N = 1,000$. Thus, we estimate an AVF value at the granularity of every $M * N = 1$ million cycles of an application. We refer to this as the *estimation interval* below. This gives us 100-200 AVF estimates (one for each distinct 1M cycle interval) for each application and each processor structure.

To validate the accuracy of our AVF estimates, we compare against the AVF reported by the SoftArch method [6]. As mentioned, SoftArch is a detailed soft error model that estimates the AVF offline with a lot of analysis. We use SoftArch since it is the best AVF estimation we have access to.

Additionally, to justify the full complexity of our method, we also compared its accuracy to that of a simpler, intuitive method. Specifically, for logic structures, it is intuitive to consider utilization as an estimation for the AVF (the higher the utilization, the higher the vulnerability to soft errors). The utilization of a logic structure can be easily estimated in hardware by counting the number of cycles it is busy out of all cycles. It is natural to use the utilization as a proxy for AVF since errors in the structure will be masked if the structure is idle and errors may not be masked when the structure is busy. An analogous concept is harder to extend to storage structures. We are not aware of any other general, workload-independent algorithm for online estimation of AVF of storage structures. Thus, in this study, we use a simple alternative (utilization-based) method only to estimate AVF for logic structures.

5 Results

Figures 3(a), (b), (c), and (d) show aggregate statistics to demonstrate the accuracy of our AVF estimation algorithm relative to SoftArch for the instruction queue, register file, FXU, and FPU respectively. The FXU and FPU figures also show the accuracy of the simple utilization-based estimation method relative to SoftArch (right bar for each application).

Below, by *absolute error* of an estimation method for a given application interval that contains 1 million cycles, we refer to the absolute difference between the AVFs reported by that method and by SoftArch. By *relative error* of an estimation method, we refer to $\frac{|Estimated\ AVF\ -\ SoftArchAVF|}{SoftArch\ AVF} * 100$. Also, we often refer to the SoftArch AVF as the *real AVF*.

The charts on the left side of Figure 3 give three statistics for the absolute errors. For each bar, the lowest (shaded) stack gives the mean absolute error (referred to as Mean) for the corresponding estimation method and application (averaged across the different 1M cycle estimation intervals for that application). The full height of the bar is the maximum absolute error, ignoring the top four errors to exclude unrepresentative outliers (referred to as Max). The middle stack is the standard deviation of the absolute error (referred to as Standard_Deviation).

Since AVF values can range only from 0 to 1, it is most meaningful to compare the absolute errors. Small absolute errors may be acceptable even if the relative error is large; e.g., an estimate of AVF=0.12 for a real AVF of 0.1 reflects a 20% relative error; however, it is unclear if this difference of 0.02 absolute error is practically significant. Nevertheless, the charts on the right side of Figure 3 provide the relative errors for reference.

For a more detailed look, we take two applications as examples and plot AVF values for them for each 1M cycle estimation interval for each structure in Figure 4. For each application, we show the AVF value calculated by SoftArch and the AVF value estimated by our method. For both the FPU and FXU, we also show the AVF calculated by the utilization-based method.

