SOFT ERROR MODELING AND ANALYSIS FOR MICROPROCESSORS

BY

XIAODONG LI

B.Eng., University of Science and Technology of Beijing, 1997
M.Eng., Institute of Automation, Chinese Academy of Sciences, 2000
M.S., Purdue University, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Abstract

Soft errors are a growing concern for processor reliability. Recent work has motivated architecture level studies of soft errors since the architecture level can mask many raw errors and architectural solutions can exploit workload knowledge. My dissertation focuses on the modeling and analysis of soft error issues at the architecture level.

We start with the widely used method for estimating the architecture level mean time to failure (MTTF) due to soft errors. The method first calculates the failure rate for an architecture level component as the product of its raw error rate and an architecture vulnerability factor (AVF). Next, the method calculates the system failure rate as the sum of the failure rates (SOFR) of all components, and the system MTTF as the reciprocal of this failure rate. Both steps make significant assumptions. We analyze the validity of the two steps using both mathematical analysis and experiments. We find that although the AVF+SOFR method is valid for most current systems under current raw error rates, for some cases it can lead to significant discrepancies. We explore scenarios in which such discrepancies could occur in practice.

To find an alternative model that is not subject to such limitations, we propose a model and tool called SoftArch that does not make the above AVF+SOFR assumptions. SoftArch is based on a probabilistic model of error generation and propagation process in a processor. Our experiments show that SoftArch does not exhibit the discrepancies the AVF+SOFR suffered. We apply SoftArch to an out-of-order processor running SPEC2000 benchmarks. Our results motivate selective and dynamic architecture level soft error protection schemes. Next, as another application, we quantify the impact of technology scaling on the processor soft error rate, taking the architecture level masking and workload characteristics into consideration.

By using the SoftArch tool, we observe that there is much architecture level masking and that the degree of such masking can vary significantly across workloads, individual units, and workload phases. Thus, it is natural to consider the architecture level solutions to take advantage of such variations. In order to do that, one would need reasonably accurate estimate of the amount of masking effect in real time. For most current systems, AVF is an accurate abstraction of the architecture level masking effect. Existing solutions for estimating AVF are often based on offline simulators and usually hard to implement in real processors. In this dissertation, we propose a novel way of estimating AVF online, using simple modifications to the processor. Our method applies to both logic and storage structures on the processor and does not require complex offline calibration for different workloads. We test our method with a widely used simulator from industry for SPEC benchmarks. The results show that the method provides reasonably accurate run-time AVF estimates.

To sum up, this dissertation studies the architecture level soft error modeling and analysis problems. It provides new techniques to examine and take advantage of architecture level soft error behavior. We apply our tool to investigate the impact of technology scaling on soft errors. We also propose an efficient online AVF estimation algorithm.

Acknowledgements

This dissertation would not have been possible without the support of many people.

First, I would like to express my deep gratitude to my thesis advisor, Prof. Sarita Adve, for her constant guidance and invaluable support through my graduate study. Her insights and attention to detail were instrumental in helping me achieve my academic goals. I also want to thank Dr. Pradip Bose and Dr. Jude Rivers from IBM research, whom I have been fortunate enough to work with during my Ph.D. research. They provided vital industrial perspectives for all my research work.

I am also indebted to the members of my dissertation committee, Prof. Joseph Torrellas, Prof. Craig Zilles, Prof. Yuanyuan Zhou and Dr. Jude Rivers for their critical feedback, especially at my preliminary exam and thesis defense.

I would like to thank the members of the RSIM group, especially Alex Li, and Pradeep Ramachandran, for their important feedback during my practice talks for the prelim and thesis defense.

Finally, I would like to extend my thankfulness to my family for their support throughout all these years. I thank my wife, Jing Liu, for her patience and love that helped me survive the hardest time.

Table of Contents

Chapter 1

Introduction

1.1 Motivation and goal

CMOS technology scaling has brought tremendous improvement in performance for semiconductor devices. As we move to sub-100nm lithographies, however, soft errors are emerging as a new challenge in processor design. Soft errors or single event upsets are transient errors caused by high energy particle strikes such as neutrons from cosmic rays [1, 2] and alpha particles from packaging material. Such strikes can flip the bit stored in a storage cell and change the value being computed by a logic element. Although a consensus on exact soft error rates is still lacking, there is a growing concern about the phenomenon.

Recent work has shown that many of the raw errors that occur at the device and circuit level may be masked at the architecture level, potentially motivating lower cost protection mechanisms. For example, Wang et al. report that about 85% of the raw errors are masked at the architecture level [3]. By considering solutions at the architecture level, knowledge of workload behavior can be exploited, leading to potentially more efficient protection solutions (e.g., [3, 4]). These observations motivate the need for comprehensive models and tools to study soft errors at the architecture level and solutions at the architecture level to remedy the problem.

In this dissertation, we focus on the architecture level modeling and analysis of the soft error problem.

1.2 Contributions

This dissertation makes three key contributions.

1. Analyzes the assumptions and limitations of the AVF+SOFR method [5].

2. Proposes SoftArch, a new architecture level model and tool for modeling and analyzing soft errors at the architecture level [6].

3. Proposes a novel method for online estimation of AVF for soft errors [7].

We now discuss each of the contributions in more detail.

1.2.1 Limitations of AVF+SOFR

The first contribution of this dissertation is a detailed analysis of the limitations of the widely used method for estimating MTTF due to soft error – the AVF+SOFR method. First, our work builds a fundamental understanding of the AVF+SOFR method at the architecture level. It explicitly identifies some fundamental assumptions in the AVF+SOFR approach and shows that these assumptions depend on three parameters. We then use both mathematical and experimental techniques to check the validity of the above method across a large design space. We find that the above method is valid for most of the realistic cases under current raw error rates. However, for some combinations of large systems, long running workloads with large phases, and/or high raw error rates, the AVF+SOFR method can lead to significant discrepancies.

1.2.2 SoftArch

In search for an alternative model, we propose SoftArch, to enable analysis of soft errors at the architecture level in modern processors. SoftArch is based on a probabilistic model of error generation and propagation process in a processor. Compared to prior architecture level tools, SoftArch is more comprehensive or faster. What is more important, our experiments show that SoftArch does not need to make the same assumption that AVF+SOFR method does. The MTTF computed by SoftArch has less than 2% error relative to the MTTF value calculated using the Monte-Carlo method for the whole wide design space we have studied.

We also use SoftArch to quantify the MTTF of a modern out-of-order processor and the contribution of different structures to the failure rate, for various SPEC benchmarks. Our results are consistent with previous studies. We show that not only is there significant architecture level masking effects, there is substantial inter- and intra-application variation in MTTF or failure rate and substantial application-dependent variation that contributes to the failure rate from different structures. These results motivate selective protection of only the most vulnerable structures and dynamic, application-aware protection schemes.

As another application, we apply SoftArch to quantify the impact of technology scaling on the processor soft error rate, taking the architecture level masking effects and workload characteristics into consideration. For our evaluation, we use SoftArch to study the AVF and soft error rate (SER) for different structures in a modern superscalar processor running SPEC2000 benchmarks. We compare the SERs across four different technologies ranging from 180nm to 65nm with the same microarchitecture. We find that with scaling, the AVF for logic structures often decrease, the AVF for storage elements remains roughly unchanged, and the MTTF for the full processor roughly follows the trend for the raw SER of storage structures (i.e., the MTTF decreases from 180nm to 90nm and increases from 90nm to 65nm.) This study assumes the number of transistors on the chip stays the same.

1.2.3 Online AVF estimation

Using SoftArch, we find that there is much architecture level masking and that the degree of such masking can vary significantly across workloads and also individual workload phases. This provides opportunities for an architecture level solution to take advantage of the application behavior variation. For that to happen, it is important to be able to accurately estimate the masking effect which is captured by the architecture vulnerability factor (AVF) for most current systems. Existing solutions for estimating AVF are often based on off-line simulators and are usually hard to implement in real processors. This dissertation proposes a novel way of estimating AVF on-line while the program is running. We propose some simple hardware modifications for the processor and use an algorithm to effectively estimate AVF. It is a general method that applies to both logic and storage units on the processor. Compared to previous methods for estimating AVF, our method does not require offline simulation with simulators, nor does it require calibration for different workloads. We test our method with SoftArch coupled to a widely used simulator from industry and SPEC benchmarks. The results show that our method provides accurate run-time AVF estimates.

1.3 Organization

The rest of the dissertation are organized as follows. Chapter 2 analyzes the current state-of-the-art AVF+SOFR soft error modeling method. Chapter 3 proposes SoftArch, an architecture level model and tool to analyze soft errors at the architecture level. Chapter 4 proposes a new efficient on-line AVF estimation method. Chapter 5 discusses related work and Chapter 6 presents the conclusions of this dissertation and possible avenues of future work.

Chapter 2

Assumptions and limitations of the AVF+SOFR method

The AVF+SOFR method has been widely used in estimating processor MTTF values. The methodology computes MTTF using two simple steps [8], illustrated in Figure 2.1. The **AVF step** calculates the failure rate of each individual processor *component* (e.g., ALU, register file, issue queue) as the product of its raw failure rate and a factor that accounts for architecture level masking effect. Mukherjee et al. formalize the notion of the architecture level masking effect as the architectural vulnerability factor (AVF) [8] and show how to calculate it for various architectural components [9, 8]. The **SOFR step** calculates the failure rate of the entire processor (or any system) as the Sum Of the Failure Rates (SOFR) of the individual components of the processor or system (as calculated in the AVF step). It calculates the MTTF of the processor (or system) as the reciprocal of its failure rate.

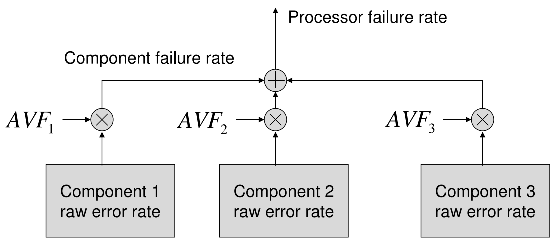

Processor failure rate
Component failure rate
AVF_1
AVF_2
AVF_3
Component 1 raw error rate
Component 2 raw error rate
Component 3 raw error rate

Figure 2.1 The AVF and SOFR steps for MTTF.

Both the AVF and SOFR steps implicitly make certain assumptions about the statistical properties of the underlying error process. While these assumptions, described below, may hold for the raw error process, it is unclear whether they hold for the architecturally masked process. Our goal is to examine the validity of these assumptions underlying the mathematical basis of the AVF and SOFR steps, and the implications of these assumptions for evaluating soft error MTTF for real systems.

Next, we will analyze the assumptions through mathematical analysis and experiments. Our rigorous mathematical methods analyze the limits and value ranges of various parameters within which the AVF+SOFR assumptions hold true. In order to validate the conclusions and quantify the limits, we design simulation-based experiments to explore a wide design space.

We find that the impact of the above assumptions on the MTTF calculation depends on three parameters related to the environment, system, and the workload respectively: (1) the raw error rate of the individual components, (2) the number of components in the system on which SOFR is applied, and (3) the length of the full execution or the longest repeated phase of the workload. Specifically, our evaluations show the following.

1. For systems where the individual components have small raw error rates, the total number of components is small, and where the workload consists of repeated executions of a short program, the AVF+SOFR assumptions introduce negligible error. To our knowledge, previously published work using the AVF+SOFR methodology considers systems and workloads that obey the above constraints. This result is by itself significant since it, for the first time, validates the mathematical basis for using the AVF+SOFR methodology.

2. Our results show that the AVF+SOFR method can result in large discrepancies in MTTF (up to 100%) for individual components that have large raw error rates (e.g., as would be the case in space or in accelerated tests or with components consisting of many millions of bits) and/or systems that have many components (e.g., large clusters of thousands of processors) and/or long-running workloads with different utilization characteristics over large time windows (e.g., server workloads that run at high utilization in the day but low utilization in the night). This problematic part of the design space is certainly much smaller and less common than the space over which AVF+SOFR is valid; however, it is not negligible and represents several realistic systems. Our results give a note of caution against blind use of the AVF+SOFR method for such systems.

We discuss the AVF+SOFR method and their assumptions in detail in Section 2.1. Section 2.2

and Section 2.3 provide an analytical and an experimental view.

2.1 AVF+SOFR method and assumptions

2.1.1 The AVF step

In a given cycle, only a fraction of the bits in a processor storage component and only some of the logic components will affect the final program output. A raw error event that does not affect these critical bits or logic components has no adverse effect on the program outcome. Mukherjee et al. used the term architecture vulnerability factor (AVF) to express the probability that a visible error (failure) will occur, given a raw error event in a component [8]. The AVF for a hardware component can be calculated as the percentage of time the component 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.

Mukherjee et al. calculate the FIT rate of a processor component as the product of the component's AVF and its raw FIT rate (i.e., the FIT rate of the component if every bit were ACE). Denoting the raw FIT rate of the component as λ_c (also called the raw soft error rate or raw SER) and its AVF as AVF_c, they derive the MTTF of the component as:

$$MTTF_c = \frac{1}{\lambda_c \cdot AVF_c} \tag{2.1}$$

We show in Section 2.2.1 that an assumption underlying the above equation is that the time to failure for a program is uniformly distributed over the program. We explore the cases where this assumption is and is not true to assess the validity of the AVF step.

2.1.2 The SOFR step

Sum of failure rates (SOFR) is an industry standard model for combining failure rates of individual processor (or system) components to give the failure rate and MTTF of the entire processor (or system). Let the system contain k components with failure rate of component i as $FailureRate_i$ (which is assumed to be the reciprocal of the MTTF of component i or $1/MTTF_i$). The SOFR model calculates the failure rate ($FailureRate_{sys}$) and the MTTF ($MTTF_{sys}$) of the system as:

$$FailureRate_{sys} = \sum_{i=1}^{k} FailureRate_i = \sum_{i=1}^{k} \frac{1}{MTTF_i} \tag{2.2}$$

$$MTTF_{sys} = \frac{1}{FailureRate_{sys}} \tag{2.3}$$

The SOFR model makes two major assumptions [10]. First, it assumes that each component has a constant failure rate (i.e., exponentially distributed time to failure) and the failures for different components are independent of each other. Section 2.2.2 shows that architectural masking may violate this assumption in some cases. Second, the SOFR model assumes a series failure system; i.e., the first instance of a component failure causes the entire processor to fail. This assumption holds if there is no redundancy in the system. Since our focus is on the impact of program-dependent architectural masking on the statistical properties of the failure process, we continue to make this assumption as well and focus only on the first assumption.

2.1.3 AVF+SOFR assumptions

A key assumption behind the AVF step is that the probability of failure due to a soft error in a given component is uniform across a program's execution. This allows a single AVF value to be used to derate the raw error rate of a component. The uniformity assumption is reasonable for raw error events since the probability of a high energy particle strike is no different at different points in the program's execution for most realistic scenarios. However, it is unclear that the assumption holds after incorporating architectural masking. Similarly, a well-documented assumption for the SOFR step is that the time to failure for a given component follows an exponential distribution. Again, the assumption is reasonable and widely accepted for raw error events, but it is unclear that it holds for failures after architectural masking.

Thus, both the AVF and SOFR steps make assumptions about the error process that may be considered questionable, once architectural masking effects are taken into account. The question we address is: Under what conditions (if any) does the violation of the above AVF+SOFR assumptions introduce significant errors in the calculation of the MTTF?

2.2 AVF+SOFR limitations: an analytical view

This section uses mathematical analysis to understand the limits of the basic assumptions underlying the AVF+SOFR methodology for estimating MTTF for soft errors. Later sections back these results with detailed Monte-Carlo simulations for actual workloads.

Our analysis makes two assumptions that are also made by the AVF+SOFR methodology.

(1) Inter-arrival times for raw errors in a component are independent and exponentially distributed with density function $\lambda e^{-\lambda t}$. It is reasonable to assume that the time to the next high energy particle strike is independent of the previous strike and is exponentially distributed (the process is memoryless). In practice, there is some device- and circuit-level masking, which could possibly render the raw error process that is subject to architectural masking as non-exponential. In our experiments, however, we do not have this low-level masking information available; we therefore assume the best case for the AVF+SOFR methodology – that the inter-arrival time for raw errors before any architectural masking is an exponential process with density function $\lambda e^{-\lambda t}$. We refer to λ as the *raw error rate.*

(2) The workload runs in an infinite loop with similar iterations of length L. This work considers the effect of real application workloads. For a workload that runs for a finite time, there is a possibility that no failure occurs during its execution. For a meaningful interpretation of MTTF for a system running such a workload, we assume that the workload runs repeatedly in a loop until the first failure. All iterations of this loop are identical and each represents a single invocation of the original workload. We refer to the size of this loop iteration as L. Workloads that are naturally infinite also run in a loop. We assume that such a workload also consists of identical iterations, each of size L. This assumption is trivially satisfied since L can potentially be infinite. (All the prior work on AVF+SOFR has been in the context of finite workloads.)

We additionally assume that program failure occurs if a raw error is not masked. Although the time to failure and the time to the next raw error event are continuous random variables, for convenience, we often consider time in units of processor cycles below (for architectural masking, for a given cycle, all raw error events during any part of the cycle are either masked or not masked).

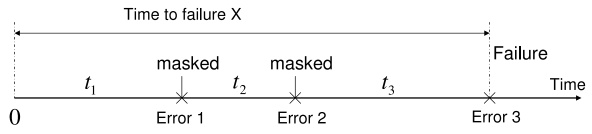


Figure 2.2 Sequence of raw error events. t_i is the time between two raw error events and is exponentially distributed. X is a random variable representing the time to the first raw error event that is not masked and leads to program failure. The figure shows a case where $X = t_1 + t_2 + t_3$.

