To appear in the Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO), December 2003

A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor

Shubhendu S. Mukherjee,[1] Christopher Weaver,[1,2] Joel Emer,[1] Steven K. Reinhardt,[1,2] and Todd Austin[2]

{shubu.mukherjee, christopher.t.weaver, joel.emer}@intel.com, {stever, austin}@umich.edu

[1] VSSAD, MMDC, Intel Corporation
334 South Street. Shrewsbury, Massachusetts 01545

[2] Advanced Computer Architecture Lab
EECS Department, University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109

ABSTRACT

Single-event upsets from particle strikes have become a key challenge in microprocessor design. Techniques to deal with these transient faults exist, but come at a cost. Designers clearly require accurate estimates of processor error rates to make appropriate cost/reliability trade-offs. This paper describes a method for generating these estimates.

A key aspect of this analysis is that some single-bit faults (such as those occurring in the branch predictor) will not produce an error in a program's output. We define a structure's architectural vulnerability factor (AVF) as the probability that a fault in that particular structure will result in an error. A structure's error rate is the product of its raw error rate, as determined by process and circuit technology, and the AVF.

Unfortunately, computing AVFs of complex structures, such as the instruction queue, can be quite involved. We identify numerous cases, such as prefetches, dynamically dead code, and wrong-path instructions, in which a fault will not affect correct execution. We instrument a detailed IA64 processor simulator to map bit-level microarchitectural state to these cases, generating per-structure AVF estimates. This analysis shows AVFs of 28% and 9% for the instruction queue and execution units, respectively, averaged across dynamic sections of the entire CPU2000 benchmark suite.

1. INTRODUCTION

Moore's Law—the continuous exponential growth in transistors per chip—has brought tremendous progress in the functionality and performance of semiconductor devices, particularly microprocessors. Each succeeding technology generation has also introduced new obstacles to maintaining this growth rate. Transient faults due to single event upsets have emerged as a key challenge whose importance is likely to increase significantly in the next few design generations.

Single event upsets arise from energetic particles—such as neutron particles from cosmic rays and alpha particles from packaging material—generating electron-hole pairs as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges. A sufficient amount of accumulated charge may invert the state of a logic device—such as a latch, SRAM cell, or gate—thereby introducing a logical fault into the circuit's operation [27]. Because this type of fault does not reflect a permanent error of the device, it is termed *soft* or *transient*.

A device's error rate due to single event upsets depends on both the particle flux it encounters and its circuit characteristics. The particle flux depends on the environment. For example, at an altitude 1.5km—the altitude of Denver, Colorado—the neutron flux due to cosmic rays is 3 to 5 times higher than at sea level. Device circuit parameters that influence the error rate include the amount of charge stored, the vulnerable cross-section area, and the charge collection efficiency [22]. As feature sizes shrink, the smaller amount of charge per device makes a particle strike more likely to cause an error, but the reduced cross-section makes a strike on any given device less likely. These effects roughly cancel for latches and SRAM cells; thus, the error rate per latch or SRAM bit at a specific altitude is projected to remain roughly constant or decrease slightly for the next several technology generations ([12], [11]). However, in the absence of error correction schemes, the chip error rate will grow in direct proportion to the number of bits on the chip. Thus, while Moore's Law gives us exponential increases in transistor counts, this bounty comes at the cost of exponential increases in error rates for unprotected chips!

Soft errors due to cosmic rays are already making an impact in industry. In 2000, Sun Microsystems acknowledged cosmic ray strikes on unprotected cache memories as the cause of random crashes at major customer sites in its flagship Enterprise server line [3]. Sun is documented to having lost a major customer to IBM from this episode [3]. In 1996, Normand [17] reported numerous incidents of cosmic ray strikes by studying the error logs of several large computer systems. The fear of cosmic ray strikes prompted Fujitsu to protect 80% of its 200,000 latches in its recent SPARC processor with some form of error detection [1].