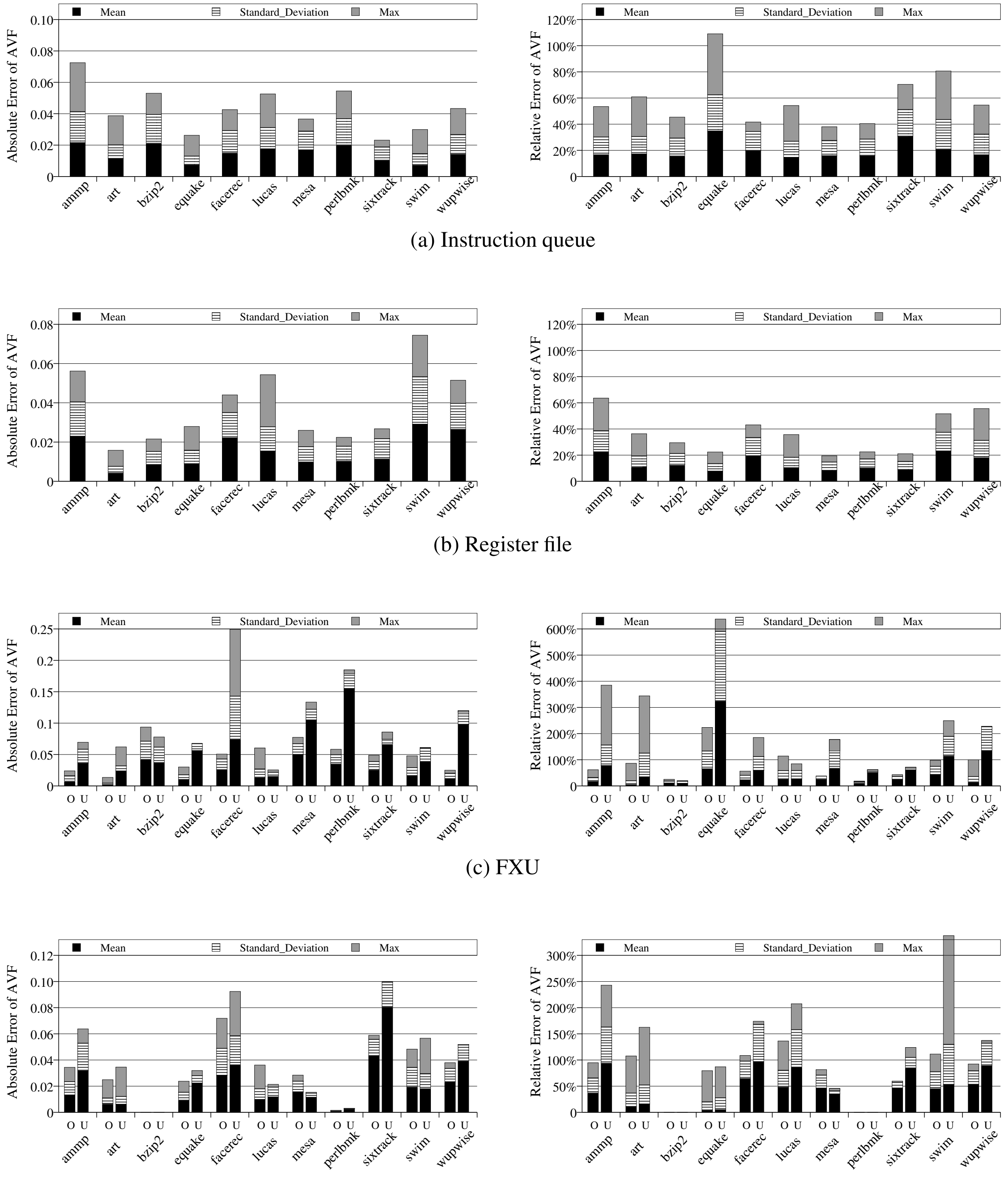
We make the following observations from the figures.

Absolute errors.

Comparing absolute errors (left charts in Figure 3), we find that our method shows low mean absolute errors – for all but 3 cases, the mean is less than 0.04 across all four structures and eleven applications. Even the Max absolute error for our method is less than 0.08 for all the structures and applications. The standard deviation for the absolute error is less than 0.05 for all cases.

In contrast, the utilization-based method has significantly larger mean absolute error in several cases. For example, for the FXU, the mean absolute error is over 0.16 for *perlbmk* and almost 0.1 for *mesa* and *wupwise*. The maximum errors are even higher.