2.2.1 The AVF step: MTTF for an isolated functional or storage unit

The AVF step computes the MTTF of a single component of the processor using equation 2.1. We examine the validity of this step by deriving the MTTF of a given component from first principles.

Figure 2.2 illustrates a sequence of raw error events with inter-arrival times of $t_1, t_2, .., t_n, ...$ Each of these times is an instance of a random variable, say T, with exponential density function $\lambda e^{-\lambda t}$. Each raw error has some probability of being masked. Failure occurs at the first raw error that is not masked.

Let X be the random variable that denotes the time to failure. Then $X = t_1 + t_2 + .. + t_k$ if the first $k-1$ raw errors are masked and the kth raw error is not masked. Thus, $X = \sum_{i=1}^{K} t_i$, where K is a random variable such that $K = k$ denotes the event that the first $k-1$ raw errors are masked and the kth raw error is not masked.

Now the MTTF of the component is simply the expected value of X, $E(X)$. Using a standard result for the expectation of a sum of random variables [11], it follows that: $MTTF = E(X) = E(K)E(T)$. We know that $E(T) = \frac{1}{\lambda}$ (this would be the MTTF if there were no architectural masking and every raw error resulted in failure). Thus,

$$MTTF = E(K)\frac{1}{\lambda} \tag{2.4}$$

Comparing with equation 2.1, to validate the AVF step, we would need to show that $E(K) = \frac{1}{AVF}$ for all cases. However, $E(K)$ depends on the workload characteristics and the raw error rate λ, and, in general, cannot be analytically derived. Nevertheless, with certain assumptions, we show that we can derive $E(K)$ to be 1/AVF, validating the AVF step for cases where the assumptions hold. We then show counter-examples where these assumptions do not hold, and the MTTF derived from first principles is significantly different from the MTTF derived from the AVF equation 2.1.

AVF is valid when $L \cdot \lambda \rightarrow 0$

We first show that if the product of the raw error rate and the program loop size is very small, then $E(K) = \frac{1}{AVF}$ (and so the AVF equation holds). Below we show that in this case, any of the L cycles in the program loop are equally vulnerable to a raw error event occurrence. From this, it will follow that the expected value of K (i.e., the count of the first raw error event that is not masked) is the same as 1/AVF.

Let T be the cycle count at which the next raw error event occurs. Then, without loss of generality, $T\ mod\ L$ is the cycle count for this event relative to the start of the loop iteration. Appendix A of an extended version technical report [12] shows that if $L \cdot \lambda \rightarrow 0$, the random variable $T\ mod\ L$ follows a uniform distribution over $[0, L]$. In other words, for very small $L \cdot \lambda$, any of the L cycles of program execution are equally vulnerable to a raw error event occurrence.

Thus, the probability that the next raw error event occurs at cycle i (relative to the start of the loop iteration) is $1/L$. Let p_i be the probability that a raw error event that occurs at cycle i (relative to the start of the loop iteration) is masked (p_i is 0 or 1 for a given program execution). Therefore, the probability that the next raw error event is masked is $\sum_{i=1}^{L} \frac{1}{L} \cdot p_i$. This value is a constant that we denote by M.

Now to calculate $E(K)$, we first calculate P{K=k}. This is the probability that the first $k-1$ raw error events are masked and the kth raw error event is not masked. Since raw error events are independent, it follows that P{K=k} $= M^{k-1}(1 - M)$. That is, K is a geometrically distributed random variable and so $E(K) = 1/(1 - M)$. Thus, we just need to show that $1 - M$ is the same as the AVF.

$(1 - M)$ can be expressed as $\sum_{i=1}^{L} \frac{1-p_i}{L}$. $1 - p_i$ is the probability that a raw error event at cycle i will not be masked and will cause failure. $1 - M$ is therefore the average of this probability over the entire program length. This is exactly the definition of AVF. Thus, we have shown that the AVF equation 2.1 is valid when $L \cdot \lambda \rightarrow 0$.

AVF is not valid for some values of λ and L

In this section, we construct a simple (synthetic) program that serves as a counter-example to show that the assumptions behind the AVF step do not always hold.

Consider a program with an infinite loop with iteration size L, such that the considered system component is active for the first A cycles and is idle for the remaining $A+1$ to L cycles of the iteration. As before, let X be the random variable denoting the time to failure for the component running the above program. Let T be the random variable denoting the time to the first raw error event. If T is in cycles $[0, A], [L, L+A], ...$, then the component is active and the time to failure is simply the value of T. Otherwise, the raw error occurs in an idle period, say, of iteration k, and it is masked. Further, any raw errors until the next active period (i.e., until cycle kL) will also be masked.

As seen at cycle kL, the distribution for the time to the next raw error event (starting from kL) is the same as that starting from time 0. This is due to the memoryless property of the exponential distribution.[1] Further, as seen from kL, the masking process is also the same as at time 0, since all iterations are identical. Thus, given that there is no failure until cycle kL, the expected time to failure from cycle kL is again E(X).

It follows that given that the first raw error event occurs in the idle period of the *kth* iteration, the expected time to failure is $kL + E(X)$. Now using a standard result for conditional expectation [11], we get the following:

$$\begin{aligned} E(X) &= E(E(X|T)) = \textstyle\int_0^\infty E_{X|T}(t) \cdot f_T(t)dt \\ &= \textstyle\int_0^A \lambda e^{-\lambda t} t dt + \int_A^L \lambda e^{-\lambda t}(L + E(X))dt + \\ &\quad \textstyle\int_L^{L+A} \lambda e^{-\lambda t} t dt + \int_{L+A}^{2L} \lambda e^{-\lambda t}(2L + E(X))dt ... \end{aligned}$$

The above equation has the following closed form solution (Appendix A of [12]), giving the MTTF of the component from first principles:

$$\begin{aligned} E(X) &= \tfrac{1-e^{-\lambda L}}{1-e^{-\lambda A}} \cdot \Big(\tfrac{Le^{-\lambda L}}{(1-e^{-\lambda L})^2} - \tfrac{Le^{-\lambda A}e^{-\lambda L}}{(1-e^{-\lambda L})^2} - \tfrac{Ae^{-\lambda A}}{(1-e^{-\lambda L})} + \\ &\quad \tfrac{1}{\lambda}\tfrac{(1-e^{-\lambda A})}{(1-e^{-\lambda L})} + L\tfrac{e^{-\lambda A}-e^{-\lambda L}}{(1-e^{-\lambda L})^2}\Big) \end{aligned}$$

[1]Recall that for an exponential distribution, $P(T < t + \triangle t | T > t) = \frac{(e^{-\lambda t} - e^{-\lambda(t+\triangle t)})}{e^{-\lambda t}} = 1 - e^{-\lambda \triangle t}$. That is, given that a raw error has not occurred at time t, the probability that the error will occur within some time $\triangle t$ after t is the same as that of it occurring within $\triangle t$ after time 0.

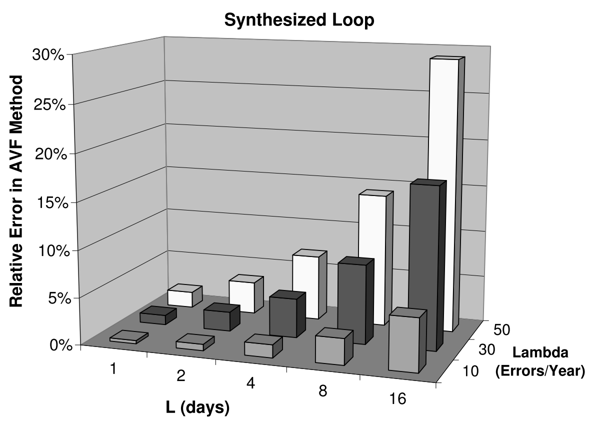


Figure 2.3 The relative error in the AVF step applied to a large 100MB cache running a loop with iteration size of L days with each iteration busy for $L/2$ days and idle for the rest. Lambda is the raw error rate of the entire cache (the smallest value represents 0.001 FIT per bit).

The AVF for our program is $\frac{A}{L}$; therefore, the MTTF according to the AVF method is:

$$E_{AVF}(X) = \frac{L}{A} \cdot \frac{1}{\lambda}$$

Now we can calculate the relative difference between the MTTF from first principles and from the AVF method as:

$$\frac{|E_{AVF}(X) - E(X)|}{E(X)}$$

When λL is very small, we can show that the two MTTFs converge to the same value. For other cases, there can be a significant difference. Figure 2.3 shows the difference between the two MTTF values for a 100MB cache for different values of L and λ. We vary L from 1 to 16 days, setting A as $L/2$ in each case. We start with λ at 10^{-8} errors/year per bit (0.001 FIT/bit) [6] which translates to 10 errors/year for the full cache. We additionally show results for λ of 3 and 5 times this value to represent changes in technology and altitude. Although the errors are small for the baseline (smallest) value of λ, they can be significant for higher values. Later sections perform a more systematic experimental exploration of the full parameter space.

2.2.2 The SOFR step: MTTF for multiple functional and/or storage units

The SOFR step derives the MTTF of a system using the MTTFs of its individual components, as shown in equations 2.2 and 2.3. As discussed in Section 2.1.2, it assumes that for each component, the time to failure follows an exponential distribution with a constant failure rate (in conjunction with the AVF step, this rate is the product of the component's raw error rate and AVF). We next explore the validity of this assumption, given that each component sees significant architectural masking.

Again, the validity of the assumption depends on the values of the component's raw error rate λ and the program loop size L. Sections 2.2.2 and 2.2.2 respectively discuss cases for which the assumption is and is not valid.

SOFR is valid when $L \cdot \lambda \to 0$

We show that if $L \cdot \lambda \to 0$ for a component, then the time to failure, X, for that component is exponentially distributed with rate parameter $\lambda \cdot AVF$.

Section 2.2.1 showed that in this case, $X = \sum_{i=1}^{K} t_i$, where K follows a geometric distribution with mean $1/AVF$ and the t_i's are exponentially distributed with rate λ. We can calculate the density function of X as follows:

$$\begin{aligned} f_X(x) &= \lim_{\triangle x \to 0} \frac{P(x<X<x+\triangle x)}{\triangle x} \\ &= \lim_{\triangle x \to 0} \sum_{i=1}^{\infty} \frac{P(x<X<x+\triangle x|K=i)P(K=i)}{\triangle x} \end{aligned}$$

where $P(x < X < x + \triangle x|K = k) = P(x < \sum_{j=1}^{k} t_j < x + \triangle x)$.

$\sum_{j=1}^{k} t_j$ is the sum of several independent exponentially distributed random variables with rate λ. Such a sum follows the Erlang-n distribution which has the probability density function of $\frac{\lambda(\lambda x)^{n-1}}{(n-1)!}e^{-\lambda x}$ [10]. Thus,

$$\begin{aligned} f_X(x) &= \sum_{i=1}^{\infty} ((1 - AVF)^{i-1}(AVF)\frac{\lambda(\lambda x)^{i-1}}{(i-1)!}e^{-\lambda x} \\ &= (AVF)\lambda e^{-\lambda x} \sum_{i=1}^{\infty} \frac{((1-AVF)\lambda x)^{i-1}}{(i-1)!} \\ &= \lambda(AVF)e^{-\lambda(AVF)x} \end{aligned}$$

This is an exponential distribution with rate $\lambda \cdot AVF$. This validates the assumption for the SOFR step for the case when $\lambda \cdot L$ is small.

The general case for λ and L values

In general, it is difficult to analytically characterize the time to failure distribution function for real (or even synthetic) programs after architectural masking. In this section, to demonstrate a mathematical basis, we choose a distribution that is "close" to exponential (and mathematically tractable) and determine the validity of using SOFR on that distribution.

We choose the following probability density function for the time to failure (after architectural masking) for a component.

$$f_X(x) = \begin{cases} \frac{2}{\sqrt{\pi}} e^{-x^2} & x \in [0, \infty] \\ 0 & \text{elsewhere} \end{cases}$$

The cumulative distribution function (CDF) of X is $F_X(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$, $x \in [0, \infty]$.

It follows that the MTTF of the component is $E(X) = \frac{2}{\sqrt{\pi}} \int_0^\infty x e^{-x^2} dx = \frac{1}{\sqrt{\pi}}$.

Assume a system with N such identical components where X_i denotes the time to failure for component i. Since we assume series failure, it follows that the time to failure of the system, Y, is $\min(X_1, X_2, ..., X_N)$.

The CDF of Y is $F_Y(y) = 1 - (1 - F_X(y))^N$.

The PDF is $f_Y(y) = \frac{dF_Y(y)}{dy} = N * (1 - F_X(y))^{N-1} * f_X(y)$

The MTTF of the system is $E(Y) = \int_0^\infty f_Y(y) y dy$

The above integration cannot be calculated analytically. We solve it numerically using a software package to derive the real MTTF for N from 2 to 32.

The SOFR step calculates the MTTF of the system using Equations 2.2 and 2.3. For the component MTTFs used in the equations, we use the real MTTF derived above ($\frac{1}{\sqrt{\pi}}$):

$$MTTF_{sofr} = \frac{1}{\sum_{i=1}^{N} \sqrt{\pi}} = \frac{1}{N\sqrt{\pi}}$$

Figure 2.4 shows the error in $MTTF_{sofr}$ relative to the MTTF derived from first principles. We see that the error grows from 15% for a system with two components to about 32% for a system with 32 components.

2.2.3 Summary of implications

Our mathematical analysis so far provides intuition for when the AVF+SOFR method works. The AVF step averages the "utilization" of a component over the whole program. It therefore makes

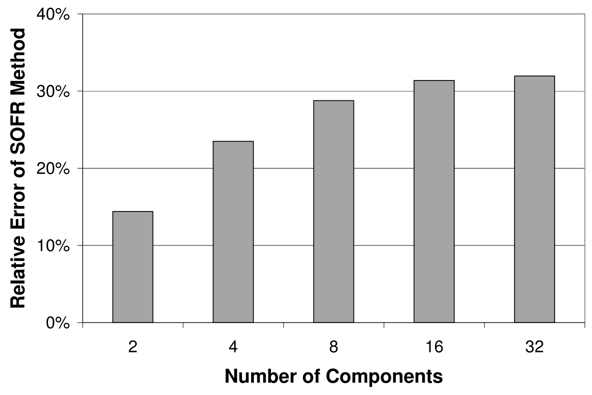


Figure 2.4 The relative error introduced by the SOFR step for a synthesized example.

the implicit assumption that every point of the program will have uniform probability of being hit by a soft error. The SOFR step assumes that the time to failure for each individual component follows the exponential distribution. Our analysis shows that the above assumptions are valid when $\lambda \cdot L \to 0$. However, in the general case, these assumptions may not hold. We show mathematically tractable synthetic examples to illustrate a few such cases. The next sections provide a more systematic experimental exploration of the parameter space to assess the extent of the errors due to these assumptions.

2.3 AVF+SOFR limitations: an experimental view

In this section, we show that significant discrepancies can arise in many realistic scenarios using experiments with SPEC benchmarks.

2.3.1 Experimental methodology

This section describes the methodology for our experimental analysis of the assumptions of the AVF and SOFR steps. For each step, we first evaluate the assumptions for single processor systems common today running SPEC CPU2000 applications, and using detailed simulation to determine architectural masking. We then take a broader view, and evaluate the assumptions for a large design space, including large clusters of processors and a broader range of (synthesized) workloads, but with less detailed simulation of architectural masking.

For both cases, we first generate a *masking trace* that indicates, for each system component, whether a raw error in a given cycle would be masked for the evaluated system and workload.

Base Processor Parameters	
Processor frequency	2.0 GHz
Fetch/finish rate	8 per cycle
Retirement rate	1 dispatch-group (=5, max) per cycle
Functional units	2 integer, 2 FP, 2 load-store, 1 branch
Integer FU latencies	1/4/35 add/multiply/divide
FP FU latencies	5 default, 28 divide (pipelined)
Reorder buffer size	150 entries
Register file size	256 entries (80 integer, 72 FP, and various control)
Memory queue size	32 entries
iTLB	128 entries
dTLB	128 entries
Base 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
Base Contentionless Memory Latencies	
L1 Latency	1 cycles
L2 Latency	10 cycles
Main memory Latency	77 cycles

Table 2.1 Base POWER4-like processor configuration.

To calculate the real MTTF of the system (without the AVF+SOFR assumptions), we use the Monte Carlo technique to model the raw error process, apply the masking trace to the process, and determine the MTTF of the modeled system.

Today's uniprocessors running SPEC

To determine the impact of architectural masking in a modern processor, we study an out-of-order 8-way superscalar processor (Table 2.1) running programs from the SPEC CPU2000 suite (9 integer and 12 floating point benchmarks). To generate the masking trace, we use Turandot [13], a detailed trace-driven microarchitecture level timing simulator. We simulate an instruction trace of 100 million instructions for each SPEC benchmark running on the above processor configuration.

We choose four processor components to study the impact of architecture masking: the integer, floating point, and instruction decode units, and the 256 entry register file, with raw error rates of $2.3 * 10^{-6}$, $4.5 * 10^{-6}$, $3.3 * 10^{-6}$, and $1.0 * 10^{-4}$ errors/year respectively (10^{-8} errors/year = 0.001 FIT). Li et al. [6] derived these error rates using published device error rates for current technology [14] and estimates of the number of devices of different types in different components [6].

For the integer, floating point, and instruction decode units, we assume that a raw error is masked in a cycle if the unit is not processing an instruction in that cycle (i.e., the unit is not busy). If the unit is busy processing an instruction, then for simplicity, we conservatively assume that the error is not masked and will lead to failure. For the register file, we assume that the raw error strikes happen on each register with equal probability and error in a given register is masked if the register contains a value that will never be read in the future. If the register's value will be read, we conservatively assume the error is not masked and will lead to failure. Our assumptions of when an error is not masked are conservative since it is possible that an error in an active unit or in a register value that will be read may not affect the eventual result of the program. We did not perform a more sophisticated analysis to more precisely determine when an error is masked because such an analysis is orthogonal to the point of this dissertation and beyond the scope of this work.