A variety of techniques exist to deal with such faults, from special radiation-hardened circuit designs (e.g., [6]) to localized error detection and correction (e.g., [1]) to architectural redundancy (e.g., [26], [23], [20], [2], [16], [18]). However, all of these approaches introduce a significant penalty in performance, power, die size, and design time. Consequently, designers must carefully weigh the benefits of adding these techniques against their cost. Although a microprocessor with inadequate protection from transient faults may prove useless due to its

unreliability, excessive protection may make the resulting product uncompetitive in cost and/or performance. Unfortunately, tools and techniques to estimate processor transient error rates are not readily available or fully understood. Furthermore, because a comprehensive error-handling strategy is best designed in from the ground up, these estimates are needed early in the design cycle.

The key to generating these error-rate estimates is understanding that not all faults in a microarchitectural structure affect the final outcome of a program. As a result, an estimate based only on raw device fault rates will be pessimistic, leading architects to over-design their processor's fault-handling features. For example, a single-bit fault in a branch predictor will not affect the sequence or results of any committed instructions. We call the probability that a fault in a processor structure will result in a visible error in the final output of a program that structure's *architectural vulnerability factor (AVF)*. Thus, the branch predictor's AVF is 0%. In contrast, a single-bit fault in the committed program counter will cause the wrong instructions to be executed, almost certainly affecting the program's result. Hence, the AVF for the committed program counter is effectively 100%. Many structures will have an AVF that is in between these two extremes. The overall error rate of a microarchitectural structure is the product of its raw fault rate and its AVF. By summing the contributions of all on-chip structures, a processor architect can map the raw fault rate (dictated by process and circuit issues) to an overall processor error rate, and thus determine whether the design meets its error rate goals (set according to the target market). Significantly, this allows an architect to examine the relative contributions of various structures and identify the most cost-effective areas to employ fault protection techniques.

This paper estimates AVFs using a novel approach that tracks the subset of processor state bits required for architecturally correct execution (*ACE*). Any fault in a storage cell that contains one of these bits, which we call *ACE bits*, will cause a visible error in the final output of a program in the absence of error correction techniques. We call the remaining processor state bits *un-ACE bits*, as their specific values are unnecessary for architecturally correct execution. A fault that affects only un-ACE bits will not cause an error. The AVF for a single-bit storage cell is simply the fraction of time that it holds ACE bits. Assuming that all cells have equal raw fault rates, the AVF for a structure is the average AVF of its storage cells, or the average fraction of its cells that hold ACE bits at any point in time.

The branch predictor's AVF is thus 0% because all predictor bits are always un-ACE bits. Similarly, all the bits in the committed PC are always ACE bits, leading to an AVF of 100%. The real power of ACE-bit analysis lies in computing AVFs for structures that hold ACE bits at some times and un-ACE bits at other times—i.e., most other processor structures. Rather than enumerate—for each structure—which bits may matter and which may not, we simply track the ACE bits through the pipeline, determine the average number of ACE bits in each particular structure, and take the ratios of these numbers to the bit capacities of the structures. This assumption relies on fault-inducing particle strikes being randomly and uniformly distributed, as is the case for cosmic rays [28].

A straightforward application of our methodology is to count the ACE bits in a structure directly using a performance model. We can also estimate the AVF of a buffering structure by counting the ACE bits that flow past a point in the pipeline and applying Little's Law.

To compute upper bounds on AVFs, we conservatively assume that every bit is an ACE bit unless we can prove otherwise. We identify un-ACE bits at both the architectural and microarchitectural levels. We identify five classes of architectural un-ACE bits. These un-ACE bits come from NOP instructions, performance-enhancing instructions (e.g., prefetches), predicated-false instructions, dynamically dead code, and logical masking. Similarly, we identify four classes of microarchitectural un-ACE bits. These are idle or invalid bits, mis-speculated bits, such as wrong-path instructions, predictor structure bits, and microarchitecturally dead bits.

