SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors *

Xiaodong Li[†‡1], Sarita V. Adve [‡], Pradip Bose[†], Jude A. Rivers[†]

[†]IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
{xjli, jarivers, pbose}@us.ibm.com

[‡]Department of Computer Science
University of Illinois at Urbana-Champaign
{xli3, sadve}@uiuc.edu

Abstract

Soft errors are a growing concern for processor reliability. Recent work has motivated architecture-level studies of soft errors since the architecture can mask many raw errors and architectural solutions can exploit workload knowledge. This paper proposes a model and tool, called SoftArch, to enable analysis of soft errors at the architecture-level in modern processors. SoftArch is based on a probabilistic model of the error generation and propagation process in a processor. Compared to prior architecture-level tools, SoftArch is more comprehensive or faster. We demonstrate the use of SoftArch for an out-of-order superscalar processor running SPEC2000 benchmarks. Our results are consistent with, but more comprehensive than, prior work, and motivate selective and dynamic architecture-level soft error protection mechanisms.

1 Introduction

CMOS technology scaling has brought tremendous improvement in performance for semiconductor devices. As we move to sub-100nm lithographies however, these gains appear to face fundamental reliability related challenges. In particular, 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 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. Various studies have predicted an increase in soft error rates (SER) in different types of circuits (SRAM, latches, and logic) with scaling [3, 8, 10]. Although a consensus on exact SER values is still lacking, there is a growing concern about the phenomenon.

Until recently, the majority of the work in soft errors has focused on the device and circuit level. More recently, however, there has been work at the architecture level [1, 4, 7, 12, 13] for at least two broad reasons. First, recent work has shown that many of the raw errors that occur at the device/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 microarchitecture level [12]. The reasons for such a high masking rate include the relatively low resource utilization in a modern processor; the large number of resources that only affect performance and not correctness (e.g., branch predictor structures); and values that are used but do not affect the eventual program outcome. Second, by considering solutions at the architecture level, knowledge of workload behavior can be exploited, leading to potentially more efficient protection solutions (e.g., [12, 13]). These observations motivate the need for comprehensive models and tools for quantitative studies of soft errors at the architecture level.

This work presents an architecture-level model and tool, called SoftArch, to quantify the impact of soft errors on a modern processor (e.g., its architectural mean time to failure or MTTF). To our knowledge, this is the first such tool to model soft errors in most significant microarchitectural structures for applications with millions of instructions in reasonable time. (A detailed comparison with prior work appears in Section 5.)

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 com-

*This work was supported in part by an equipment donation from AMD and the National Science Foundation under Grant No. CCR-0313286 and EIA-0224453.

[1]Xiaodong Li is a Ph.D. student at the University of Illinois at Urbana-Champaign. This work was done while he was a co-op at the IBM T.J. Watson Research Center.

puted 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.

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.

2 The SoftArch Model

The SoftArch model consists of the following components, covered in Sections 2.1– 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.

2.1 Error Generation Model

2.1.1 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 3.2). Residence time T depends on the program and the processor architecture, and can be determined through architecture level timing simulation (Section 3.1).

2.1.2 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 [10].

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 [10] and the gate and latch counts for the logic circuit (Section 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 P(V_i) as the probability 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 P(V_3) = P($V_1 \bigcup V_2$) = P(V_1) + P(V_2) $-$ 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 P($V_1 \cdot V_2$) is simply P(V_1)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 P($V_1 \cdot V_2$) = P(V_1) = P(V_2). In general, however, the errors in two values could be partially correlated and estimating 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 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 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.

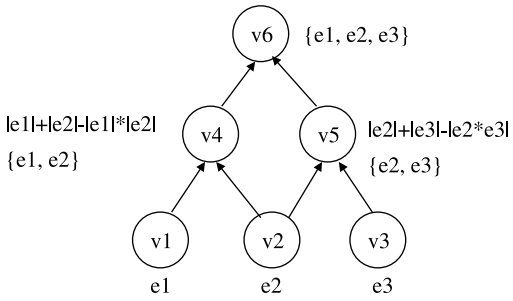

Figure 1. An example for error propagation.

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 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 $\geq$ 1, in_j's

[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.

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 2.1). For example, in Figure 1, the probability of error for $v6$ is $|e1|+|e2|+|e3|-|e1|\cdot|e2|-|e2|\cdot|e3|-|e1|\cdot|e3|+|e1|\cdot|e2|\cdot|e3|$.

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 3.5 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 $i < j$.

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 $\mathbf{MTTF'}$, 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) · (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)*

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

Table 1. Parameters for the simulated processor.

Rearranging the terms slightly,

$MTTF = \sum_{k=1}^{\infty}(1 - FailureProb')^{k-1} \cdot \sum_{i=1}^{N}\{(k-1)T_{exec} + t_{fi}\}$· *(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 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.