Our detailed Turandot simulation produces a masking trace for each simulated SPEC application. The trace contains information on whether a raw error in a given cycle in one of the four considered processor components will or will not be masked.

Broader design space exploration

We also explore a broad design space for the AVF and SOFR steps. We consider a variety of systems consisting of various numbers of components, operating in various environments, with different raw error rates, and running different workloads. We use the term *system* to include a single processor (either a full processor or only a part of it) or a large cluster of thousands of processors. A *component* of a system is the smallest granularity at which the analysis for architectural masking is applied. Specifically, the AVF is calculated at the granularity of a component; the SOFR step then aggregates the information from the different components to give the MTTF for the entire system. In our SOFR experiments, we use component MTTFs obtained from the Monte Carlo method; therefore, the error reported is only that caused by the SOFR step.

Based on our analysis in Section 2.2, the key parameters affecting the AVF and SOFR steps are the raw error rate of the different components of the processor (or system), the number of components in the system (only for SOFR), and the program loop size or workload. The following discusses the space we explore for each of these important parameters. Table 2.2 summarizes this space.

Component raw error rate. The component raw error rate depends on the number of devices

Dimension	Value				
N	10^5	10^6	10^7	10^8	10^9
S	1	5	100	2000	5000
C	2	8	5000	50000	500000
Workload	SPEC fp	SPEC int	day	week	combined

Table 2.2 The design space explored. N = number of elements (e.g., bits) in a component; S = scaling factor for the baseline raw error rate of an element (depends on technology and altitude); and C = number of components in the system (e.g., processors in a cluster).

or elements (bits of on-chip storage or logic elements such as gates) in the component and the raw error rate per element. We denote the number of elements in a component as N. N can be as large as 10^9 for large cache structures or if we consider the entire processor as one component in a large cluster of multiple processors. To keep the design space exploration tractable, without loss of generality, we assume that all N elements have the same raw error rate.

We also explore different values for the raw error rate per element. Under current technology, the terrestrial raw error rate per bit for on-chip storage is about 10^{-8} errors/year (0.001 FIT), which we refer to as the baseline raw error rate. To account for changes in the raw error rate due to technology scaling and at high altitudes, we introduce a parameter S that we use to scale the above baseline rate. We use scaling factors of 1, 5, 100, 2,000, and 5,000 in our analysis. The larger factors correspond to systems running in airplanes flying at a high altitude and for systems in outer space because of strong radiation at those heights [2]. Test systems using accelerated conditions are also subject to high raw error rates.

The raw error rate for a given component is determined as the product of N, S, and the above baseline raw error rate (Table 2.2).

Number of components: We denote the number of components in the system as C. We study a wide range of values for C, ranging from 2 to $500,000$. The larger numbers represent large cluster systems with C components (each of which may be a full processor or a microarchitectural component within a processor, depending on the granularity at which AVF is collected).

Workload and generation of the masking traces: We evaluate all systems in the broad design space with the SPEC CPU2000 benchmarks mentioned in Section 2.3.1. However, these are short programs (small loop iteration size L). Many real world workloads show large differences in behavior over long time scales (large L) that are difficult to capture with the SPEC benchmarks.

In an attempt to simulate some of the behaviors of real world applications, we construct three synthetic applications. The first (called *day*) is a continuous loop where the loop iteration size is set to 24 hours. The loop is busy during the day (half the time) and idle at night. The second (called *week*) is a loop with iteration size one week. It is busy during the five business days of the week and idle for the weekend. The third (called *combined*) concatenates two SPEC benchmarks in a loop with iteration size of 24 hours. The first half of the iteration runs one benchmark and the second half runs the other benchmark.

For a system with multiple processors, we assume all processors run the same workload. Additionally, for the synthesized workloads, we assume that a component is a full processor; e.g., C=2 implies a 2 processor system. We assume that each processor masks raw errors only during the idle portion of the workload (e.g., night time for the day workload). For the SPEC workloads, we again assume that each component is a full processor (running the same benchmark). For the masking trace, we use the SPEC masking traces for three units in each processor (integer, floating point, and instruction decode) – we apply these three traces to the corresponding units simultaneously to determine whether there is a processor-level failure.

Monte carlo simulation

To calculate the real MTTF, we perform Monte Carlo simulation where we do the following for each trial. For each component in the modeled system, we generate a value from an exponential distribution with rate specified by the modeled system (Table 2.2). This value gives the arrival time of the next raw error event for the component. We use the masking trace of the workload to determine whether a raw error at that time would be masked. If it is masked, we generate a new raw error event from an independent exponential distribution for that component and repeat. If it is not masked, we consider the component failed. The component that is earliest to fail gives the time to failure of the system for this trial. We run a total of 1,000,000 trials and report the average of the time to failure as the MTTF of the modeled system/workload configuration.

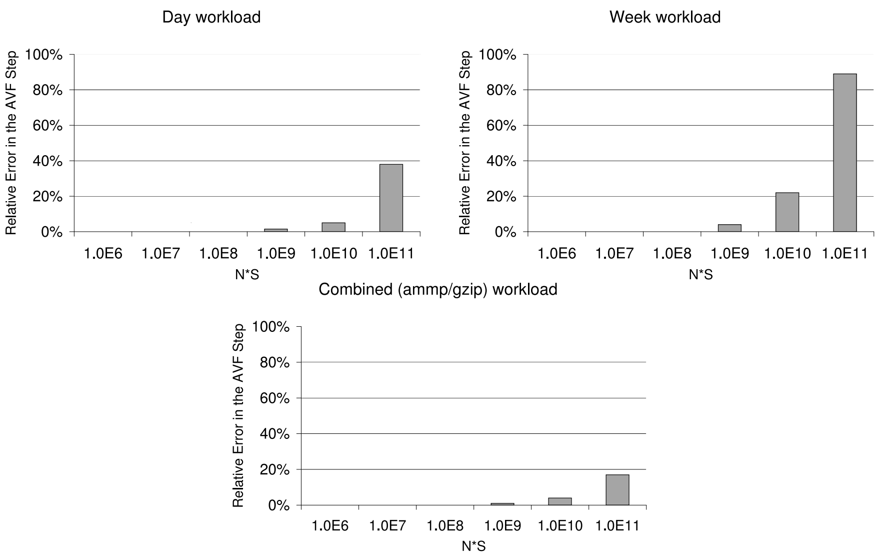


Figure 2.5 Error in MTTF from the AVF step relative to the Monte Carlo method for the synthesized workloads for representative values of $N \times S$ (# bits in the component $\times$ scaling factor for baseline raw error rate).

2.3.2 Results

AVF and SOFR with Today's Uniprocessors Running SPEC

We first evaluated the discrepancy between the Monte Carlo MTTF and the MTTF using the AVF and SOFR steps for today's uniprocessors running SPEC (as described in Section 2.3.1). We found that the MTTF from the AVF step matched the Monte Carlo MTTF very well for each of the four processor components and each benchmark ($< 0.5\%$ discrepancy for all cases). Similarly, the processor MTTF calculated using the SOFR step also matched the Monte Carlo MTTF very well.

Thus, for single processor systems with a small number of small components running SPEC benchmarks, the AVF+SOFR method works very well. We note that in prior work, the method has been applied primarily in this context. These results are consistent with our mathematical analysis. The loop size L for the SPEC benchmarks and the component raw error rates used here are small; therefore, from Sections 2.2.1 and 2.2.2, we expect that the AVF and SOFR assumptions would be valid.

AVF: A Broad Design Space View

For the design space described in Table 2.2, we computed the component MTTF using the Monte Carlo and AVF methods as described in Section 2.3.1. Note that since the AVF step is applicable to only a single component, $C = 1$ for all experiments in this section. Further, for a given workload, since only the product of N and S matters, we report relative error in the AVF step as a function of different values of $N \times S$.

We found that for each SPEC benchmark, the AVF step works well for all N and S values studied (relative error $< 0.5\%$). However, for the longer running synthesized workloads, we observe significant discrepancy when $N \times S$ is large (i.e., component raw error rate is large). Figure 2.5 shows the error in the AVF MTTF relative to the Monte Carlo MTTF for representative values of $N \times S$ for the three synthesized workloads. In all three cases, for $N \times S \geq 10^9$, the AVF step sees significant errors (up to 90%). This high value of $N \times S$ may occur when the AVF step is applied to either large components (e.g., a 125MB cache with $N = 10^9$ bits), or when the component size is moderate but the raw error rate per element (bit) is high (e.g., $S = 1000$ because of high radiation at high altitudes).

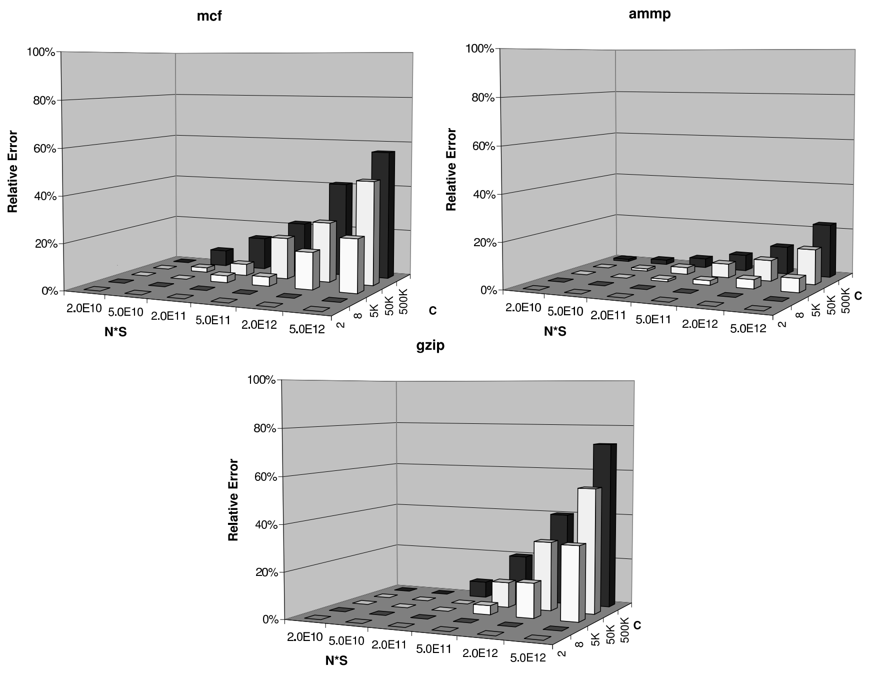
Our experiments show both positive and negative errors, depending on the workload. Thus, AVF may either over- or under-estimate MTTF in practice.

Again, the above observations match well with our theoretical analysis in Section 2.2.1. Thus, for SPEC like benchmarks that run for a short time, it is safe to use the AVF step to calculate the MTTF of a component. However, the AVF step must be applied carefully when using a workload with large variations over large time scales coupled with either a large component or a large per-element raw error rate for the component.

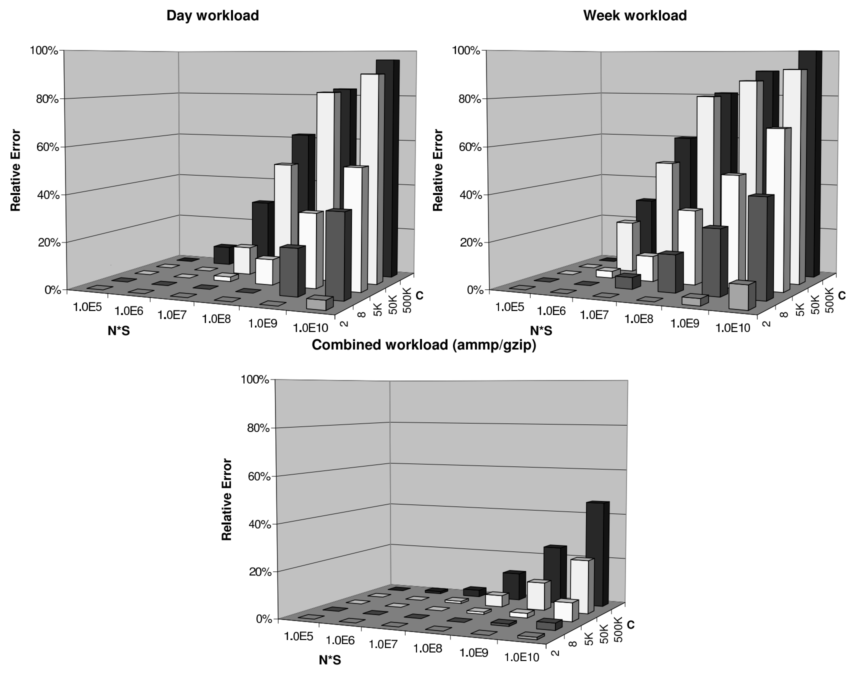
SOFR: A Broad Design Space View

Figures 2.6(a) and (b) report the error in the SOFR step relative to the Monte Carlo method for three representative SPEC benchmarks and the three synthesized benchmarks respectively. For each case, the error is reported for representative values of C and $N \times S$ covered by the design space in Table 2.2.

Focusing on the SPEC workloads (Figure 2.6(a)), we see that the SOFR step is accurate for



(a) SPEC benchmarks



(b) Synthesized benchmarks

Figure 2.6 Error in MTTF from the SOFR step relative to the Monte Carlo method for representative values of C (# components) and $N \times S$ (bits per component $\times$ scaling factor for baseline raw error rate) for (a) SPEC and (b) synthesized benchmarks.

systems with a small number of components ($C = 2$ or 8) for all studied values of $N \times S$. When system size grows to 5,000 components or larger, we see significant errors, but only with very large values of $N \times S$. For example, for a cluster of 5,000 processors with each processor containing $N = 10^9$ bits of on-chip storage, the baseline raw error rate would need to scale 2,000 times or more to see a significant error. In practice, terrestrial systems will likely fall into the part of the design space where the SOFR step does not introduce any significant error for SPEC applications.

Focusing on the synthesized workloads (Figure 2.6(b)), for the day workload, we see a significant error using the SOFR step when $N \times S \geq 10^8$ and $C \geq 5{,}000$. The error increases as these parameters increase. For example, with 12.5MB of storage for each processor ($N = 10^8$) and baseline raw error rate ($S = 1$), a 5,000 processor cluster sees an error in MTTF of 11%. For a similar cluster of 50,000 processors, the error jumps to 50%. While large, such a cluster is not unrealistic. For the week workload, since the loop size is larger than the day workload, the MTTF errors are correspondingly larger. Thus, the 5,000 and 50,000 processor systems mentioned above respectively see MTTF errors of 32% and 80% for this workload. With larger processors (more storage bits) or larger systems, the error can grow to 90% or more. Thus, for these workloads, the SOFR step incurs significant errors for realistic systems.

Finally, the combined workload (with two SPEC applications) shows a relative error smaller than for the day or week workload, but there is still a significant error for some cases.

In summary, for SPEC benchmarks under current technology and on the ground, the SOFR step gives accurate MTTF estimates. However, in general, for larger scale workloads, care must be taken to examine the workload behavior, number of system components (e.g., processors), and the raw error rate for the components (governed by component size and per-bit or per-element error rate) before applying SOFR.

2.4 Summary

We have examined key assumptions behind the AVF+SOFR method for estimating the architecture level processor MTTF due to soft errors. We use rigorous theoretical analysis backed by simulation-based experiments to systematically explore the applicability of the AVF and SOFR steps across a wide design space. Our analysis and experiments show that while both steps are valid under the terrestrial raw soft error rate values of today's technology for standard workloads (e.g., SPEC), there are cases in the design space where the assumptions of the AVF and SOFR steps do not hold. In particular, for long running workloads with large component-level utilization variations over large time scales, the assumptions are violated for systems with a large number of components and/or with high component-level raw error rate (i.e., large component size and/or large per-bit or per-element raw error rate). Under these conditions, the projected MTTF of the modeled system or chip could show large errors. In general, our work builds a better understanding of the conditions under which the standard AVF+SOFR method may be used to project MTTF accurately, and alerts users to the risks of using the model blindly in conditions where the foundational axioms of the model break down.

Chapter 3

SoftArch model

As we have shown in Section 2.3.2, the AVF+SOFR method is based on significant assumptions. The assumptions become questionable for some systems in the design space. In this section, we propose a model called SoftArch which can provide fast and accurate analysis of soft errors at the architecture level. We will show in Section 3.4.1 that SoftArch does not need to make the same assumptions as the AVF+SOFR method. In Section 3.4.2, We apply SoftArch to evaluate the MTTF of a processor. In Section 3.4.3, we examine the effect of technology scaling on the architecture level soft error rate taking the architecture level masking effect into consideration.

3.1 Introduction

SoftArch works with a high-level architecture timing simulator to track the raw probability of error in the value of each bit (instruction or data) communicated or computed by any pipeline stage in the processor. A value may be erroneous either because (i) it is physically struck by a particle during its residence time in a structure, or (ii) it is the result of a communication of an erroneous value, or (iii) it is computed using one or more erroneous input values. We refer to the first case as error *generation* and to the second and third cases as error *propagation.* To model the error generation probability, we use a combination of residence time and raw SER numbers for storage structures, and a simple abstraction for logic. For error propagation probability, we apply simple probability theory on the error probabilities of the sources of the propagation.

During program execution, SoftArch identifies the values that could affect program outcome. For each such value, it uses the tracked errors for the value and the simulator timing data to determine the probability of failure and time to failure due to that value. This enables determining the mean time to failure using basic probability theory. SoftArch also keeps enough information on the microarchitectural structures occupied by each value to determine the contribution of different structures to the overall MTTF.