Using the above methodology, dynamic slices of the SPEC 2000 benchmark suite, and a performance model, we compute the AVF for the instruction queue and execution units of an Itanium2®-like IA64 processor. We find that the AVF of the instruction queue ranges between 14% and 47%, whereas the AVFs of the execution units range between 4% and 27%. Because our methodology is conservative, these fractions are upper bounds on the AVF numbers. Further refinement of this analysis (e.g., derating the hint bits in an IA64 load instruction) could further lower the AVF estimates. However, we believe we have captured most of the dominant AVF effects for the IA-64 architecture and Itanium2®-like microarchitecture we examined and, hence, we expect the contribution from further refinement to be small.

The rest of the paper is organized as follows. Section 2 provides background on reliability metrics used in the industry to express soft error rates. Section 3 describes the components we must consider to compute the AVF. Section 4 shows how to compute AVF using the classification of Section 3. Section 5 describes our methodology and Section 6 presents our results. Section 7 describes related work and Section 8 presents our conclusions.

2. SOFT ERROR BACKGROUND AND TERMINOLOGY

Section 2.1 describes the error metrics MTBF and FIT. We discuss vulnerability factors and their impact on error detection and correction requirements in Section 2.2.

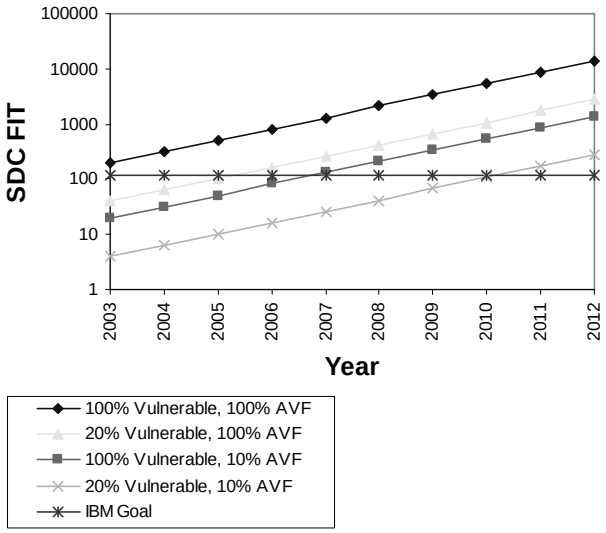

Figure 1. Impact of AVF on SDC FIT of future microprocessors. For 2003 we assumed 200,000 bits may be vulnerable to cosmic ray strikes [1]. This figure assumes a FIT/bit of 0.001. The number of vulnerable bits grows with Moore's Law.

2.1 MTBF and FIT

Vendors express an error budget at a reference altitude in terms of Mean Time Between Failures (MTBF). Errors are often further classified as undetected or detected. The former are typically referred to as *silent data corruption* (SDC); we call the latter *detected unrecoverable errors* (DUE). Note that detected recoverable errors are not errors.

For example, for its Power4 processor-based systems, IBM targets 1000 years system MTBF for SDC errors, 25 years system MTBF for DUE errors that result in a system crash, and 10 years system MTBF for DUE errors that result in an application crash [4]. Note that the processor MTBF must be significantly higher than the system MTBF, particularly for large multiprocessor systems.

In this paper we focus on SDC errors. Adding error detection (but not correction) to a structure eliminates SDC errors, converting those faults to DUE errors.

Another commonly used unit for error rates is FIT (Error in Time), which is inversely related to MTBF. One FIT specifies one failure in a billion hours. Thus, 1000 years MTBF equals 114 FIT (10^9 / (24×365×1000)). A zero error rate corresponds to zero FIT and infinite MTBF. Designers usually work with FIT because FIT is additive, unlike MTBF.

To evaluate whether a chip meets its soft error budget—possibly via the use of error protection and mitigation techniques—microprocessor designers use sophisticated computer models to compute the FIT rate for every device—RAM cells, latches, and logic gates—on the chip. The effective FIT rate for a structure is the product of its raw circuit FIT rate and the structure's *vulnerability factor*, i.e., an estimate of the probability that a circuit fault will result in an observable error (see the following section). The overall FIT rate of the chip is calculated by summing the effective FIT rates of all the structures on the chip.