3.1 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 uses Turandot, a trace-driven performance simulator that models the timing of the various pipeline stages of a modern out-of-order superscalar processor in detail [6]. Table 1 summarizes the parameters for the simulated processor; these were chosen to roughly correspond to the POWER4 microarchitecture [5].

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.

3.2 Estimation of λ

Irom et al. [2] and Swift et al. [11] 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 [14]:

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

From [2], 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 [11], 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. [3] 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 paper.

According to Ziegler [14], 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. [3]. 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. [2] and Swift et al. [11] 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 Estimation of e_{logic}

At 100nm, Shivakumar et al. [10] 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.

3.4 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.

3.5 Identifying Values for Program Failure

Based on Section 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.

4 Results

We evaluate 21 SPEC CPU2000 benchmarks (9 integer and 12 floating point) with the reference input set. We use sampled traces with 100 million instructions per benchmark that were validated for acceptable representativeness against the full trace.

[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.

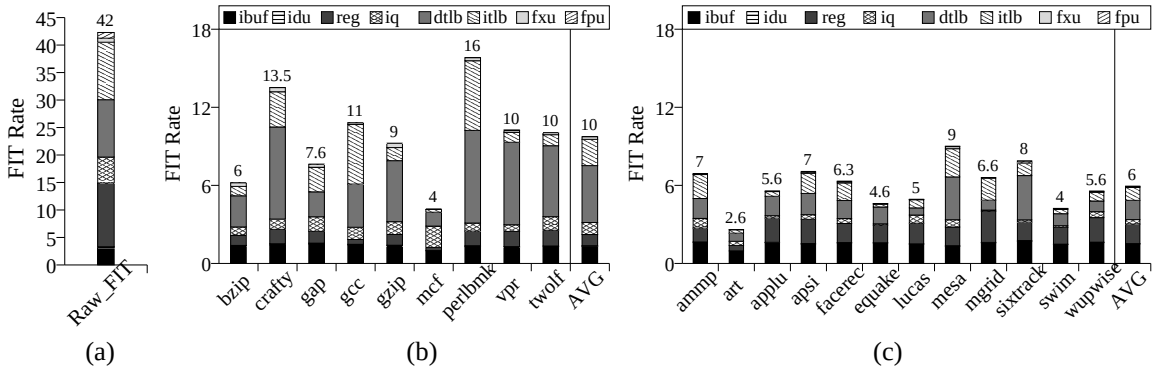

Figure 2. FIT rates (a) for raw errors, (b) with architectural masking for SPECint benchmarks, and (c) with architectural masking for SPECfp benchmarks.

4.1 Metrics

Our experiments report MTTF for an application (Section 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. This additive property is convenient when attempting to understand the relative contribution of failure rate and importance of different system components. However, while the constant failure rate assumption for raw soft errors is reasonable, it is unclear that the assumption holds after the errors are architecturally masked. Our model does not make such an assumption since it computes MTTF from first principles. Nevertheless, due to the small raw SERs, for our results, we find that the FIT rates across components are indeed additive. Therefore, for convenience and following other literature (e.g., [7]), we report our results in terms of FITs (= 1/MTTF) for the entire system and for each component.

4.2 Overall Results

Our results are presented in Figures 2 – 5. Figure 2 shows the FIT rate for an entire application. Figure 2(a) shows the raw processor FIT rate, which is calculated assuming that each raw error causes a program failure. Figures 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).

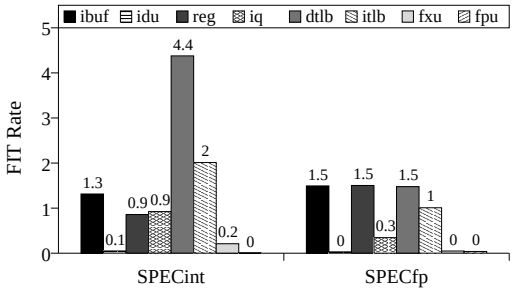

Figure 3. FIT rate for each structure, averaged across SPECint and SPECfp benchmarks.

Figure 3 summarizes the structure-wise information by showing the average FIT rate for each structure across the SPECint and SPECfp benchmarks. Figures 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). The derating factor is defined as $\frac{FIT}{rawFIT}$ (i.e., the ratios of the values of the bars in Figure 2(a) and Figure 2(b) or (c)), and is also referred to as the architectural vulnerability factor (AVF) by Mukherjee et al. [7]. Note that the lower the derating factor, the less vulnerable the structure is.