In all cases, our estimation method shows better or almost the same absolute error as the utilization-based method. The main reason that our method shows lower error is that it is able to account for more sources of masking (e.g., masking due to dead values and instructions) than the utilization-based method. In four cases, the utilization-based method shows slightly lower mean absolute error because our method does make some statistical errors. Specifically, we use only a finite number of samples. Further, we assume that the samples are independent and, for the case of structures with multiple entries, an entry in a structure is



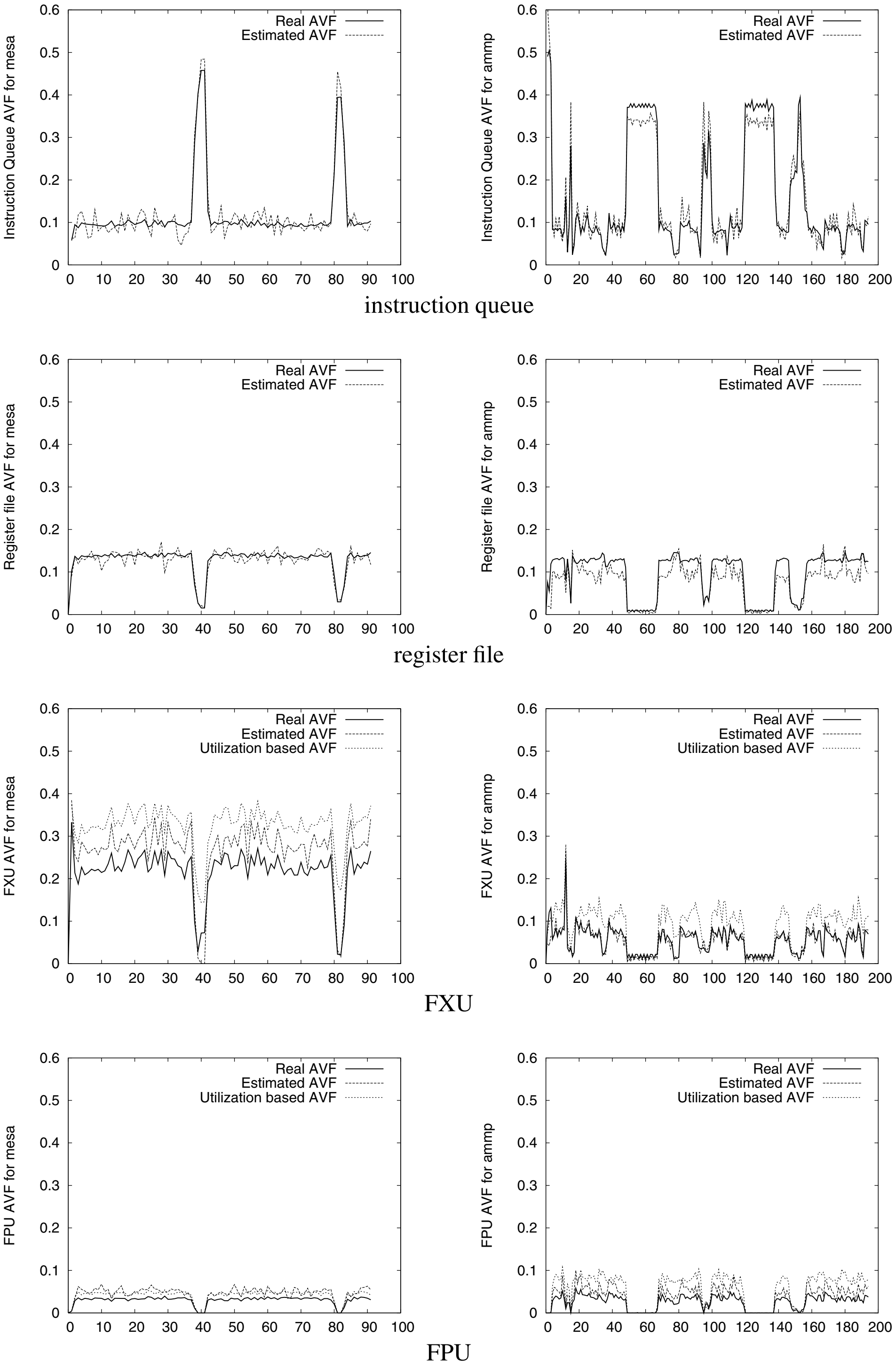
(a) Instruction queue

(b) Register file

(c) FXU

(d) FPU

Figure 3. Error in AVF estimation when compared to the SoftArch reference for (a) instruction queue, (b) register file, (c) FXU, and (d) FPU. The left charts show *absolute error* **- mean, standard deviation and maximum - across all estimation intervals of the application. The right charts show** *relative error***. The errors are shown for AVF estimates using our online method (denoted O) and the simple utilization-based method (denote U, for parts (c) and (d) only).**



instruction queue

register file

FXU

FPU

Figure 4. AVFs of instruction queue, register file, FXU, and FPU, as reported by SoftArch, our online method, and the utilization-based method (for FXU and FPU only), for applications *mesa* (left side) and *ammp* (right side). AVFs are reported for 1M cycle intervals.