Current predictions show that typical raw FIT rate numbers for latches and SRAM cells vary between 0.001 – 0.01 FIT/bit at sea level ([24],[17],[12],[11]). The FIT/bit is projected to remain in this range for the next several technology generations, unless microprocessors aggressively lower the supply voltage to reduce the overall power dissipation of chip. The total FIT contribution of logic gates today is a negligible fraction of the FIT contribution from latches [22], so we concern ourselves only with faults due to strikes on latches and SRAM cells. In the future, if the contribution of logic becomes non-negligible, we could incorporate the effective FIT rate due to a logic block into the FIT rate of the latch that it feeds.

2.2 Vulnerability Factors

The effective FIT rate per bit is influenced by several *vulnerability factors* (also known as *derating factors or soft error sensitivity factors*). In general, a vulnerability factor indicates the probability that an internal fault in a device's operation will result in an externally visible error.

For example, when a level-sensitive latch is accepting data rather than holding data, a strike on its stored bit may not result in an error, because the erroneous stored value will be overridden by the (correct) input value. If the latch is accepting data 50% of the time, this effect results in a *timing vulnerability factor* for the latch of 50%. For simplicity, we assume this timing vulnerability factor is already incorporated in the raw device fault rate. The computation of the device fault rate also includes some circuit-level vulnerability factors which are beyond the scope of this paper.

The *architectural vulnerability factor* (AVF) expresses the probability that a visible system error will occur given a bit flip in a storage cell. The AVF can have a significant impact on the effective error rate of a processor. Prior studies with statistical fault injection into RTL models have demonstrated AVFs of 1%-10% for latches [25] and 0% - 100% across a range of architectural and microarchitectural state bits [13].

Figure 1 illustrates the impact of AVF on the extent of error detection needed in a microprocessor. Thus, to meet IBM's SDC target of 114 FIT in 2005, with 100% AVF we must protect 80% of the bits (and, hence have 20% vulnerable bits). However, with an AVF of 10%, we do not have to protect any bits to meet the target 114 FIT. Similarly, in 2010 we can meet the target SDC FIT with 80% protection and 10% AVF, but not with 100% AVF.

3. IDENTIFYING UN-ACE BITS

The key to computing AVFs is to determine which bits affect the final system output (i.e., which are the ACE bits) and which do not (i.e., the un-ACE bits). Interestingly, this does not necessarily correspond to meeting the precise semantics of the architecture. Given a specific execution

of program, we only care that the final system output as observed by a user is correct. Incorrect results not observed by any user (e.g., dynamically dead instruction results) are irrelevant, and bits encountering faults leading to such incorrect results are un-ACE bits.

The definition of a program's output depends on the user's interaction with the program. Normally a program's outputs are just the values sent by the program via I/O operations. However, if a program is run under a debugger, then program variables examined via the debugger become outputs, and influence the determination of which bits are ACE bits.

Similarly, in a multiprocessor system, multiple executions of the same parallel program may yield different outcomes due to race conditions. Whether a bit is ACE or un-ACE may depend on the outcome of a race; our analysis would use the outcome of the race in the particular execution under study. (We analyze only a uniprocessor system in this paper.)

For our purposes, we require a precise definition of what constitutes an output. Conceptually, we take the broadest view: an output is a program's generated value that is sent to an I/O device. In practice, we do not typically track values this far. However, we do track values well beyond the point that they are committed to architectural registers or stored to memory to determine whether they could potentially influence the output.

Given this definition of outputs, we would like to determine which bits are ACE and which are un-ACE. Because we desire a conservative (upper-bound) AVF estimate, we first assume that all bits are ACE bits unless we can show otherwise. We then identify as many sources of un-ACE bits as we can. We do not need (nor claim to) have a complete categorization of un-ACE bits; however, the more comprehensive our analysis is, the tighter our bound will be. Nevertheless, we believe that the sources below capture the dominant contributors of un-ACE bits.