SoftArch is based on the first principle of the MTTF calculation, thus it does not need to make the AVF+SOFR assumptions. We perform the same set of experiments for the same designed space as described in Section 2.3. We find that for every point in the design space, the error in the MTTF computed by SoftArch is less than 1% for a single component and 2% for the full system. Thus, SoftArch does not exhibit the discrepancies shown by AVF+SOFR.

Next, we use SoftArch to quantify the MTTF of a modern out-of-order processor and the contribution of different structures to the failure rate, for various SPEC benchmarks. Our results (consistent with, but more comprehensive than, previous studies) are as follows: (1) there is significant architecture level masking of soft errors, (2) there is substantial inter- and intra-application variation in MTTF or failure rate, and (3) there is substantial application-dependent variation in the contribution to the failure rate from different structures. These results motivate selective protection of only the most vulnerable structures and dynamic, application-aware protection schemes.

Finally, as another application, we apply SoftArch to quantify the impact of technology scaling on the architecture level processor soft error rate, taking the architecture level masking effects and workload characteristics into consideration. We scale the same design with the same number of transistors over four technology generations. We find that with scaling, the derating factors for logic structures often decrease, the derating factors for storage elements remain roughly unchanged, and the FIT for the full processor roughly follows the trend for the raw SER of storage structures.

3.2 SoftArch details: a model for architecture level MTTF

The SoftArch model consists of the following components, covered in Sections 3.2.1– 3.2.4 respectively. (1) A probabilistic model for *soft error generation* in values residing in storage structures or passing through logic. (2) A model for *soft error propagation*, which results in the propagation of generated errors to other values. (3) A definition of when an erroneous value contributes to *processor failure*. (4) A model for calculating *mean time to failure (MTTF)* for a processor for a given workload.

3.2.1 Error generation model

Error generation in storage elements

Current processors include several storage structures such as caches, register files, queues, TLBs, and latches. A soft error in a storage structure occurs when a high energy particle strikes a device in the structure, and the resulting charge collected exceeds the critical charge (Q_{crit}) required to flip the stored bit value. We call this a *raw* soft error.

We seek to determine the probability that a value v_i residing in a (possibly multiple bit) storage location for time T incurs a raw soft error during T. We assume that if an error occurs, the value is corrupted; i.e., we ignore the low probability that multiple errors could correct the value. It is widely accepted that raw soft errors for storage follow a constant failure rate or exponential time-to-failure distribution model. Let λ denote the raw failure rate, also referred to as the raw soft error rate or SER, for the storage location considered. Then the probability that the value v_i will incur a raw soft error in time T, denoted e_i, is $1 - e^{-\lambda \cdot T}$. In practice, both λ and T are small enough that we can approximate $e^{-\lambda \cdot T}$ as $1 - \lambda \cdot T$. This gives $e_i = \lambda \cdot T$.

Thus, the probability that an error is generated for a value v_i in a storage location depends on the raw SER for that location, λ, and the residence time of the value in the location, T. λ is determined by circuit layout, technology, and environmental parameters (e.g., the amount of charge stored, charge collection efficiency, and particle flux). There has been extensive work on determining the value of λ using circuit level simulation or measurement (Section 3.3.2). Residence time T depends on the program and the processor architecture, and can be determined through architecture level timing simulation (Section 3.3.1).

Error generation in logic elements

Combinational logic elements are used for computation and control within a pipeline stage. A high energy particle strike on a device in a logic circuit may create a current pulse that may affect the value produced by the circuit. This transient effect becomes visible only if it is captured by the subsequent latch. Instead, the transient effect could be masked due to electrical masking (the current pulse attenuates as it goes through the gates in the circuit), logical masking (the current pulse affects parts of the circuit that do not affect the output value), or latch window masking (the corrupted result is not latched because it does not arrive within the required timing window for the latch). Logic SER has been ignored in most prior architectural studies because the above masking makes the effective SER much smaller than that of storage structures. However, as technology scales, these masking effects are diminishing and the logic SER is projected to increase significantly [14].

For our architecture level model, it is desirable to include the above circuit-level masking effects within the *raw logic SER* value. Because these masking effects depend on the circuit layout and inputs, the desired raw logic SER will differ for different logic circuits and even for different inputs. In general, it is hard to abstract all of these effects. We therefore use a simple abstraction consisting of one parameter called e_{logic} corresponding to each type of logic circuit (e.g., e_{alu} for the ALU or e_{fpu} for the FPU). e_{logic} is defined to be the probability that, given correct inputs, the result produced by the corresponding circuit at the end of the computation is incorrect because of soft errors. e_{logic} can be estimated using circuit level SER analysis tools, based on the layout of that logic circuit and technology parameters. In our implementation, we use a simple estimation based on prior work [14] and the gate and latch counts for the logic circuit (Section 3.3.3).

3.2.2 Error propagation model

In a processor, values are read from storage locations, possibly processed, and the original or newly computed values are stored elsewhere. (We consider the values stored in the new locations to be new values, even if they are identical to the original ones.) During this process, errors in the original values will propagate to the new values. For example, if the value, $v1$, in register $r1$ is corrupted and later used to generate a result $r3 = r1 + r2$, the error in $v1$ will propagate to the new value stored in $r3$.

Conceptually, we would like to track how errors are propagated to new values and determine the probability that a new value is erroneous. These probabilities will then allow us to determine the probability of failure and the mean time to failure (depending respectively on which erroneous values cause failure and when). The probability of error in a newly generated value (say v_3) depends on the probability of error in the input values (say v_1 and v_2) used to generate v_3. In general, denoting V_i to mean the event that value v_i has an error, denoting $\mathrm{P}(V_i)$ as the probability

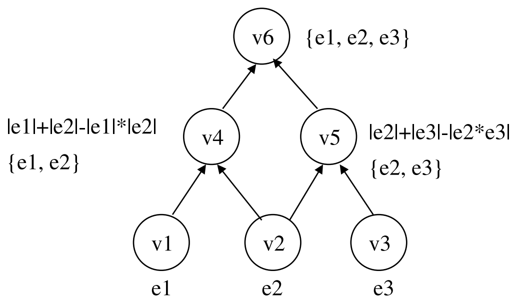


Figure 3.1 An example for error propagation.

of V_i, and assuming that any error in either v_1 or v_2 will cause an error in v_3, the probability of error in v_3 can be given by $\text{P}(V_3) = \text{P}(V_1 \bigcup V_2) = \text{P}(V_1) + \text{P}(V_2) - \text{P}(V_1 \cdot V_2)$, where $V_1 \cdot V_2$ is the event that v_1 and v_2 both have errors.

If the errors in v_1 and v_2 are independent, then $\text{P}(V_1 \cdot V_2)$ is simply $\text{P}(V_1)\text{P}(V_2)$. On the other hand, if the errors are perfectly correlated (e.g., if v_2 was just generated by copying v_1 to another location), then $\text{P}(V_1 \cdot V_2) = \text{P}(V_1) = \text{P}(V_2)$. In general, however, the errors in two values could be partially correlated and estimating $\text{P}(V_1 \cdot V_2)$ is more difficult. Accounting for the correlation and determining the resultant probability requires keeping track of the raw errors that were originally responsible for the errors in v_1 and v_2.

For example, Figure 3.1 shows a dataflow graph where values $v1$, $v2$, and $v3$ incur errors $e1$, $e2$, and $e3$ with probability $|e1|$, $|e2|$, and $|e3|$ respectively. Assuming $e1$, $e2$, and $e3$ are independent of each other, the probability of error for value $v4$ is $|e_1| + |e_2| - |e_1| \cdot |e_2|$ and that for $v5$ is $|e_2| + |e_3| - |e_2| \cdot |e_3|$. The errors in $v4$ and $v5$ are correlated since they share the same error from $v2$ – if $v2$ has an error, both $v4$ and $v5$ will have errors. Therefore, to calculate the probability of error in $v6$, the correlation between the errors in $v4$ and $v5$ needs to be taken into account. We do this by tracking the original independent raw error events that cause errors in different values.

For our model, we do not need to calculate the probability of error for a value immediately upon its generation – we only need probability calculations for values that eventually cause failure as defined in the next section. Therefore, for purposes of determining how errors propagate among values, we simply keep track of the set of all the raw error events that can cause an error in a value, and propagate this entire set when a value is used to generate a new value. For example, in Figure 3.1, the error set for $v4$ is $\{e1,\ e2\}$ and for $v5$ is $\{e2,\ e3\}$. Thus, the error set for $v6$ should be $\{e1,\ e2,\ e3\}$. We can now easily calculate the error probability for $v6$, since $e1$, $e2$, and $e3$ are independent.

More generally, consider a value v_i residing in a storage location. Let t_j be the time interval between two successive reads of v_i (or between the first write and read of v_i). We refer to the event that v_i incurs a raw soft error over time t_j as a *basic storage error event*. If v_i was generated through computation logic, then we refer to the event that v_i incurred a logic error (after considering circuit level masking effects) during this computation as a *basic logic error event*. We refer to a basic storage or basic logic error event as a *basic error event* or simply a *basic error*. All basic errors are independent of each other, with probabilities given by the error generation models in Section 3.2.1.

The error propagation model requires determining the basic errors that need to be propagated to a new value. For each value v_i, we associate a *basic error set*, denoted E_i. This is the set of basic errors directly incurred by or propagated to v_i.[1] Thus, for a new value v_i created at time t_i, the propagation model seeks to determine v_i's E_i at t_i.

First, we handle the simple case where v_i is generated by reading an old value v_0 from a storage location and writing it to another storage location. In this case, the error set E_i is simply the error set for v_0 at time t_i.[2]

Next, we handle the case where v_i is created through some computation $op(in_1, in_2, ...in_k)$, where k ≥ 1, in_j's are input operands, and op is any operation. The creation of v_i involves a possible basic logic error event, say b_i, with probability e_{op}. Then E_i is simply $E_{in_1} \cup E_{in_2} \cup \cdots \cup E_{in_k} \cup \{b_i\}$.

Thus, we can generate the basic error set for a newly created value. Since all the error events in this set are independent, the probability of error in the new value can be calculated as a function of the probabilities of the errors in its basic error set (which are known from Section 3.2.1). For example, in Figure 3.1, the probability of error for $v6$ is $|e1| + |e2| + |e3| - |e1| \cdot |e2| - |e2| \cdot |e3| - |e1| \cdot |e3| + |e1| \cdot |e2| \cdot |e3|$.

[1]Note that for v_i in a storage location, each time it is read, a new basic error event is added to E_i (to indicate an error occurrence in the interval since it was last read).

[2]We assume the process of moving a value from one location to another across wires does not induce any errors. Currently, wires do not appear to have soft error problems. However, in the future, soft errors from wires could be easily incorporated by adding another basic error due to the wires to the set E_i.

3.2.3 Program failure and time to failure

Not all erroneous values cause program failure. For example, an error that occurs in a dead value does not cause failure since the value is not used again. Similarly, an error in a speculative instruction that is later squashed does not cause program failure. We say an erroneous value results in program failure if the error is observable by an external observer. Broadly, this includes (1) values that are written to an output device, (2) values that affect program control flow (e.g., the value of a branch target), (3) the value of an instruction opcode (an error could make the opcode illegal, causing a program crash), (4) any value representing an address of a memory location (an error could cause access to prohibited locations, causing a crash), (5) and a destination register field of an instruction (an error could result in the corruption of an unknown and undesirable register).

Depending on the system modeled and the implementation, the precise set of values where errors may cause program failure will vary (e.g., in a processor with speculation, an errror in the opcode of a misspeculated instruction will not cause program failure). Further, a specific implementation of the model may choose to conservatively assume that errors in a superset of the above values will cause failure. Section 3.3.5 describes the set of values where errors are considered to cause failure in our implementation.

We call the above defined set of values where errors would lead to program failures as the *failure set*, denoted by $V_F = \{v_{f1}, v_{f2}, ...\}$. Additionally, our model also requires determining the time, t_{fi}, at which a failure due to v_{fi} occurs. This is determined through the architectural timing (performance) simulator. We assume that the failure set $\{v_{f1}, v_{f2}, ...\}$ is ordered such that $t_{fi} < t_{fj}$ for $i < j$.

3.2.4 Determining mean time to failure (MTTF)

We next derive mean time to failure (MTTF) for a processor running a given workload. Our model so far provides: (1) the values that can cause failure: $\{v_{f1}, v_{f2}, ...\}$, (2) the corresponding times for these failures: $\{t_{f1}, t_{f2}, ...\}$, (3) for each value, v_{fi}, the set of independent basic errors $E_{fi} = \{e_{fi-1}, e_{fi-2}, ...\}$ that can produce an error in v_{fi}, and (4) the probability for each independent basic error.

Infinite programs. First, consider a workload that runs forever. Its MTTF is the sum of the

t_{fi}'s, each weighted by the probability that v_{fi} is erroneous and no previous value in the failure set is erroneous. Denoting the number of elements in the failure set as N (N could be ∞), we have:

$MTTF = \sum_{i=1}^{N} t_{fi} \cdot$ *(Probability that v_{fi} has an error and none of $v_{f1}, ..., v_{fi-1}$ have an error)*

Given the basic error sets E_{fi} and the probabilities of the constituent errors, we use basic probability theory to determine the probability of the events in the above summation. For example, let $E_{f1} = \{e_1, e_2\}$ and $E_{f2} = \{e_2, e_3\}$. Then the probability that v_{f2} has an error and v_{f1} does not have an error is the probability that at least one of the errors in (E_{f2} - E_{f1}) occurs and none of the errors in E_{f1} occurs. This is $|e_3| \cdot (1 - |e_1|) \cdot (1 - |e_2|)$, denoting probability of e_i by $|e_i|$.

Finite programs. Most of our workloads, however, are finite programs that run for a relatively short amount of time. To determine MTTF in a meaningful way for a processor running such a program, we assume that the program runs repeatedly in a loop forever. If a failure always occurs in the first run of the program, then the MTTF for the finite program, denoted **MTTF$'$**, can also be represented by the above equation for infinite programs. If there is no failure in the first run, then we need to expand the equation to include possible failures in subsequent runs.

Let T_{exec} be the execution time of one run of the program. Then the time to failure due to v_{fi} in the kth run of the program is $(k-1)T_{exec} + t_{fi}$. This time to failure must be weighted by the probability that none of the prior $k-1$ (independent) runs fail, v_{fi} is erroneous in the kth run, and none of the values prior to v_{fi} in the failure set are erroneous in the kth run. That is,

$MTTF = \sum_{k=1}^{\infty} \sum_{i=1}^{N} \{(k-1)T_{exec} + t_{fi}\} \cdot$ *(Probability that none of the prior k-1 runs fail)* $\cdot$ *(Probability that v_{fi} has an error and none of $v_{f1}, ..., v_{fi-1}$ have an error)*

To simplify the above equation, we define $FailureProb'$ as the probability that a given run of the program will see a failure. That is,

$FailureProb' = \sum_{i=1}^{N}$ *(Probability that v_{fi} has an error and none of $v_{f1}, ..., v_{fi-1}$ have an error)*

Thus, in the MTTF equation, the term *Probability that none of the prior k-1 runs fail* can be represented as $(1 - FailureProb')^{k-1}$. The MTTF equation then becomes:

$MTTF = \sum_{k=1}^{\infty} \sum_{i=1}^{N} \{(k-1)T_{exec} + t_{fi}\} \cdot (1 - FailureProb')^{k-1} \cdot$ *(Probability that v_{fi} has an error and none of $v_{f1}, ..., v_{fi-1}$ have an error)*

Rearranging the terms slightly,

$MTTF = \sum_{k=1}^{\infty}(1 - FailureProb')^{k-1} \cdot \sum_{i=1}^{N}\{(k-1)T_{exec} + t_{fi}\} \cdot$ *(Probability that v_{fi} has an error and none of $v_{f1}, ..., v_{fi-1}$ have an error)*

Now applying the definition of $MTTF'$, we get:

$$MTTF = \sum_{k=1}^{\infty}(1 - FailureProb')^{k-1} \cdot \{(k-1)T_{exec} \cdot FailureProb' + MTTF'\}$$
$$= T_{exec} \cdot FailureProb' \sum_{k=1}^{\infty}(k-1) \cdot (1 - FailureProb')^{k-1} + MTTF' \sum_{k=1}^{\infty}(1 - FailureProb')^{k-1}$$

Using $\sum_{k=1}^{\infty} x^{k-1} = \frac{1}{1-x}$ and $\sum_{k=1}^{\infty}(k-1)x^{k-1} = \frac{x}{1-x^2}$ to simplify the equation, we get

$$MTTF = \frac{T_{exec} \cdot (1 - FailureProb')}{FailureProb'} + \frac{MTTF'}{FailureProb'}$$
$$= \frac{T_{exec} + MTTF'}{FailureProb'} - T_{exec}$$

Note that we can derive the contribution to MTTF from a specific processor structure by assuming zero probability for errors generated in other structures.

3.3 Implementation of the SoftArch model

We have implemented the SoftArch model in the SoftArch tool. There are five key components to the implementation: (1) integration with an architecture level timing (i.e., performance) simulator, (2) estimation of λ, (3) estimation of e_{logic}, (4) implementation of the basic error set corresponding to each value and the operations on these sets, and (5) identifying the values in the failure set. The following sections discuss each of these components.

3.3.1 Integration with timing simulation

The SoftArch model provides MTTF for a specific program running on a processor. It requires integration with a performance (or timing) simulator that runs the program, and provides to the SoftArch model timing information about the values read/written/computed in different parts of the processor. This work also uses the Turandot simulator as described in Section 2.3.1 with the same parameters that were chosen to roughly correspond to the POWER4 microarchitecture [15].

We track soft errors using the SoftArch model for most of the important structures in the processor, including the instruction buffer (IBUF), instruction decode unit (IDU), integer and floating point register files (REG), integer functional units (FXU), floating point units (FPU), instruction TLB (iTLB), data TLB (dTLB), and instruction queues (IQ). We assume the load/store queue, caches, and memory are protected using ECC, and do not consider a soft error rate for them. We also do not model soft errors for the branch prediction unit since these do not cause processor failures.