Finally, to understand dynamic application behavior, Figure 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, but they are more comprehensive since they cover more structures on chip than [7] and longer application runs than [12]):

Architectural derating. Architectural masking has a large

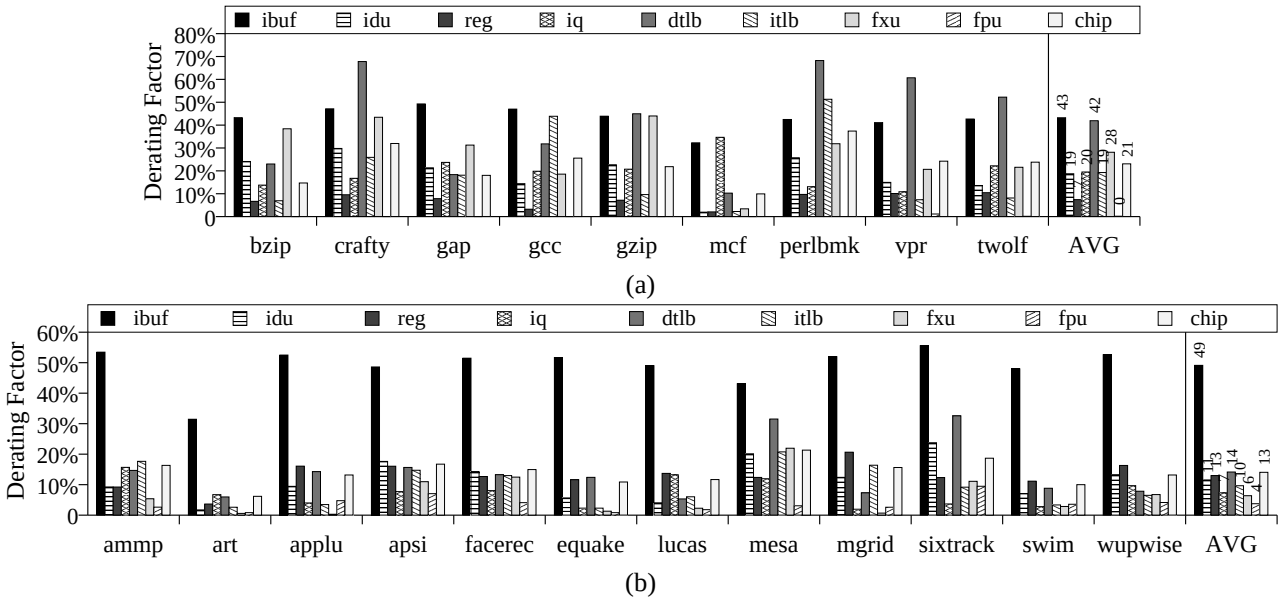

Figure 4. Architectural derating factor for each structure (a) for SPECint and (b) for SPECfp benchmarks. Note that the scales on the two graphs are different.

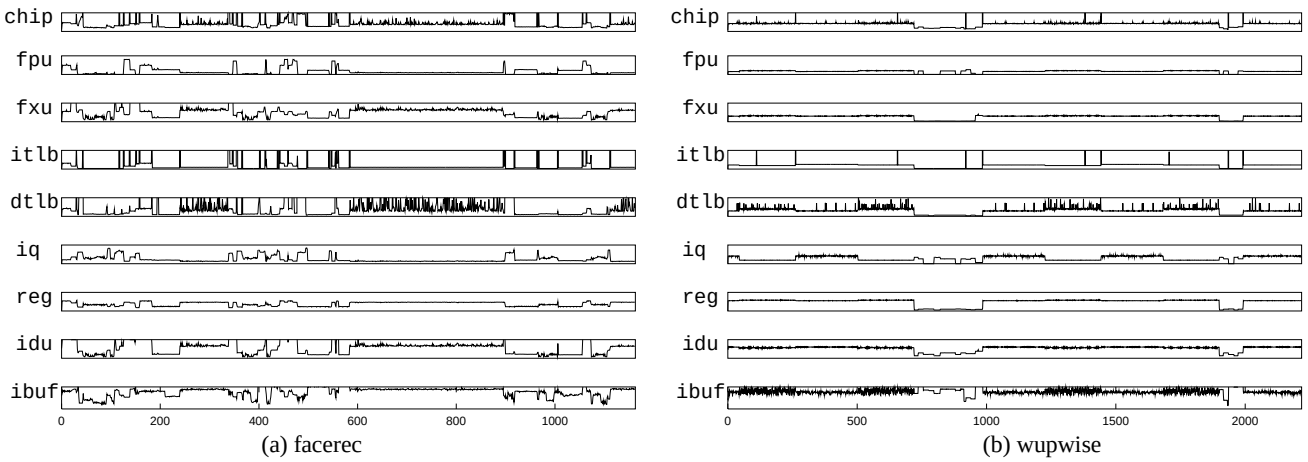

Figure 5. Intra-application variation in FIT rate for intervals of 64K instructions.

impact on the overall processor FIT rate (Figures 2 and 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 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 2 and 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. Further, Figures 2(a) and 4 show that the difference in contribution

[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.