For discussion purposes, we divide the sources of un-ACE bits into two general categories: microarchitectural un-ACE bits (Section 3.1) and architectural un-ACE bits (Section 3.2). Section 4 will show how to use this classification to compute the AVF of hardware structures.

3.1 Microarchitectural Un-ACE Bits

We call processor state bits that cannot influence the committed instruction path *microarchitectural un-ACE bits*. Microarchitectural un-ACE bits can arise from the following four situations:

- *Idle or Invalid State*. There are numerous instances in a microarchitecture when a data or status bit is idle or does not contain any valid information. Such data and status bits are un-ACE bits. Control bits are always assumed to be ACE bits because a strike on a control bit may cause idle state to be treated as non-idle state.
- *Mis-speculated State*. Modern microprocessors often perform speculative operations that may later be found to be incorrect. Examples of such operations include branch prediction or speculative memory disambiguation. The bits that represent incorrectly speculated operations are un-ACE bits.
- *Predictor Structures.* Modern microprocessors have many predictor structures, such as branch predictors, jump predictors, return stack predictors, and store-load dependence predictors. A fault in such a structure may result in a misprediction, and will affect performance, but will not affect correct execution. Consequently, all such predictor structures contain only un-ACE bits.
- *Ex-ACE State.* ACE bits become un-ACE bits after their last use. In other words, the bits are dead. This category encompasses both architecturally dead values, such as those in registers, as well as architecturally invisible state. For example, after a dynamic instance of an instruction is issued for the last time from an instruction queue, it may still persist in a valid state in the instruction queue, waiting until the processor knows that no further re-issue will be needed, but a fault in that instruction will not have any effect on the output of a program.

3.2 Architectural Un-ACE Bits

Architectural un-ACE bits are those that affect correct-path instruction execution, but only in ways that do not change the output of the system. For example, a strike on a storage cell carrying the operand specifier of a NOP instruction will not affect a program's computation. We call the bits of an instruction that are not necessary for an ACE path *un-ACE instruction bits*. Below we identify five sources of architectural un-ACE bits:

- *NOP instructions.* Most instruction sets have NOP instructions that do not affect the architectural state of the processor. Fahs, et al. [9] found 10% NOPs in the dynamic instruction stream of SPEC2000 integer benchmark suite using the Alpha instruction set. On the Intel® Itanium® processor, Choi, et al. [7] observed 27% retired NOPs in SPEC2000 integer benchmarks. These instructions are introduced for a variety of reasons, such as to align instructions to address boundaries or to fill VLIW-style instruction templates. Clearly, the only ACE bits in a NOP instruction are those that distinguish it from a non-NOP. Depending on the instruction set, this may be the opcode or the destination register specifier. The remaining bits are un-ACE bits.
- *Performance-enhancing instructions.* Most modern instruction sets include performance-enhancing instructions. For example, a non-binding prefetch instruction brings data into the cache to reduce the latency of later loads or stores. A single-bit upset in a non-opcode field of such a prefetch instruction will not affect the correct execution of a program. A fault may cause the wrong data to get prefetched, or may cause the address to become invalid, in which case the prefetch will be ignored, but the program semantics will not change. Thus, the non-opcode bits are un-ACE bits. Fahs, et al. [9] reported that 0.3% of the dynamic instructions in

SPEC2000 integer suite using the Alpha instruction set were prefetch instructions. The Itanium2® architecture has other performance-enhancing instructions, such as the branch predict hint instruction; none of these were present in the binaries we used for our evaluation.

- *Predicated-false instructions.* Predicated instruction-set architectures, such as IA64, allow instruction execution to be qualified based on a predicate register. If the predicate register is true, the instruction will be committed. If the predicated register is false, the instruction's result will be discarded. Clearly, all bits except the predicate register specifier bits in a predicated-false instruction are un-ACE bits. A corruption of the predicate register specifier bits may erroneously cause the instruction to be predicated true. Hence, we conservatively call those ACE instruction bits. However, if the instruction itself is dynamically dead (see below) and the predicate register is overwritten before any other intervening use, then the predicate register as well as the corresponding specifier can be considered un-ACE bits. In our evaluation, we found about 7% of dynamic instructions were predicated false.
- *Dynamically dead instructions*. Dynamically dead instructions are those whose results are not used. Instructions whose results are simply not read by any other instructions are termed first-level dynamically dead (FDD). Transitively dynamically dead (TDD) instructions are those whose results are used only by FDD instructions or other TDD instructions. We consider an instruction with multiple destination registers dynamically dead only if all its destination registers are unused.