3.3.2 Estimation of λ

Irom et al. [16] and Swift et al. [17] report measured values of raw SER cross section for the TLB and floating point registers for PowerPC processors. The raw SER cross section is defined as the number of errors per particle influence and is related to the raw SER as follows [2]:

Raw SER for a storage structure = (SER cross section for the structure)(nucleon flux)(# bits in the structure)

From [16], the raw proton SER cross section for the TLB structure in a 200nm PowerPC processor is about $5 \cdot 10^{-14} cm^2/bit$ for proton energy larger than 20Mev. From [17], the raw proton SER cross section for the floating point register structure in a PowerPC 750 processor is about the same value. Since protons and neutrons have similar characteristics at higher energy range, we use the proton cross section to roughly estimate the raw neutron SER of different structures. We do not model the alpha particle SER since Karnik et al. [18] show that in devices where Q_{crit} is large, neutron SER dominates. This is the case for the array structures we study here. Further, the detailed estimation of raw SERs is not the focus of this dissertation.

According to Ziegler [2], neutron flux with sufficient energy (>20 Mev) at sea level is $10^5 particles/cm^2 \cdot yr$. Using the above equation, we can derive the raw SER for the register file in 200nm technology as $5.7 \cdot 10^{-4}$ FIT/bit (1 FIT is one failure every 10^9 hours). Since we model a processor in 90nm technology, we scale the raw SER rate using scaling data by Karnik et al. [18]. Karnik et al. show that neutron SER in SRAM increases about 30% from 200nm to 90nm technology. Thus, we assume that the raw SER for the register file in 90nm technology is $7.42 \cdot 10^{-4}$ FIT/bit. Assuming a 64 bit register and a 2 GHz processor, we can derive that λ for a register value is $6.60 \cdot 10^{-24}$ errors/cycle.

Although Irom et al. [16] and Swift et al. [17] do not report data for the instruction buffer, instruction queue and integer register file, we assume the SER cross section value for these to be similar to the reported results for TLB and floating point registers (we could not find any other sources of measured data for these structures either). Using an approach similar to the above, we get λ for an instruction buffer entry as $6.60 \cdot 10^{-24}$ errors/cycle and for an instruction queue and a TLB entry as $1.13 \cdot 10^{-23}$ errors/cycle.

3.3.3 Estimation of e_{logic}

At 100nm, Shivakumar et al. [14] showed the raw SER for a latch to be $3.5 \cdot 10^{-5}$ FIT and for a 16FO4 logic chain to be $5 \cdot 10^{-6}$ FIT (after circuit level electrical and latch window masking). Based on the gate and latch counts for a logic circuit, we can therefore estimate the raw SER for that circuit at 100nm (we use the same value for 90nm). (This is conservative since it ignores circuit-level logical masking which depends on the inputs and the exact logic function.)

Specifically, let *#LogicChains* and *#Latches* be the number of logic chains and latches respectively in a logic circuit (e.g., FPU, FXU, or IDU). Then for our 2 GHz processor,

$$e_{logic} = \frac{(\#LogicChains \cdot 5 \cdot 10^{-6} + \#Latches \cdot 3.5 \cdot 10^{-5})}{10^9 \cdot 3600 \cdot 2 \cdot 10^9}$$

We estimated the gate/latch count information for our simulated processor as follows.[3] We first estimated the relative areas of each modeled structure from published floorplans of the POWER4. Since the total transistor count for the processor is known, we could then assign area-based estimates of transistor counts for each modeled structure. Reasonable assumptions about transistor density differences between SRAM and logic dominated structures were also factored in. We estimate 10K latches and 70K gates for the FXU (integer ALU), 14K latches and 100K gates for the FPU, and 7K latches and 50K gates for the IDU. (Our implementation assumes all FXU operations have the same e_{logic} and all FPU operations have the same e_{logic}). It follows that e_{logic} for the IDU, FXU, and FPU is $5.16 \cdot 10^{-23}$, $7.23 \cdot 10^{-23}$, and $3.67 \cdot 10^{-23}$ respectively.

[3]Although our microarchitectural parameters were chosen to be close to the POWER4, structure-wise gate/latch count information for such commercial processors is not available. We acknowledge that our estimates of these counts may not be close to actual values.

3.3.4 Tracking basic error set E_i for value v_i

The error propagation model requires tracking basic error sets, using set copy and union operations. These sets can potentially be unbounded. To reduce space and dynamic memory management overhead, we use a fixed size FIFO table to store the basic errors in a set (one table per set, 100 entries per table in our implementation). To further reduce space, the table entry only stores a sequence number that identifies the error. A common central table stores the pertinent information for each sequence number, including probability of the corresponding error and where it is generated. In case of overflow of a basic error table (i.e., > 100 basic error sources contribute to the corresponding value), the oldest entry in the table is discarded. This loses information about an error source for the value. We conservatively assume that the value causes failure due to the dropped error with probability of that error and at the time the error is dropped. In our experiments, overflow rarely occurs.

3.3.5 Identifying values for program failure

Based on Section 3.2.3, our implementation makes the following assumptions about values that can lead to processor failures and the times at which such failures occur.

Values to output devices: Our program traces are at the user-level and do not contain output instructions. We conservatively assume that values that are stored in memory are observable externally, and errors in them cause program failure. We assume that the failure occurs when the store instruction retires and is issued to memory.

Fields of an instruction: Errors in all fields of loads, stores, and instructions that change control flow (branches and jumps) are propagated to the retirement queue. These errors are assumed to cause failure when the instruction retires. This is because these errors can change the op code, program control flow, memory addresses, or the value stored in memory, which are assumed to be observable externally. Waiting until retirement to flag a failure ensures that misspeculated instructions do not flag failures.

For instructions other than the above, we do not consider errors in fields that specify source registers to cause failures. Instead, we propagate the errors in these fields into the value in the destination register. Errors in all other fields are considered to cause failure at retirement (similar to loads, stores, and branch instructions).

Fields in iTLB and dTLB: Any errors in the TLBs are propagated to the retirement queue entry of the corresponding instruction, and considered to cause failure on retirement of that instruction. This is because an error in these structures can lead to memory address related failures.

3.4 Experiments and results

In this section, we first apply SoftArch to calculate MTTF and compare with the MTTF calculated with Monte-Carlo methods. Next, we use SoftArch to study the workload behavior for soft errors. Finally, as an application of SoftArch, we use SoftArch to study the effect of technology scaling on the soft error rate at the architecture level.

3.4.1 Compare SoftArch to the Monte-Carlo method

We have shown in Section 2.3.2 that the AVF+SOFR method leads to significant amount of relative error for certain parts of the design space.

SoftArch's probabilistic approach does not require the AVF and SOFR assumptions; it is therefore useful to explore whether SoftArch can be applied to the parts of the design space where AVF+SOFR shows significant discrepancies from the Monte Carlo method. We used SoftArch to estimate MTTF for the entire design space studied in Section 2.3. We found that for every point in this space, the error in MTTF computed by SoftArch relative to the Monte Carlo MTTF is less than 1% for a single component and less than 2% for the full system. Thus, SoftArch does not exhibit the discrepancies shown by AVF+SOFR. These results are not meant to provide a complete validation of SoftArch or a complete comparison between SoftArch and AVF+SOFR (such an analysis is outside the scope of this work). Rather, these results suggest alternative methodologies and motivate future work combining the best of existing methodologies for the most accurate MTTF projections across the widest design space.

3.4.2 A case study with SoftArch

In this section, we show what we can get from SoftArch by applying SoftArch to the modeled processor and SPEC CPU2000 benchmarks (9 integer and 12 floating point). For each benchmark, we use

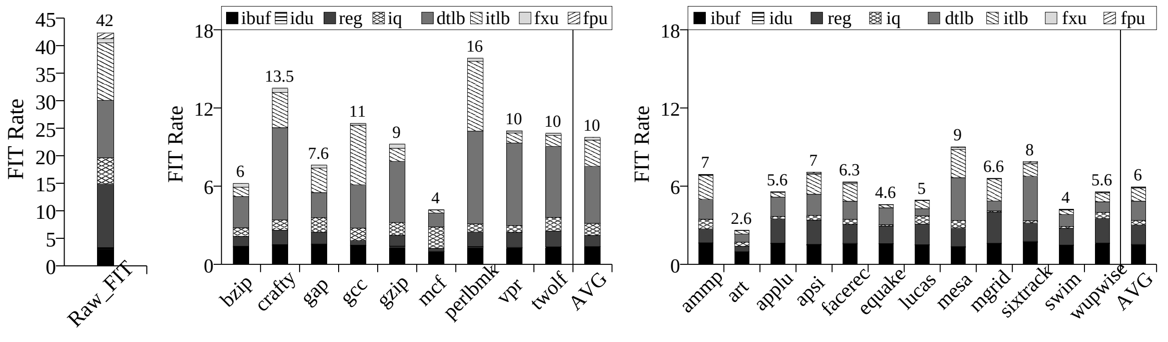


Figure 3.2 FIT rates (a) for raw errors, (b) with architectural masking for SPECint benchmarks, and (c) with architectural masking for SPECfp benchmarks.

sampled traces with 100 million instructions that were validated for acceptable representativeness against the full trace [19].

Metrics

Our experiments report MTTF for an application (Section 3.2.3). We also compute MTTF for individual structures, assuming zero raw SER for other structures. An alternative method of reporting reliability is in terms of FITs. For failure mechanisms with constant failure rate (i.e., exponential distribution for time between failures), FIT rate = 1/MTTF and the FITs of individual system components can be added to give the FITs of the entire system according to the SOFR method. This additive property is convenient when attempting to understand the relative contribution of failure rate and importance of different system components. In Chapter 2, we have shown that although the constant failure rate assumption for raw soft errors is reasonable, the assumption might not hold after the errors are architecturally masked. Nevertheless, we have shown that for SPEC benchmark under current terrestrial raw soft error rate, the SOFR method is valid and the additive property of FIT rate holds well. Our results from SoftArch confirms that the FIT rates across components are indeed additive in this case. Therefore, for convenience and following other literature (e.g., [8]), we report our results in terms of FITs (= 1/MTTF) for the entire system and for each component.

Given the FIT rate and the raw FIT rate of a component, we are able to estimate the amount of architecture level masking effect. We denote the amount of architecture level masking using use

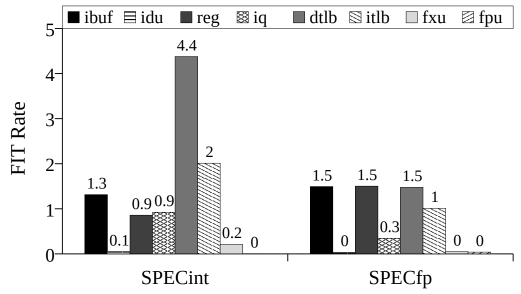


Figure 3.3 FIT rate for each structure, averaged across SPECint and SPECfp benchmarks.

the term *derating factor* which is defined as $\frac{FIT}{rawFIT}$. Derating factor is the same as AVF. In this paper, we use AVF and derating factor interchangeably.

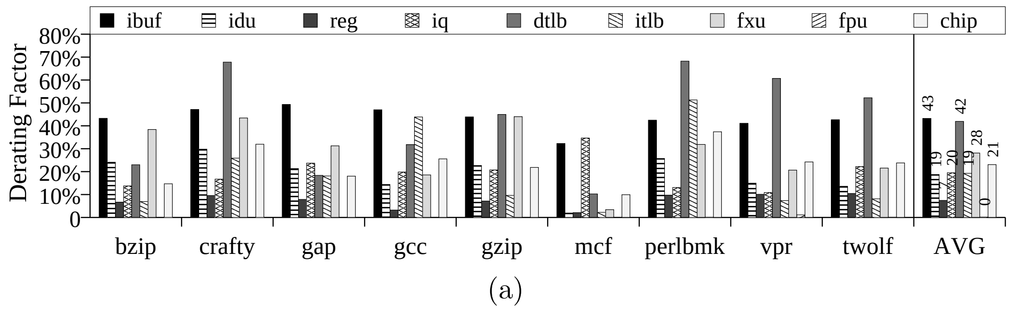
Overall results

Our results are presented in Figures 3.2 – 3.5. Figure 3.2 shows the FIT rate for an entire application. Figure 3.2(a) shows the raw processor FIT rate, which is calculated assuming that each raw error causes a program failure. Figures 3.2(b) and (c) show the FIT rates for our SPECint and SPECfp benchmarks respectively, with the rightmost bars showing the average. Each bar in these figures is further divided to show the contribution to the FIT rates from the different structures – instruction buffer (IBUF), instruction decode unit (IDU), register file (REG), instruction queues (IQ), data TLB (dTLB), instruction TLB (iTLB), integer functional unit (FXU), and floating point unit (FPU).

Figure 3.3 summarizes the structure-wise information by showing the average FIT rate for each structure across the SPECint and SPECfp benchmarks. Figures 3.4(a) and (b) show the architectural derating factors for each structure and the entire processor for SPECint and SPECfp respectively (again, the rightmost bars are the average).

Finally, to understand dynamic application behavior, Figure 3.5 reports the time variation in processor and per-structure FIT rate for two representative applications. We divide each application's execution into intervals of 64K instructions, and plot the FIT rate (Y-axis) for each such interval (X-axis), for each structure and the full processor.

The above data shows the following high level results (these are consistent with prior work,



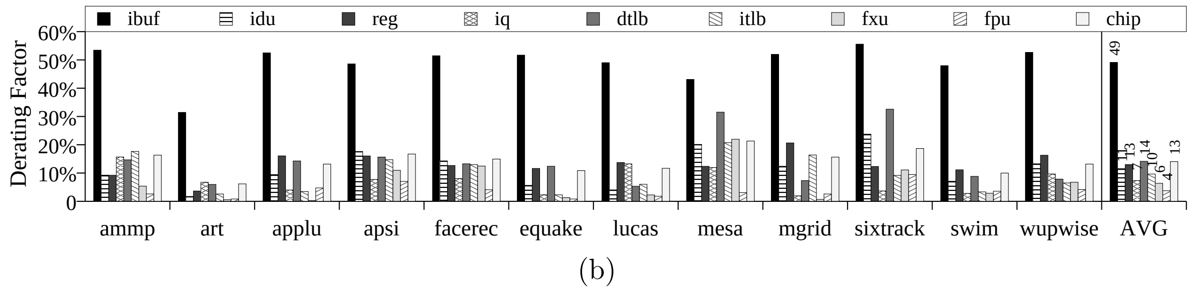


Figure 3.4 Architectural derating factor for each structure (a) for SPECint and (b) for SPECfp benchmarks. Note that the scales on the two graphs are different.

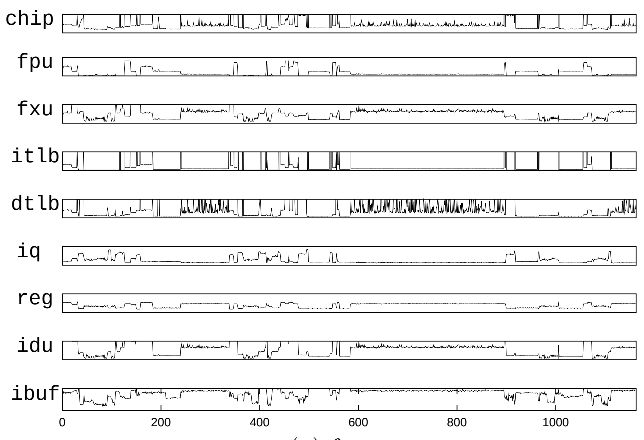
but they are more comprehensive since they cover more structures on chip than [8] and longer application runs than [3]):

Architectural derating. Architectural masking has a large impact on the overall processor FIT rate (Figures 3.2 and 3.4). While the raw failure rate is 42 FITs, the average architecturally masked rate for SPECint and SPECfp is 10 and 6 FITs respectively.[4] Thus, on average, only 21% and 13% of the raw errors cause program failure for the SPECint and SPECfp benchmarks respectively.

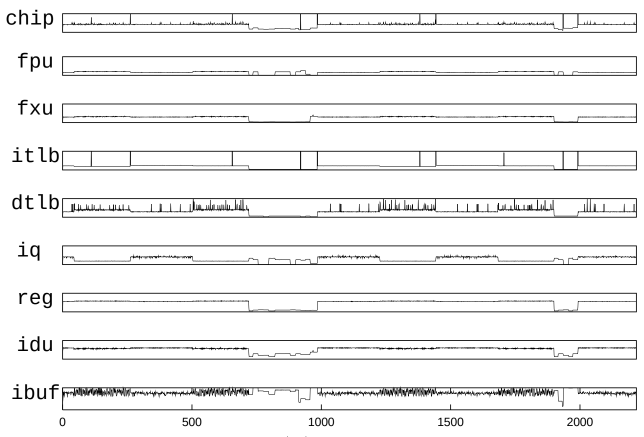
Variation across workloads. Different benchmarks exhibit significant differences in FIT rates, with a range of 2.6 for *art* to 16 for *perlbmk* (Figure 3.2). In general, SPECfp applications have a lower FIT rate than SPECint.

Variation across structures. Different structures contribute in different proportions to the overall FIT rate (Figures 3.2 and 3.3). Although there are workload-specific variations, we can identify general trends. For SPECint applications, the major contributor to the FIT rate is the dTLB followed by the iTLB and instruction buffer. For SPECfp, the major contributors are the instruction buffer, register files, and dTLB, closely followed by iTLB. The logic elements are insignificant and the instruction queues are not a strong contributor to the SPECfp applications.