not randomly selected for fault injection.

Relative errors.

Comparing relative errors (right charts in Figure 3), we find that in most cases, the mean relative error for our method is less than 20%, but in some cases, it can be as high as 65% (for FPU running facerec). The utilization-based method has a much higher mean relative error in most cases, up to over 300% for FXU running equake and 130% for FXU running wupwise.

We examine the cases where our method has a relative error larger than 20%. We find that in all these cases, the real AVF is less than 0.2. This small absolute value implies that even a small absolute error is inflated as a large relative error. At these small AVF values, the modestly large relative errors of our method are unlikely to affect design choices, given that the absolute errors are so small.

Detailed results.

The detailed plots in Figures 4 reveal several interesting observations that are not seen in the aggregate statistics. First, the absolute value of the AVF stays within 0.2 for most of the cases examined here, but it often also goes as high as 0.5. Our method is able to track this entire range of AVFs.

Second, many of the applications show significant changes in the AVF through the course of the execution. Our method is able to track all such changes very closely. The utilization-based method also tracks the changes – periods of high utilization correlate well with periods of high real AVF; however, often a significant gap remains between the absolute values of the utilization-based method and the real AVF.

Overall, these results show that our method is not only accurate on average, but also robust across a variety of scenarios. Further, for structures where a simple utilization-based method can be constructed, our results show that such a method has significantly lower fidelity than our method.

Prediction errors.

We have studied the accuracy of our scheme when used to estimate AVF. The AVF estimation is obtained at the end of each interval. However, for the AVF value to be useful for any dynamic control or adaptation scheme, we need to predict the AVF value for the next interval. Detailed AVF prediction schemes are beyond the scope of this work. In this paper, we simply illustrate that with our AVF estimation method and a simple predictor, we can quite effectively predict the AVF value for the next interval.

Such a simple predictor would work as follows. At the end of each estimation interval, it predicts the AVF of the next interval to be equal to the AVF of the past interval which is estimated using our scheme. The underlying assumption behind this simple prediction is that the AVF behavior across consecutive estimation intervals for the same application is stable or changes very slowly.

In order to evaluate the quality of our AVF prediction, for each estimation interval, we calculate the absolute error in the prediction as the absolute value of the difference between the predicted AVF and the real AVF. Figure 5 reports this absolute prediction error and the real AVF, averaged across all intervals for each application.

The results show that the absolute prediction error is quite small in all cases (less than 0.05 with two exceptions). The relative prediction error (as a percentage of the real AVF) is less than 30% of the real AVF with a few exceptions when the absolute value of the AVF is small. The prediction errors arise from two sources. The first is the predictability of the AVF itself across different intervals of the application. If the application AVF is unrelated across different intervals and changes abruptly and frequently, any predictor will fail to produce reasonable predictions. This is regardless of the accuracy of the online AVF estimation method for the current interval. The predictability of the AVF across different estimation intervals is a topic beyond the scope of this paper. Based on our observation, however, the AVF of most applications is stable across consecutive intervals, although there are a few exceptions where AVF behavior changes frequently and is harder to predict. The second source of error in the prediction is the error in our online AVF estimation method for the current interval. If the AVF estimation for the current interval has large errors, then even if the AVF is stable across all intervals, the prediction for the next interval will contain large errors. Overall, the results show that our estimation scheme combined with a simple predictor gives reasonable AVF predictions.