  We track FDD and TDD instructions through both registers and memory. For example, if two instructions A and B successively write the same register R1 without any intervening read of register R1, then A is an FDD instruction tracked via register R1. Similarly, if two store instructions C and D write the same memory address M without any intervening load to M, then C is an FDD instruction tracked via memory address M.

  Using the Alpha instruction set running the SPEC2000 integer benchmarks, Butts and Sohi [5] reported about 9% FDD and 3% TDD instructions tracked only via registers. In contrast, Fahs, et al. [9] found about 14% FDD and TDD instructions—tracked via both registers and memory—in their evaluation of SPEC2000 integer benchmarks running on an Alpha instruction set architecture. Our evaluation with IA64 across portions of 18 SPEC2000 benchmarks shows about 12% FDD and 8% TDD instructions tracked via both registers and memory. Our analysis assumes that memory results produced by FDD and TDD instructions are not used by other I/O devices. We suspect that our numbers for dynamically dead instructions are higher than earlier evaluations because of aggressive compiler optimizations, which has shown to increase the fraction of dead instructions [5].

  In this paper, we count all the opcode and destination register specifier bits of FDD and TDD instructions as ACE bits; all other instruction bits are clearly un-ACE bits. If the opcode bits get corrupted, then the machine may crash when evaluating those bits. If the destination register specifier bits get corrupted, then an FDD or TDD instruction may corrupt a non-dead architectural register, which could affect the final outcome of the program. This accounting is conservative, as it is likely that some fraction of bit upsets in the opcode or destination register specifier would not lead to incorrect program output.
- *Logical masking.* There are many bits that belong to operands in a chain of computation whose values still do not influence the computation's results. We say these bits are *logically masked*. For example, consider the following code sequence:

  (1) R2 ß R3 OR 0x00FF
  (2) R4 ß R2 OR 0xFF00
  (3) R3 ß 0
  (4) R2 ß 0
  (5) output R4

  In this case, the lower 16 bits of R4 will be 0xFFFF regardless of the values of R2 and R3. When the value of a bit in an operand does not influence the result of the operation, we call this *logical masking*. In our example, bits 0 to 7 (the low order bits) of R3 are logically masked in instruction 1, and bits 8 to 15 of R2 are masked in instruction 2. For a bit in a register to be logically masked (and thus un-ACE), it must be logically masked for all of its uses. We could identify additional un-ACE bits by considering *transitive logical masking*, where the effects of logical masking are propagated backwards transitively. In the above code sequence, bits 8 to 15 of R3 contribute only to bits 8 to 15 of R2, assuming no other uses of R3. Because bits 8 to 15 of R2 are logically masked, via transitive logical masking, bits 8 to 15 of R3 are masked as well. We do not evaluate transitive logical masking in this paper.

  We have found that logical masking arises from compare instructions prior to a branches (where it matters only if the value is zero or non-zero), bitwise logical operations, and 32-bit operations in a 64-bit architecture (where we assume that the upper 32 bits are un-ACE, which may not be true for certain ISAs, such as the Alpha ISA). For our purpose, all logically masked bits are un-ACE bits and can be factored out of the AVF calculation.

Further analysis of a program—not considered in this paper—can potentially reveal other opportunities for derating. For example, hint bits in IA64's load instructions can be considered un-ACE. Similarly, it may be possible to identify further logical masking by analyzing values used by integer adds or floating point operations, but we do not consider them in this paper.