[4]The absolute FITs may appear low; however, these are for only one processor, at 90nm, for soft errors only due to neutrons, and assume significant protection overhead in the caches.



(a) facerec



(b) wupwise

Figure 3.5 Intra-application variation in FIT rate for intervals of 64K instructions.

Further, Figures 3.2(a) and 3.4 show that the difference in contribution from the structures come both from a difference in the raw SER and in the architectural derating.

Intra-application variation is significant for the overall and per-structure FITs (Figure 3.5).

Analysis

We next describe the reasons for our results. The architectural FIT rate for a structure for a given application is determined by the following three factors for the structure:

Raw FIT rate: This depends on the structure size and the raw SER per bit or logic chain for the technology.

Base utilization: For logic, this is the fraction of time that the structure is used. For storage, this is the fraction of values that are live; i.e., values that will be read before being overwritten or before program termination.

Effective utilization: This is the fraction of values that are read or computed from the structure that contribute to program outcome. For example, if the instruction queues are always full, then their base utilization is high. However, if most of these instructions will be squashed, then the effective utilization is low. The product of the base and effective utilization is the architectural derating factor.

The above factors explain the differences in contributions to architectural FIT rates from the different structures as follows. The instruction buffer and instruction queues have relatively low raw FIT rates due to their small size (relative to the register files and TLBs). However, the instruction buffer has a high derating factor due to its high base and effective utilization; therefore, it is one of the three largest contributors to the architectural FIT rate on average. The instruction queues, on the other hand, have a more modest derating factor, and hence a modest to low contribution to the architectural FIT rate.

For the register file, the raw FIT rate is among the highest. For SPECint, however, its architectural FIT rate is much lower than that of the TLBs because the base utilization of the floating point register file is negligible. For SPECfp, the register file is one of the three largest FIT contributors.

The raw FIT rate of the dTLB and iTLB are the same; however, the dTLB's FIT rate is larger than that of the iTLB for SPECint, and is larger for SPECint than for SPECfp. We consider any erroneous value read from the TLBs to cause program failure; therefore, the above differences occur from the base utilization. Thus, the fraction of values that are live appears higher for the dTLB than for the iTLB for SPECint (likely because of smaller footprint for instructions), and higher for the dTLB for SPECint than for SPECfp (partially corroborated with prior data cache lifetime results).

For the IDU, FXU, and FPU, the main reason for the low contribution to the overall FIT rate is the low raw FIT rate of logic and latches relative to array structures. Some predictions expect this trend to reverse for future technologies [14], in which case the logic elements can be expected to contribute more to the overall SER.

Similar analysis explain the differences between and within workloads. For example, consider *mcf* with its low FIT rate. It is well-known that it spends most of its execution stalled for memory.

Thus, most structures exhibit a small FIT rate because of low base utilization. The instruction buffer and queues, however, contain live instructions stalled for memory, and so show higher derating.

Implications and limitations

The above results have at least three broad implications. First, they motivate selective protection, and can be used to determine which parts of the processor are most cost-effective to protect. Second, they motivate application-aware protection. As shown, different applications have different behavior, both in absolute FIT rate and in the structures that contribute most to the FIT rate. Third, along the same lines, our results show significant variations in FIT rate and in the structures contributing to FIT rate within an application. This is similar to the phase behavior noted in prior studies for other metrics (e.g., IPC, cache miss rate) [20]. These results motivate consideration of dynamic adaptation schemes for managing soft errors, much like adaptation for energy and temperature management.

SoftArch has at least two limitations. First, it depends on architectural timing simulation. Typically, such simulators do not include all microarchitectural and circuit-level details, introducing inaccuracies (e.g., use of e_{logic} and latch/gate count estimates). Second, SoftArch does not simulate changes to the execution path after an error; therefore, it cannot model effects such as application-level masking. Please note that the AVF+SOFR method also has the same set of limitations.

3.4.3 Another application of SoftArch: architecture level scaling analysis

With the SoftArch tool, we are now able to analyze the soft error rate of the processor taking into consideration both the raw error rate and the architecture level masking effect.

The effect of technology scaling on raw error rates for different type of circuits has been extensively studied. However, there has been no previous work examining the effect of scaling on processor SER considering architectural derating effects. In this section, we apply SoftArch to quantify the impact of technology scaling on the architecture level processor soft error rate, taking the architecture level masking effects and workload characteristics into consideration.

In our experiments, we scale the same design with the same number of transistors over four

Tech	**Freq**	**Vdd**	**Off-chip Lat**	λ (FIT/bit)	FPU e_{logic}	FXU e_{logic}	IDU e_{logic}
180nm	1.1GHz	1.8 V	77 cycles	$5.7 * 10^{-4}$	$1.45 * 10^{-22}$	$1.06 * 10^{-22}$	$7.61 * 10^{-23}$
130nm	1.35GHz	1.5 V	94 cycles	$6.0 * 10^{-4}$	$9.96 * 10^{-23}$	$7.34 * 10^{-23}$	$5.25 * 10^{-23}$
90nm	1.65GHz	1.2 V	115 cycles	$7.4 * 10^{-4}$	$5.97 * 10^{-23}$	$4.40 * 10^{-23}$	$3.15 * 10^{-23}$
65nm	2.0GHz	0.9 V	140 cycles	$7.1 * 10^{-4}$	$3.26 * 10^{-23}$	$2.40 * 10^{-23}$	$1.73 * 10^{-23}$

Table 3.1 Scaling parameters for the simulated processor.

technology generations ranging from 180nm to 65nm. We find that with scaling, the derating factors for logic structures often decrease, the derating factors for storage elements remain roughly unchanged, and the FIT for the full processor roughly follows the trend for the raw SER of storage structures (i.e., the FIT rate increases from 180nm to 90nm and decreases from 90nm to 65nm.) Please note that this result is valid only when the transistor number is constant during the scaling process. In reality, we expect the transistor count to increase and the overall FIT rate per chip to increase.

Scaling Methodology

The parameters of the base processor we simulate are the same as in Table 2.1. We study the architecture level FIT rate for the modeled processor for four technology generations, ranging from 180nm to 65nm. We assume that there are no modifications to the processor micro-architectural pipeline with scaling. Effectively, we scale the same chip from 180nm to 65nm technologies.

Table 3.1 summarizes the parameters that change with scaling. Although with ideal scaling, the best base frequency scaling per generation should be about 43%, it is hard to achieve the ideal frequency boosts without significantly re-tuning all the circuit delay paths in the processor. Therefore, we conservatively assume 22% frequency scaling per generation. Since everything on chip is scaled, we assume that the on-chip storage structures such as register files, instruction queues, TLBs, and caches scale linearly with the transistors and their access times *in terms of processor cycles* stay the same. For the off-chip L3 cache, we assume that the *absolute* access time stays the same and therefore, its access time in terms of processor cycles increases about 22% for each generation.

Table 3.1 also gives the scaled values for the raw SER for storage structures (denoted as λ) and logic circuit (denoted as e_{logic}). The base case (180nm) parameters have been estimated in

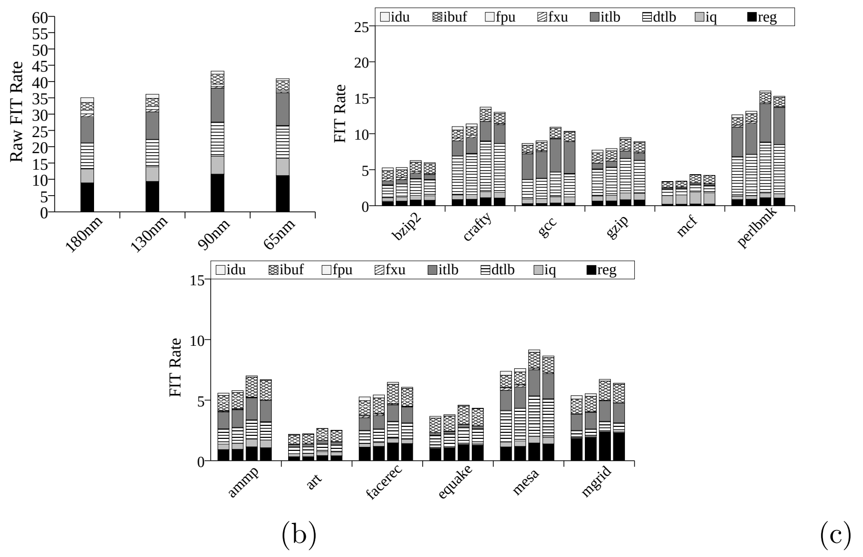


Figure 3.6 FIT rates (a) for raw errors, (b) with architecture masking for SPECint benchmarks, and (c) with architecture masking for SPECfp benchmarks.

Section 3.3. For storage structures, we then scale the raw SER for different technologies using the scaling curve provided by Karnik et al. [18]. As shown in Table 3.1 the raw SER (λ) of storage elements increases as technology scales down from 180nm to 90nm and then decreases slightly from 90nm to 65nm. For logic structures, we use the same methodology as described in Section 3.3.3 to determine the raw SER for each technology generation. The raw logic SER (e_{logic}) decreases with scaling. This is because logic error rate is dominated the SER by latches and the SER for latches will decrease with future technology scaling.

Below we report experimental results for 12 SPEC CPU2000 benchmarks including 6 integer benchmarks and 6 floating point benchmarks.

Results

Our results are presented in Figures 3.6 and 3.7. Figure 3.6 shows the FIT rate for the processor. Figure 3.6(a) shows the raw processor FIT rates which are calculated assuming that each raw error causes a program failure for the four technology generations. Figures 3.6(b) and (c) show the FIT rates for our SPECint and SPECfp benchmarks respectively. Each group consists of four bars which are for four technology generations starting from 180nm to 65nm. Each bar is further divided to show the contribution to the FIT rates from the different structures.

Figures 3.7 (a) and (b) show the architectural derating factors for each structure and the entire processor for the SPECint and SPECfp benchmarks respectively for the four technology generations.

In view of the inaccuracies in our method of estimating the raw SER values (Table 3.1), the focus of the results presented here is not on absolute FIT rates which are almost certainly inaccurate. Instead, the goal of the ensuing analysis is to show the trends with scaling. We believe these trends are reasonably accurate.

Our high level results are the following:

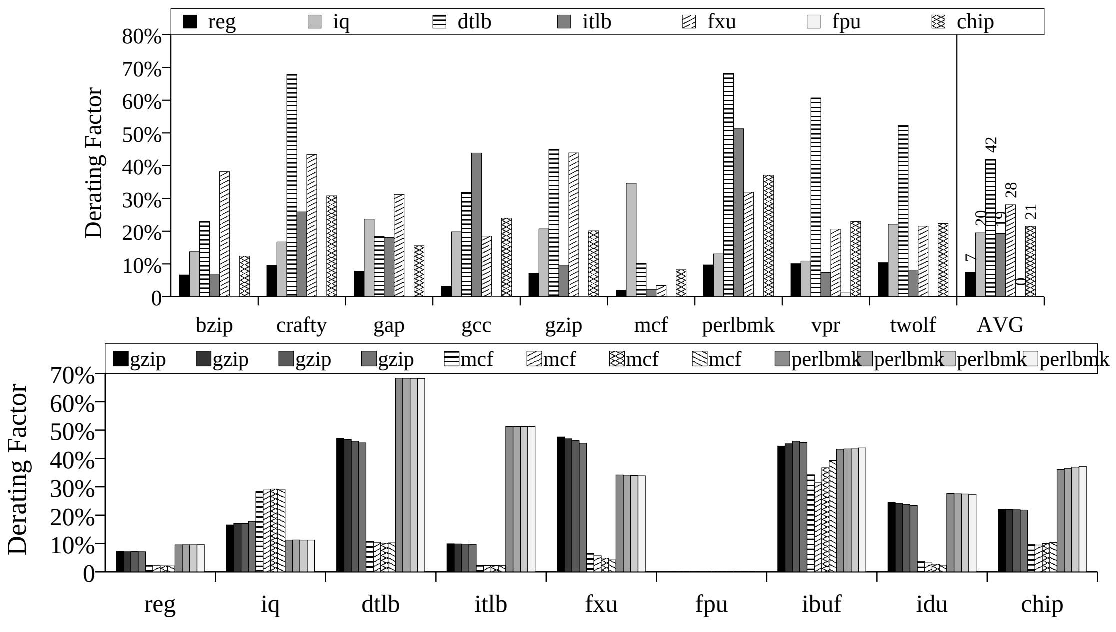
FIT rate scaling: The FIT rate of the whole processor increases as technology scales from 180nm to 90nm and decreases slightly from 90nm to 65nm. The reason is that the dominating source of the FIT rate is the storage structures. The logic FIT rate is insignificant compared to the storage element FIT rate. From 180nm to 90nm, the FIT rate of storage structures increases. From 90nm to 65nm, the FIT rate of storage structures decreases slightly.

Derating factor scaling: (1) The derating factor of logic structures (FPU, FXU, IDU) decreases as technology scales down. (2) The derating factor of storage elements does not change much with technology scaling and increasing memory latency. This is the case for both SPECint and SPECfp applications.

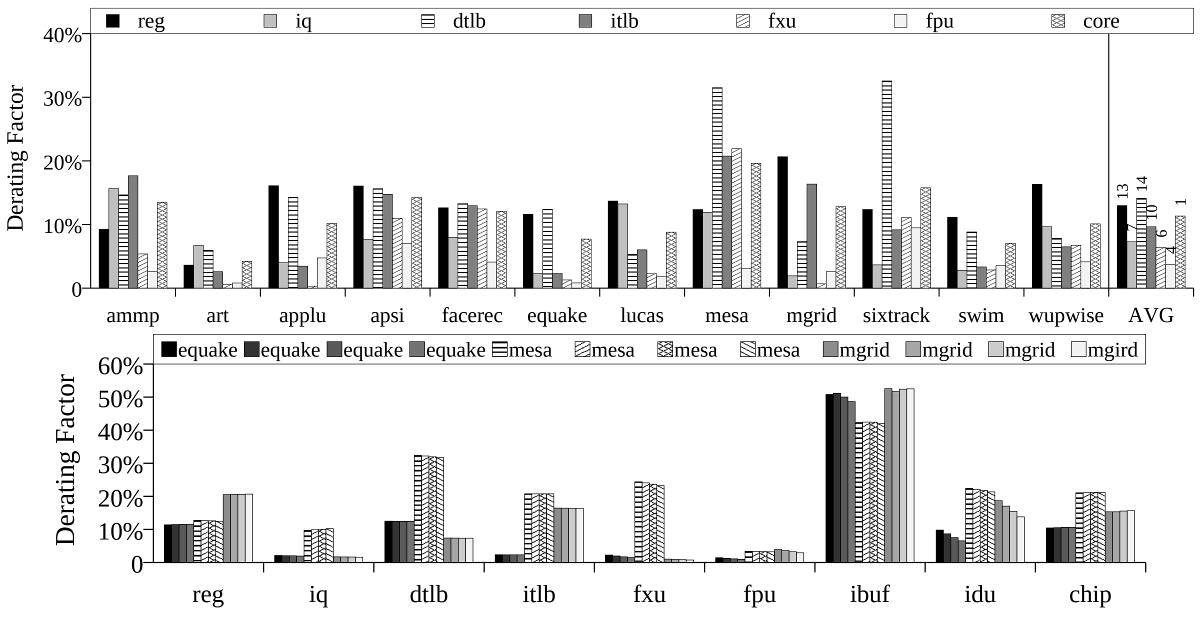
Analysis

To help explain the results, we use a simple model to analyze the FIT rate and derating factor scaling trends. As discussed in Section 3.4.2, the FIT rate for a given structure is determined by three factors: *raw FIT rate for the structure, base utilization of the structure,* and *effective utilization of the structure.* The derating factor is only determined by the latter two factors. The raw FIT rate factor depends on the technology. For storage structures, the base utilization is the fraction of values that are alive. For logic structures, the base utilization is the fraction of time the structure is used. The effective utilization of a structure is the fraction of values that are read or computed from the structure that affect the program outcome.

The scaling of raw FIT rate has been summarized in Section 3.4.3. Next we will analyze the scaling trend of the base utilization factor. It can be expressed as T_{busy}/T_{exec}. Here T_{exec} is the total execution time of the program. For logic structures, T_{busy} is the time the structure is used.



(a) SPEC int



(b) SPEC fp

Figure 3.7 Architectural derating factor for each structure (a) for SPECint and (b) for SPECfp benchmarks. Note that the scales on the two graphs are different. For each application, the four bars in a graph represent the four technology generations, going from 180nm to 65nm.

For storage elements, T_{busy} is the time that the element holds live values. This is equivalent to $Cycles_{busy}/Cycles_{exec}$. Here $Cycles_{exec}$ is the number of cycles for program execution. $Cycles_{busy}$ for a logic structure is the number of cycles the structure is busy. $Cycles_{busy}$ for a storage element is the number of cycles the element has live data.

From 180nm to 65nm, the processor frequency increases 22% every generation. If there is no memory access, the value of $Cycles_{exec}$ would stay the same. But for real applications, $Cycles_{exec}$ typically increases because the increasing memory latency would delay the program execution. Figure 3.8 shows the increase in number of cycles for each application when the memory latency increases from 77 to 140 cycles. From Figure 3.8, the execution time of most integer applications is not very sensitive to the memory latency (except *mcf*), while the execution time of most floating point applications is more sensitive.

The scaling of the $Cycles_{busy}$ value is more complex. Next, we will discuss the scaling for logic and storage structures separately.

For logic structures (FPU, FXU, IDU), although the processor frequency changes, the number of committed instructions and the instruction sequence stays the same. Thus the number of operations that are critical to the outcome of the program will not change. For example, for each technology generation, there would be the same number of FPU operations and FXU operations that are critical to the program outcome. Thus, the value of $Cycles_{busy}$ would be the same for logic. As a result of the increase of $Cycles_{exec}$, the derating factor of logic would decrease.