6 Conclusion

In this paper, we have proposed and studied a novel technique to estimate architectural vulnerability factors for soft errors in real-time. We have described the AVF estimation algorithm and the simple hardware modifications to the processor for effectively estimating the AVF. Our method is general and applies to both logic and storage structures in a microprocessor. We tested our method with a widely used simulator from industry, for four processor structures running SPEC benchmarks. The results show that our method provides acceptably accurate run-time AVF estimates under a wide variety of scenarios, compared to a detailed (and complex) offline AVF estimation tool.

References

[1] E. W. Czeck and D. Siewiorek. Effects of Transient Gate-level Faults on Program Behavior. In *Proc. of the International Symposium on Fault-Tolerant Computing*, 1990.

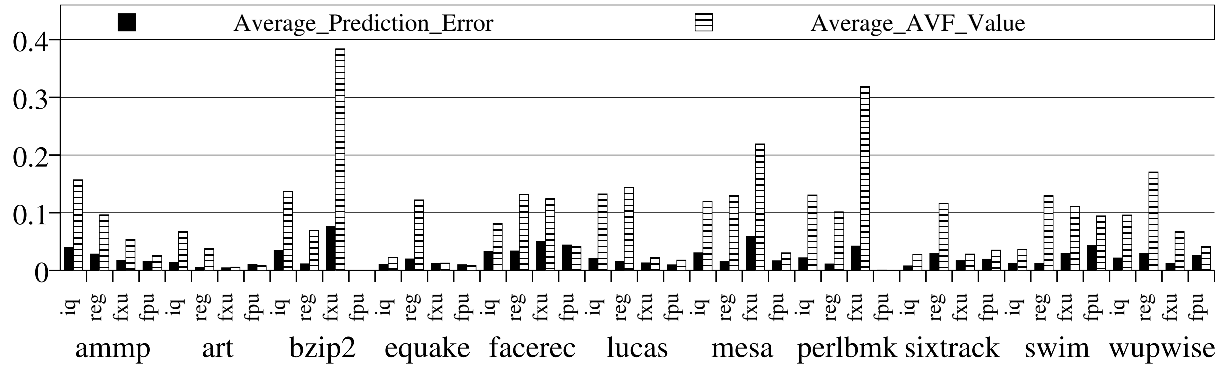


Figure 5. The relative error of the predicted AVF using a simple predictor. The predictor assumes the AVF of the next interval is equal to that of the previous interval.

[2] V. Iyengar, L. H. Trevillyan, and P. Bose. Representative Traces for Processor Models with Infinite Cache. In *Proc. of the 2nd Intl. Symp. on High-Perf. Comp. Architecture*, 1996.

[3] T. Karnik et al. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, June 2004.

[4] 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*, 2002.

[5] X. Li, S. Adve, P. Bose, and J. A. Rivers. Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2007.

[6] X. Li et al. SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[7] C. Moore. The POWER4 System Microarchitecture. In *Microprocessor Forum*, 2000.

[8] M. Moudgill et al. Environment for PowerPC Microarchitectural Exploration. In *IEEE Micro*, 1999.

[9] M. Moudgill et al. Validation of Turandot, a Fast Processor Model for Microarchitecture Evaluation. In *International Performance, Computing and Communication Conference*, 1999.

[10] S. Mukherjee et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[11] H. T. Nguyen and Y. Yagil. A Systematic Approach to SER Estimation and Solutions. In *Proceedings of the 41st IEEE International Reliability Physics Symposium*, 2003.

[12] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support For Programming Languages and Operating Systems*, 1992.

[13] S. Ross. *A First Course in Probability* (Chapter 7). Prentice Hall, 2001.

[14] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5), 2000.

[15] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[16] N. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In *Proceedings of the International Symposium on Computer Architecture*, June 2007.

[17] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the International Symposium on Computer Architecture*, June 2007.

[18] N. Wang et al. Characterizing the Effects of Transient Faults on a Modern High-Performance Processor Pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[19] N. Wang et al. Examining ACE Analysis Reliability Estimates Using Fault Injection. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[20] C. Weaver et al. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proceedings of the International Symposium on Computer Architecture*, 2004.