4. COMPUTING AVF

This section describes how to compute AVF based on the sources of un-ACE bits from Section 3. Section 4.1 outlines the equations we use to compute AVFs for storage cells. Section 4.2 shows how to compute AVFs using

Little's Law [15]. Finally, Section 4.3 shows how to compute the AVF using a performance model.

4.1 AVF Equations for a Hardware Structure

The AVF of a storage cell is the fraction of time an upset in that cell will cause a visible error in the final output of a program. Thus, the AVF for an unprotected storage cell is the percentage of time the cell contains an ACE bit. For example, if a storage cell contains ACE bits for a million cycles out of an execution of ten million cycles, then the AVF for that cell is 10%.

Although we defined the AVF equations with respect to a storage cell, in this paper we will compute the AVF for a whole hardware structure. The AVF for a hardware structure is simply the average AVF for all its bits in that structure, assuming that all bits in that structure have the same circuit composition and, hence, the same raw FIT rate. Then, the AVF of a hardware structure is equal to:

$$\frac{\text{average number of ACE bits resident in a hardware structure in a cycle}}{\text{total number of bits in the hardware structure}}$$

The above equation can be rewritten as:

$$\frac{\sum \text{residency (in cycles) of all ACE bits in a structure}}{\text{total number of bits in the hardware structure} \times \text{total execution cycles}}$$

The latter equation makes it easier to compute the AVF using a simulator.

4.2 Computing AVFs using Little's Law

Using Little's Law [15], we can compute the average number of ACE bits resident in a structure and, therefore, the AVF of the structure. Little's Law can be translated into the equation N = B×L, where N = average number of bits in a box, B = average bandwidth per cycle into the box, and L = average latency of an individual bit through the box. Applying this to ACE bits, we get the average number of ACE bits in a box as the product of the average bandwidth of ACE bits into the box (B_{ace}) times the average residence cycles of an ACE bit in the box (L_{ace}). Thus, we can express the AVF of a structure as:

$$\frac{B_{ace} \times L_{ace}}{\text{total number of bits in the hardware structure}}$$

In many cases, it is possible to compute the bandwidth of ACE bits into a structure and the average residence cycles of ACE instructions using hardware performance counters, allowing AVF estimation without a simulation model. In Section 6.2, we will show how to approximate the AVF of an instruction queue using the formulation in this section.

Table 1. SPEC2000 Benchmarks used in this paper. M = 1 Million.

Integer Benchmarks	Instructions Skipped	Floating Point Benchmarks	Instructions Skipped
bzip2-source	48,900 M	ammp	50,900 M
cc-200	16,600 M	applu	500 M
crafty	120,600 M	apsi	100 M
eon-kajiya	73,000 M	art-110	36,400 M
gap	18,800 M	equake	1,500 M
gzip-graphic	2,9000 M	facerec	64,100 M
mcf	26,200 M	fma3d	23,600 M
parser	71,400 M	galgel	5,000 M
perlbmk-makerand	0 M	lucas	123,500 M
twolf	185,400 M	mesa	73,300 M
vortex_lendian3	59,300 M	mgrid	200 M
vpr-route	49,200 M	sixtrack	4,100 M
		swim	78,100 M
		wupwise	23,800 M

4.3 Computing AVFs using a Performance Model

In this paper, we compute the AVFs of two structures—the instruction queue and execution units—using the Asim performance model framework (Section 5). To compute the AVF of a structure using the equation in Section 4.1, we need the following information:

- sum of all residence cycles of all ACE bits of the objects resident in the structure during the execution of the program,
- total execution cycles for which we observe the ACE bits' residence time, and
- total number of bits in a hardware structure.

Using a performance model, we can compute all of the above. Specifically, in this paper, we examine objects that carry instruction information along the pipeline.

Our AVF algorithm is divided up into three parts. As an instruction flows through different structures in the pipeline, we record the residence time of the instruction in the structure. Then, before the instruction disappears from the machine—either via a commit or via a squash—we update the structures it flowed through with a variety of information, such as the residence cycles, whether the instruction committed, etc. (part 1). Also, if the instruction commits, we put the instruction in a post-commit analysis window to determine if the instruction is dynamically dead or if there are any bits that are logically masked (part 2). Finally, at the end of the simulation, using the information captured in parts 1 and 2, we can easily compute the AVF of a structure (part 3).