For storage elements (reg, IQ, TLB, IBUF), $Cycles_{busy}$ is the number of cycles data is live in the element. It tends to increase with technology scaling because the memory latency gets larger from 180nm to 65nm. According to our experiments, the rate of increase of $Cycles_{busy}$ and $Cycles_{exec}$ is similar. Therefore, the increases roughly cancel out with each other and the derating factor stays the same.

Scaling summary

Based on on the scaling trend of the above factors, we explain the scaling trend of the derating factor and the FIT rate as follows:

Scaling trend of the derating factor: The derating factor depends on the utilization factor.

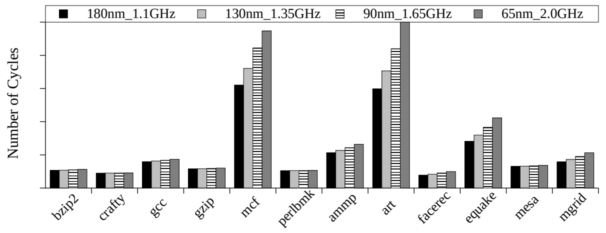


Figure 3.8 $Cycles_{exec}$ **for each application for different frequencies (technologies)**

As the technology scales down from 180nm to 65nm, the derating factor of the storage elements stays roughly the same, while for logic, the derating factor gets smaller.

Scaling trend of the FIT rate: The processor FIT rate depends on the raw FIT rates and the derating factors. For the processor and the technologies we modeled, storage elements FIT rate dominates and the logic FIT rate is insiginifant. Since the derating factor for storage elements does not change, the FIT rate of the whole processor follows the same trend as the raw FIT rate. It increases from 180nm to 90nm and decreases slightly from 90nm to 65nm.

3.5 Summary

In this section, we have presented SoftArch, a model and tool for studying and analyzing architecture level soft error behavior of modern processors. SoftArch can be integrated into high-level performance simulators and used to (1) determine the architecture level soft error MTTF of a processor running a specified workload, (2) identify the soft error contributions from various microarchitectural structures, and (3) study the soft error contributions of different phases of an application. We first show that SoftArch does not need to make the assumptions that AVF+SOFR does, thus does not exhibit the discrepancies shown by AVF+SOFR. Then we demonstrate the use of SoftArch by applying it to a modern out-of-order processor running SPEC2000 benchmarks. We use SoftArch to study workload behavior and show significant architecture level derating and large variations of soft error failure rate across workloads, processor structures, and within the same workload. Finally, as another application, we apply SoftArch to quantify the impact of technology scaling on the architecture level processor soft error rate, taking the architecture level masking effects and workload characteristics into consideration.

Chapter 4

Online estimation of the AVF

4.1 Introduction

As we have mentioned in Chapter 2, the AVF of a structure is defined as the probability that a visible error (failure) will occur, given a raw error event in the structure. AVF is a simple abstraction for the amount of architecture level masking in the processor and can be used to estimate the MTTF. We have shown in Chapter 2 that although the AVF method has strong assumptions, for a large class of systems and workloads, including those that will be studied in this section, the AVF method is accurate and AVF value 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 be able to estimate the AVF in the design stage to meet the reliability goal of the system.

Many soft error 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.

Furthermore, the results in Section 3.4.2 show significant intra-application variation in the AVF, which motivate the need to estimate AVF at runtime as well. 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. [21] propose to use the AVF input to control instruction throttling and selective redundancy schemes. They show that using AVF-controlled selective redundancy scheme, the AVF of the re-order buffer can often be reduced by more than half (which means that the MTTF more than doubles) with a relative small performance degradation. 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 are some previous studies that provides online AVF estimation; however, they are either dependent on extensive offline workload analysis [22] or targeted to a single structure [21] (see Section 5.2). 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 dissertation, we describe a general online method to estimate AVF for a variety of structures (including logic and storage structures) without the 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 is determined to potentially result in failure provides 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 less accurate estimates. The parameters should be set based on the required estimation precision. The needed accuracy of the AVF estimation depends on how the AVF value will be used. For example, Walcott et al. [22] propose to use the AVF estimation to enable/disable the redundant multithreading protection scheme. In this case, the AVF is used to make a binary decision. As long as the AVF estimation is below or above a certain threshold, the decision will be the same. Thus, there is no need for a very high precision estimation. However, for the instruction throttling scheme proposed by Soundararajan et al., since the throttling amount is a continuous variable that can take any value between 0 and 100%, even a small error in the AVF will cause the throttling amount to change. The required AVF precision in this case is higher. Our AVF estimation scheme is flexible and can be configured based on the needed precision. In Section 4.2, we will show how 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 them with the results SoftArch, which is a more detailed (but complex) offline AVF estimation method (see Section 5.2). The 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 with 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.

4.2 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.

4.2.1 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.

Algorithm 1 Algorithm to estimate AVF for a structure

```
1: Set the counters $injectionCount = 0$ and $failureCount = 0$
2: while $injectionCount < N$ ($N$ is a predetermined threshold) do
3:     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.
4:     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.
5:     If the processor failure bit is set, $failureCount = failureCount + 1$.
6:     Clear all error bits in the processor.
7:     $injectionCount = injectionCount + 1$.
8: end while
9: AVF = $\frac{failureCount}{injectionCount}$
```

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.

4.2.2 Determining potential failure

In reality, an error causes program failure only if it propagates to the program output. Unfortunately, similar to SoftArch, 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 some pre-defined failure points. In this study, we have use the same failure points as we have defined in Section 3.3.5.

4.2.3 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:

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

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 $E(\bar{X}) = E(X)$ [11].

Independence of the samples can be ensured using random sampling; i.e., by using a random number generator to determine the error injection time. Many hardware random number generators are very complex. There are some simpler pseudo-random number generators such as the linear feedback shift register(LFSR) [23]. LFSR can generate a pseudo-random number in between 0 and 2^n. In our case, if we need a random number between 0 and any m, we will still need some more hardware to transform the original random number. 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

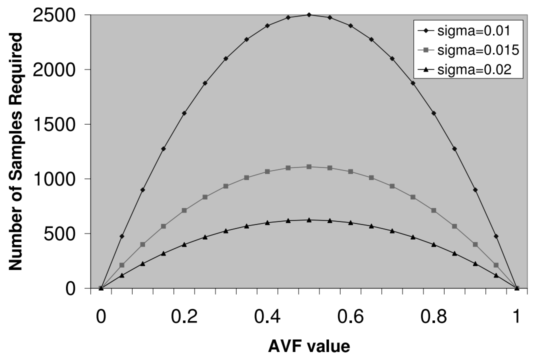


Figure 4.1 The number of samples N needed for different values of AVF and estimation precision of $\sigma_{\bar{X}}$.

all X_i are i.i.d. [11]. 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

$$N = \frac{\sigma_X^2}{\sigma_{\bar{X}}^2} \tag{4.1}$$

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 4.1 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 4.1 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 might be very 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 [11]
$\sigma_{\bar{X}} = \sqrt{\frac{\sigma^2_{X^1}+\sigma^2_{X^2}+..+\sigma^2_{X^K}}{N*K}}$.

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.

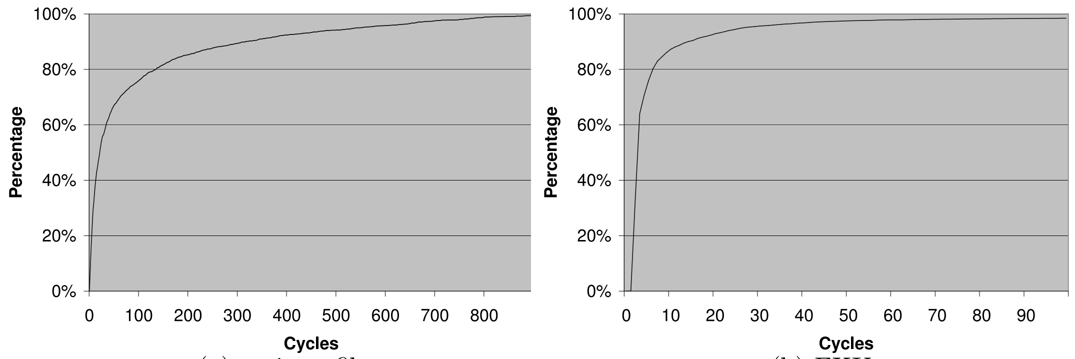
Above we also assume that all the samples are identically distributed. However, we know that workload behavior may change significantly over long intervals of time. If the estimation interval includes such large-scale changes, then we can think of the interval as consisting of multiple phases (each with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

4.2.4 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



(a) register file (b) FXU

Figure 4.2 The cumulative distribution for the time taken by an error to propagate to points of potential failure (defined in Section 4.2.2) for *bzip2*.

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 4.2 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 4.2. 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.

4.2.5 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.

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.

4.2.6 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.

4.3 Experimental methodology

To evaluate the accuracy of our AVF estimation method, we again use the Turandot simulator [13]. The parameters for the processor are chosen to correspond to the POWER4 microarchitecture and were the same as in Table 2.1.

We implemented our AVF estimation algorithm in Turandot as described in Section 4.2 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]

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 [24], 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 [19].

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).

[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.

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.

4.4 Results

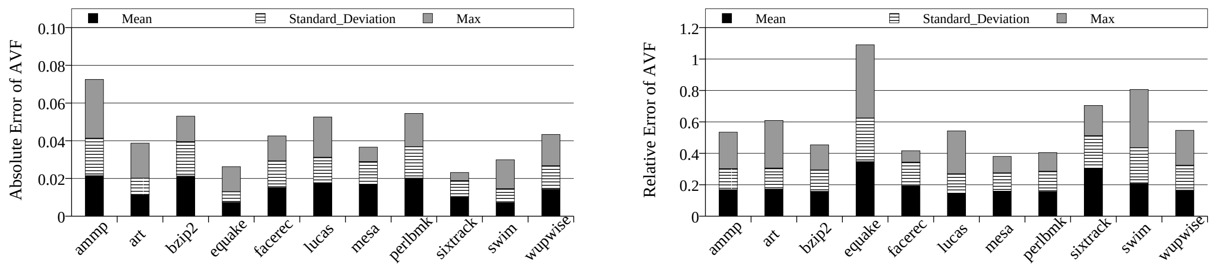
Figures 4.3(a), (b), (c), and (d) show aggregate statistics to demonstrate the accuracy of our AVF estimation algorithm relative to SoftArch for the instruction queue, register file, FXU, and FPU respectively. The FXU and FPU figures also show the accuracy of the simple utilization-based estimation method 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*.

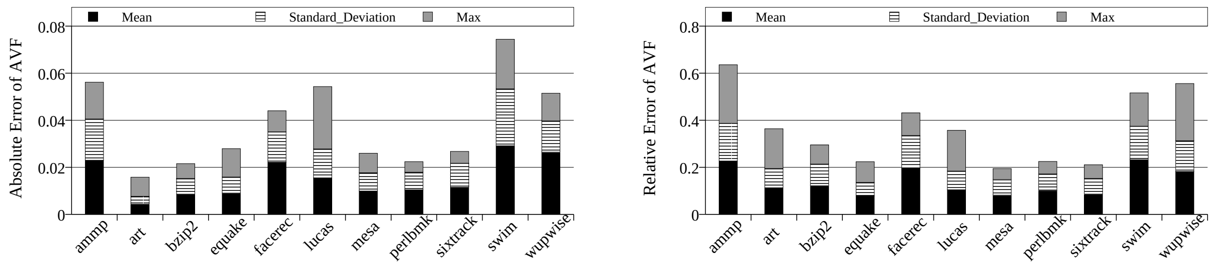
The charts on the left side of Figure 4.3 give three statistics for the absolute errors. For each bar, the lowest (shaded) stack gives the mean absolute error (referred to as Mean) for the corresponding estimation method and application (averaged across the different 1M cycle estimation intervals for 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 4.3 provide the relative errors for reference.

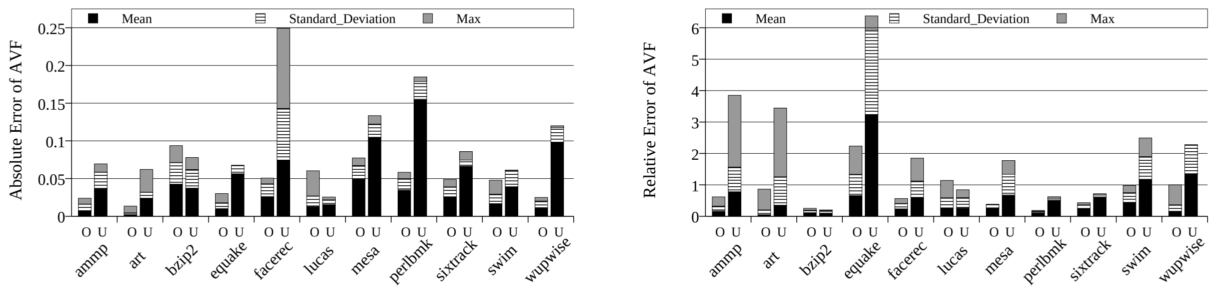
For a more detailed look, we take two applications as examples and plot AVF values for them



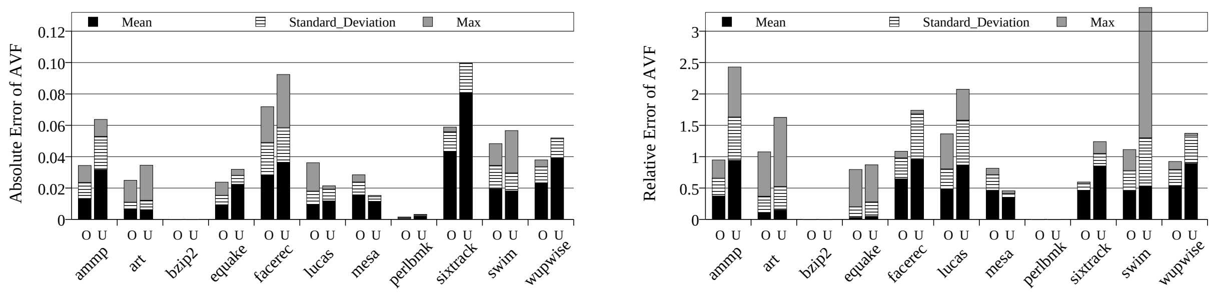
(a) Instruction queue



(b) Register file

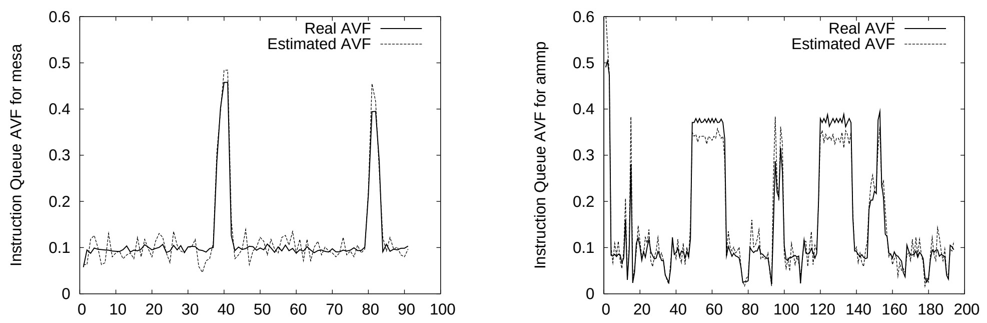


(c) FXU

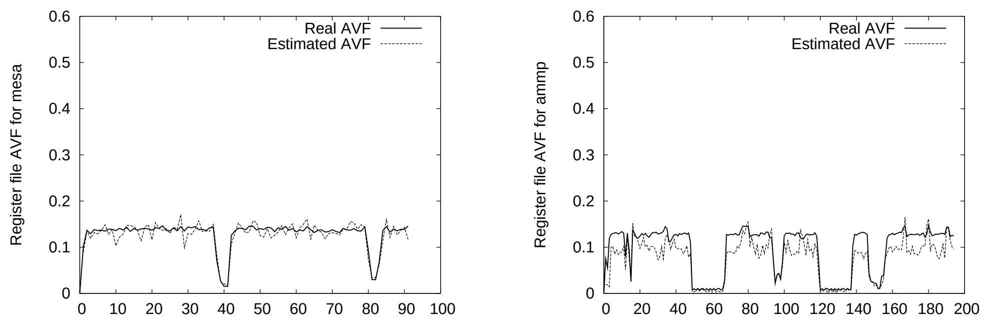


(d) FPU

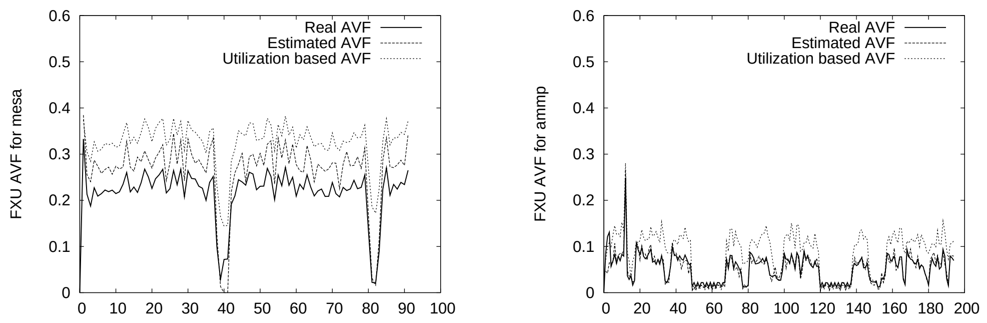
Figure 4.3 Error in AVF estimation when compared to the SoftArch reference for (a) instruction queue, (b) register file, (c) FXU, and (d) FPU. The left charts show *absolute error* - mean, standard deviation and maximum - across all estimation intervals of the application. The right charts show *relative 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).