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 5).

4.3 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 arrary structures. Some predictions expect this trend to reverse for future technologies [10], in which case the logic elements can be expected to contribute more to the overall SER.

Similar analyses 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.

4.4 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. Our model can be used to determine the best protection schemes for the anticipated workloads at design time or to adapt the protection scheme depending on the application at runtime. 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) [9]. 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.

5 Related work

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 architeture level [1, 4, 12]. Of these, the study by Wang et al. [12] is the most relevant to ours since it models a modern superscalar processor. Wang et al. perform fault injection experiments on a latch-accurate Verilog model of a modern Alpha processor (about 25,000 experiments on about 10,000 cycles of each benchmark). Key strengths of this work are that the low level Verilog model allows for high accuracy and the methodology is able to simulate the execution path after an error occurs (enabling evaluation of effects such as application level masking). The limitation, however, is the slow speed – each run is slow and many tens of thousands of runs are needed for each benchmark, limiting the simulations to about 10,000

cycles of an application's execution. SoftArch uses a higher level (hence faster, but less accurate) simulator with only one run required per benchmark, thereby enabling simulation of millions of instructions per benchmark.

The second approach, by Mukherjee et al., proposes the concept of ACE or architecturally correct execution bits, which are the bits required to be correct for correct program execution [7]. The average fraction of bits in a structure that are ACE 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. To determine the processor's architectural failure rate, they implicitly assume that a given structure's architectural failure rate is constant in time (i.e., exponential distribution for time between failures). The sum of the architectural failure rates of all structures then gives the total processor failure rate, and the reciprocal gives the processor MTTF.

The key to the above methodology is determining which bits in a structure are ACE. This is done using an *instruction-based* approach – each instruction is monitored through all stages of the pipeline, keeping track of how long it spends in each structure. Various criteria are then used to determine whether specific instruction bits are ACE (e.g., the result of a dynamically dead instruction is not ACE). They report the AVF for the instruction queue and execution units in an Itanium 2 processor.

Given the instruction-based approach used, it is unclear how to determine ACE bits for structures such as register files that store *data* values (vs. structures through which instructions flow). For example, a value deposited in the register file may stay live for a long or a short time after the instruction that computed it has retired. There is likely some form of analysis that could track the AVF contribution for such values; however, the analysis is not obvious and not provided in [7]. In contrast, the SoftArch approach is *value-based* and treats instruction and data bits with a unified mechanism, calculating MTTF from first principles. This allows the evaluation of the soft error behavior of various structures, including those carrying instructions and data.

Our experimental results and observations are qualitatively similar to those in [7, 12]. However, our methodology allows us to report results from significantly longer application runs than [12] and for more microarchitectural structures than [7] (including structures carrying data).

6 Conclusions and Future Work

This paper has 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 demonstrated the use of SoftArch by applying it to a modern out-of-order processor running SPEC2000 benchmarks. Our results, which are consistent with, but more comprehensive than, prior work show significant architecture-level derating and large variations of soft error failure rate across workloads, processor structures, and within the same workload. In the future, we plan to integrate SoftArch with circuit-level tools to improve its accuracy and to compare it with fault-injection based tools. We also plan to explore application-aware, dynamic, and selective microarchitectural soft error protection schemes.

References

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

[2] F. Irom et al. Single-Event Upset in Commercial Silicon-on-Insulator PowerPC Microprocessors. *IEEE Transactions on Nuclear Science*, 49(6):3148–3155, Dec. 2002.

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

[4] S. Kim and A. K. Somani. Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy. In *Proc. Intl. Conf. on Dependable Systems and Networks*, Sept. 2002.

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

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

[7] S. S. Mukherjee et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. 36th Intl. Symp. on Microarchitecture*, 2003.

[8] H. T. Nguyen and Y. Yagil. A Systematic Approach to SER Estimation and Solutions. In *Proc. 41st IEEE Intl. Reliability Physics Symposium*, 2003.

[9] T. Sherwood et al. Phase Tracking and Prediction. In *Proc. 30th Intl. Symp. on Computer Architecture*, 2003.

[10] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proc. Intl. Conf. Dependable Systems and Networks*, 2002.

[11] G. M. Swift et al. Single-Event Upset in the PowerPC750 Microprocessor. *IEEE Transactions on Nuclear Science*, 48(6):1822–1827, Dec. 2001.

[12] N. Wang et al. Characterizing the Effects of Transient Faults on a Modern High-Performance Processor Pipeline. In *Proc. Intl. Conf. on Dependable Systems and Networks*, 2004.

[13] C. Weaver et al. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proc. 31st Intl. Symp. on Computer Architecture*, 2004.

[14] J. F. Ziegler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, 1996.