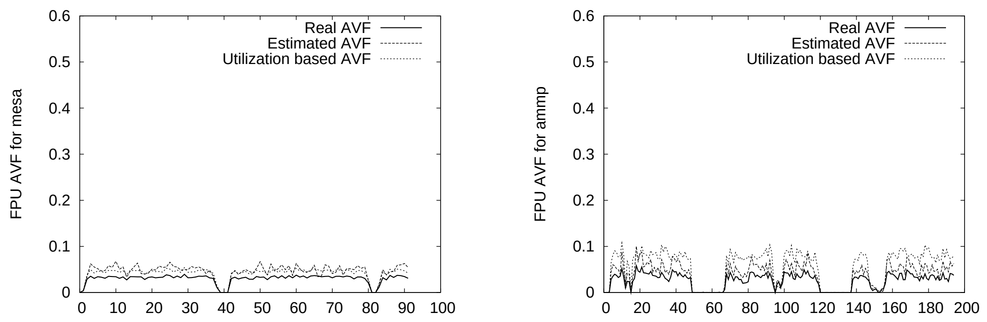
instruction queue



register file



FXU



FPU

Figure 4.4 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.

for each 1M cycle estimation interval for each structure in Figure 4.4. 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 4.3), 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 not randomly selected for fault injection.

Relative errors.

Comparing relative errors (right charts in Figure 4.3), 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.

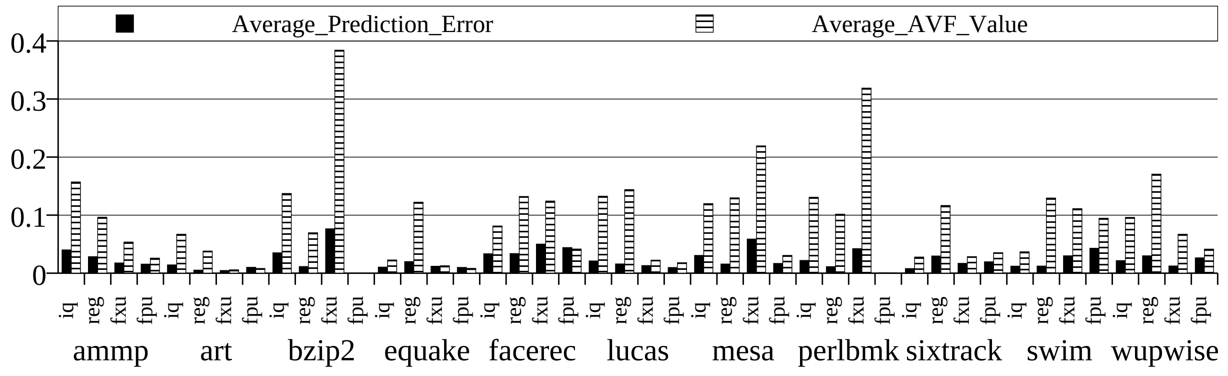


Figure 4.5 The relative error of the predicted AVF using a simple predictor. The predictor assumes the AVF of the next interval is equal to that of the previous interval.

Detailed results.

The detailed plots in Figures 4.4 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 dissertation, 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 4.5 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 dissertation. 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. Fu et al. [25] show that AVF exhibits phase behavior similar to the performance and power domain. They show that the AVF behavior is mostly related to the program code-structure and run-time events. Thus, the stable AVF behavior across multiple intervals might be explained by the the underlying similar code-structure such as code running in the same loop. Similarly, a sudden change of AVF might indicate that the program has entered a new phase.

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.

4.5 Summary

In this chapter, 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 test 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.

Chapter 5

Related work

We classify related soft error work into three main categories: soft error modeling, AVF estimation, and soft error protection solutions. The first class of work models and investigates the soft error behavior of a processor at the architecture level. The second class estimates and predicts the AVF for a running application. The third class proposes new solutions to protect the processors from soft errors.

In Section 5.1 to 5.3, we describe each of the three categories. In Section 5.4, we discuss other related work that bears similarity to our online AVF method.

5.1 Soft error modeling

There have been two broad approaches to architecture level modeling of the impact of soft errors. The first involves fault injection in a simulator to determine whether an injected error is exposed at the architecture level [26, 27, 3]. For example, Wang et al. perform fault injection experiments on a latch-accurate Verilog model of a modern Alpha processor (about 25,000 experiments for each benchmark). This kind of approach is accurate, but slow. The typical simulation is limited to the order of 10,000 cycles for an application's execution.

Mukherjee et al. propose the AVF method to calculate the MTTF of a processor [8]. The average fraction of bits in a structure that will affect the program outcome is termed as the architecture vulnerability factor (or AVF) for that structure (equivalent to the derating factor). The product of AVF and the raw SER for a structure gives its architectural failure rate. Biswas et al. [9] extends the method to cover address-based structures as well.

To our knowledge, there has been no prior attempt to understand the basic assumptions of the

AVF+SOFR method and parameter value ranges that bound its validity (or accuracy), when it comes to reliability modeling at the (micro)architecture level.

5.2 AVF estimation

There have been several studies on estimating the AVF [8, 6, 28, 22, 21]. They can be classified into two categories.

The first category is the offline method which estimates AVF with complex simulators [8, 6, 28]. 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.

The second category is the online method which estimates the AVF in real time [21, 22]. Walcott et al. [22] 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. What is more important is that the parameters are dependent on the workload as mentioned in the paper. It is not clear that the parameters calibrated for one set of workloads will give accurate estimation for another set. Compared to Walcott et al., our online AVF estimation approach requires little offline analysis. It only requires two parameters for any structure compared to five to ten parameters in Walcott et al. The parameters in our method can also be chosen to achieve the best trade-off between the estimation precision and estimation time. Soundararajan et al. [21] 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.

5.3 Soft error protection schemes

There has been a rich body of work in the soft error field on soft error protection schemes. Here we will focus on some of the key schemes that address the soft error problem from the software level and architecture level.

On the software level, the compiler or operating system can help introduce redundancy into the program to check program control flow, memory access, and control signals. For example, Ohlsson et al. [29] propose that the compiler automatically generate some code for a watchdog processor to check the protected processor. Along with the recent popularity of SMT and CMP processors, there have been several papers dealing with soft error protection using redundant threads. SMT and CMP are able to execute two threads simultaneously, which creates an opportunity for thread level redundancy. There have been several papers on the implementation of redundant multi-threading. Depending on the platform (SMT vs. CMP), detection or recovery, different schemes have been proposed. Some examples are: AR-SMT [30], SRT [31], CRT, Slip-stream, SRTR, CRTR, etc. The basic idea is very simple – to run two copies of the same thread and check with each other before the result can be committed to the architecture state. All the proposed schemes require small amount of modification to the original SMT or CMT processor. Oh et al. [32] propose a technique called EDDI where all instructions are duplicated and appropriate "check" instructions are inserted for error checking. There are other variants of the scheme, such as control-flow checking scheme (CFSCC) where each control transfer generates a run-time signature that is validated by error checking code generated by the compiler for each block. Reis et al. [33] propose a software based technique called SWIFT. SWIFT is a single-threaded software-based error detection approach and improved version of EDDI. It uses more optimization to cover more kinds of errors and reduces the overhead of the simple EDDI scheme.

On the micro-architecture and architecture level, redundancy can also be introduced to protect the processor. Austin et al. [34] propose DIVA. The idea is to use a simple and reliable checker processor at the commit stage of the main core to verify its result. Since the checker sits at commit stage, it's an in-order core and can be made very reliable because of its simplicity. Ernst et al. [35], propose Razor, a special latch to protect the data path. The idea is to use a shadow latch to redundantly latch the data and compare with the main latch. When timing error happens, the shadow latch will latch in data different from the main latch, thus errors can be detected. Not only can the Razor latch detect timing errors, it can also provide a mechanism to recover from the timing errors. Recovery can be achieved using methods including clock gating.

Wang et al. [36] propose a scheme called ReStore to detect symptoms of soft errors and use existing recovery ability for branch mispredictions in the processor. Racunas et al. [37] detect soft errors by monitoring for departures from expected program behavior. The SWAT system detects a variety of faults by monitoring for high level symptoms, including software invariant violations [38, 39].

5.4 Other related work

Our AVF prediction scheme has some similarity to the work by Fields et al. on critical-path prediction [40]. Like the AVF prediction, finding the critical-path in the instruction dependence graph is a very complex task even for offline processing. It is even harder to find the critical paths in hardware online. Fields et al. propose an online critical-path predictor that passes tokens to explicitly track dependence chains. Instead of seeking some heuristic events in the pipeline that are correlated with the instruction criticality [41], Fields et al. directly observe and track dependence chains in hardware. Using a probabilistic approach, Fields et al. avoided the complex tracking and analysis. They are able to predict the critical-path in real time with more than 80% accuracy using a hardware predictor. Our work on the AVF prediction is similar in that we also use a probabilistic approach. Instead of tracking all the values in the processor, we use a simple error injection and counting process to estimate the AVF.

There have been at least two major studies that use bits similar to the error bits used in our online AVF estimation scheme, but for different purposes. First, Weaver et al. [4] propose 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 [42] and the analogous NaT bit of the Itanium architecture [43] 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.

Chapter 6

Conclusions and future directions

This chapter summarizes the analysis and insights of this dissertation and discusses future directions for research motivated by the results of this dissertation.

6.1 Conclusions

The continuous scaling of CMOS technology brought tremendous improvement in processor performance. However, soft errors are becoming an increasing concern for processor reliability. In this dissertation, we address soft error reliability issues from an architectural perspective.

We first analyze the current state-of-the-art in soft error modeling and analysis techniques – the AVF+SOFR methodology. Our results show that both the AVF step and the SOFR step make significant assumptions. We then use both mathematical and experimental techniques to check the validity of the above method across a large design space. We find that the above method is valid for most cases under the current raw error rates. However, our results show that for some combinations of large systems, long running workloads with large phases, and/or high raw error rates, the MTTF calculated using the AVF+SOFR method shows significant discrepancies from that using first principles.

To find an alternative model that is not subject to such limitations, we propose a model and tool called SoftArch that does not make the above AVF+SOFR assumptions. SoftArch is based on a probabilistic model of error generation and propagation process in a processor. We show that SoftArch does not show the discrepancies shown by the AVF+SOFR method. We also apply SoftArch to analyze the effect of technology scaling on the processor soft error rate. We scale a processor over four technology generations and identify the trend of the processor FIT rate taking the architecture level masking effect into consideration.

By using the SoftArch tool, we observe that there is much architecture level masking and that the degree of such masking can vary significantly across workloads, individual units, and workload phases. Thus, it is natural to consider the architecture level solutions to take advantage of such variations. To do that, one would need an reasonably accurate estimates of the amount of masking effect in real time. We have shown in our analysis that for most current systems (that are the focus of our study), AVF proves an accurate abstraction of the architecture masking effect. In this dissertation, we put forward a novel way of estimating AVF in real time. We propose some simple hardware modifications for the processor and use an algorithm to effectively estimate AVF. It is a general method that applies to both logic and storage units on the processor. Compared to previous methods for estimating AVF, our method does not require any offline simulation, nor does it require any calibration for different workloads. We test our method with a widely used simulator from industry for SPEC benchmarks. The results show that our method provides accurate run-time AVF estimates. 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.

6.2 Future directions

This dissertation lays the foundation for architecture level analysis of soft errors and provides new tools and techniques to handle this critical emerging technology challenge. There exist many potential avenues for future work. Below we will highlight two possible directions.

6.2.1 Architecture level solution for soft errors

The ultimate goal of the architecture level soft error study is to design effective architecture level solutions to help mitigate the problem. Our results in both Section 3.4 and Section 4.4 show that the AVF for different processor structures changes over time during the program execution. Depending on the AVF, the optimal protection scheme for these structures also changes over time. This provides opportunities to dynamically adapt the processor to achieve the best protection in a cost effective way.

A good prediction of AVF is important for such adaptation. The AVF prediction has to be real time in order for the processor to react to application behavior changes quickly. In this dissertation, we have proposed an effective online AVF prediction method. This could lead to a class of real time AVF adaptation and optimization solutions for processors.

6.2.2 Unified system wide adaptation framework

Many different adaptation schemes have been proposed for power, temperature, energy, and reliability in the past years. Although the power, temperature, energy and reliability problems are highly related inherently, the schemes have been proposed independently for each field so far. Power, temperature, energy and reliability are all important requirements for a computer system and the adaptation for any of them could potentially have an impact on the performance. Considering only one and ignoring the others could potentially cause problems. For example, most soft error protection schemes use some form of redundancy to detect and correct errors which will lead to an increase in the power and energy consumption. This might cause the power and energy constraints to be violated even though the soft error design specifications are met. Similarly, many power and energy reduction techniques such as dynamic voltage scaling will increase the processor soft error rate dramatically which may result in a non-reliable system.

Most previous adaptation schemes have been proposed for a single component of the system to achieve the component's optimal configuration. It would be beneficial, however, to adapt all the components jointly to achieve global optimal adaptation. For example, Li et al. [44] study the energy management problem for the memory considering the performance constraint. Li et al. [45] take one step further and study the joint adaptation problem for both processor and memory. The work shows that by jointly adapting different components of the system, more energy savings can be achieved.

As such, we should consider the adaptation problem as a multiple dimension problem with multiple tradeoffs and requirements in terms of performance, power, temperature, energy and reliability. We need to study and better understand the relations between different dimensions. We also need to better understand the interaction between adaptation for different components of the system. A unified framework for performance, power, temperature, energy, and reliability would potentially have a huge benefit, leading to optimal performance/power/temperature/energy/reliability tradeoffs based on the requirements of the target system.

References

[1] T. J. O'Gorman *et al.*, "Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 41–50, 1996.

[2] J. F. Ziegler, "Terrestrial Cosmic Rays," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 19–39, 1996.

[3] N. Wang, T. Rafacz, J. Quek, and S. J. Patel, "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.

[4] C. Weaver *et al.*, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," in *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[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, S. Adve, P. Bose, and J. A. Rivers, "SoftArch: an Architectural Level Tool for Modeling and Analyzing Soft Errors," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[7] X. Li, S. Adve, P. Bose, and J. A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[8] S. S. Mukherjee *et al.*, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proc. of the 36th Intl. Symp. on Microarch.*, 2003.

[9] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing the Architectural Vunerability Factor for Address-Based Structures," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2005.

[10] K. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice Hall, 1982.

[11] S. Ross, *A First Course in Probability.* Prentice Hall, 2001.

[12] X. Li *et al.*, "Architecture-Level Soft Error Analysis: Examining the Limits of Common Assimptions (extended version)," Tech. Rep. UIUCDCS-R-2007-2833, UIUC, March 2007. Available at http://rsim.cs.uiuc.edu/Pubs/07dsn-tr.pdf.

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

[14] P. Shivakumar *et al.*, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[15] C. Moore, "The POWER4 System Microarchitecture," in *Microprocessor Forum*, 2000.

[16] F. Irom, F. F. Farmamesh, A. H. Johnson, G. M. Swift, and D. G. Millward, "Single-Event Upset in Commercial Silicon-on-Insulator PowerPC Microprocessors," *IEEE Transactions on Nuclear Science*, vol. 49, pp. 3148–3155, Dec. 2002.

[17] G. M. Swift *et al.*, "Single-Event Upset in the PowerPC750 Microprocessor," *IEEE Transactions on Nuclear Science*, vol. 48, pp. 1822–1827, Dec. 2001.

[18] T. Karnik, P. Hazucha, and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 128–143, June 2004.

[19] 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.

[20] T. Sherwood *et al.*, "Phase Tracking and Prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[21] 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.

[22] K. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic Prediction of Architectural Vulnerability From Microarchitectural State," in *Proceedings of the International Symposium on Computer Architecture*, 2007.

[23] S. W. Golomb, *Shift Register Sequences (Revised edition)* . Aegean Park Pr, 1981.

[24] M. Moudgill *et al.*, "Environment for PowerPC Microarchitectural Exploration," in *IEEE Micro*, 1999.

[25] X. Fu, J. Poe, T. Li, and J. A. B. Fortes, "Characterizing microarchitecture soft error vulnerability phase behavior," in *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 2006.

[26] E. W. Czeck and D. Siewiorek, "Effects of Transient Gate-level Faults on Program Behavior," in *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*, pp. 236–243, June 1990.

[27] 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*, pp. 416–425, Sept. 2002.

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

[29] J. Ohlsson, M. Rimin, and U. Gunneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog," in *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, June 1992.

[30] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of Fault-Tolerant Computing Systems*, pp. 84–91, June 1999.

[31] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 99–110, May 2002.

[32] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[34] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pp. 196–207, 1999.

[35] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[36] N. Wang, , and S. J. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[37] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-Based Fault Screening," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[38] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[39] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[40] B. Fields, S. Rubin, and R. Bodk, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[41] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[42] 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.

[43] H. Sharangpani and K. Arora, “Itanium processor microarchitecture,” *IEEE Micro*, vol. 20, no. 5, 2000.

[44] X. Li, Z. Li, Y. Zhou, and S. Adve, “Performance directed energy management for main memory and disks,” *ACM Trans. Storage*, vol. 1, no. 3, 2005.

[45] X. Li, R. Gupta, S. V. Adve, and Y. Zhou, “Cross-component energy management: Joint adaptation of processor and memory,” *ACM Trans. Archit. Code Optim.*, vol. 4, no. 3, 2007.

Author's Biography

Xiaodong Li was born in Henan, China. He received his Bachelor degree in Electrical Engineering from the University of Science and Technology of Beijing in May of 1997, and the Master of Science degree in Electrical Engineering from the Purdue University in 2002. He joined the computer science program at University of Illinois at Urbana-Champaign in August 2002. His main research interest is in computer systems and architecture.