To compute whether an instruction is a first-level dynamically dead (FDD) or a transitively dynamically dead (TDD) instruction and whether any of the result bits have logical masking (Section 3.2), we must know about the future use of an instruction's result. We use the analysis window to capture this future use. When an instruction commits, we enter it into the analysis window, linking it with the instructions that produced its operands. At any time, we can analyze the future use of an instruction's results by examining its successors in the analysis window.

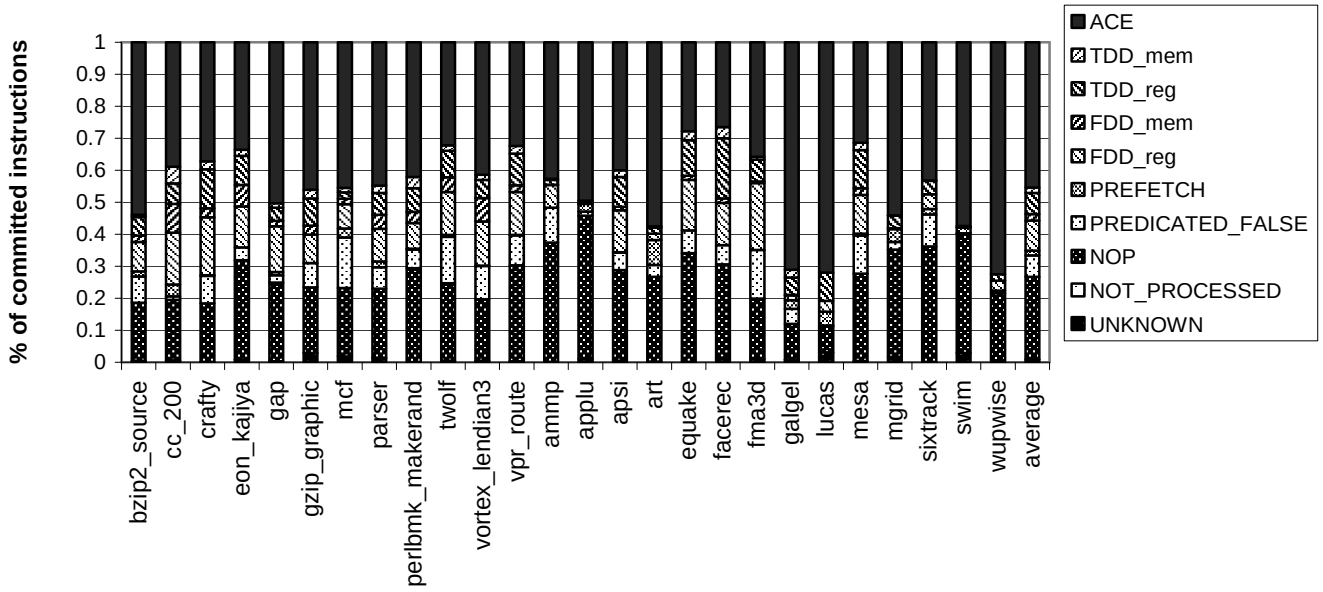

Figure 2.Program-level Decomposition of Committed Instructions. TDD = Transitively Dynamically Dead, FDD = First-level Dynamically Dead, NOT_PROCESSED = instructions not processed at the end of the simulation, UNKNOWN = live or deadness could not be determined during program execution with our analysis window.

Of course, because the analysis window must be finite in size, we cannot always determine the future use precisely. We found that an analysis window of 40,000 instructions covers most of the needed future use information.

The analysis window has three sub-windows which compute FDD, TDD, and logical masking information, respectively. Each sub-window has two primary data structures: a linked list of instructions in commit order and a table indexed via architectural register number or memory address. The linked list maintains the relative age information necessary to compute future use. Each entry in the table maintains the list of producers and consumers for that register or memory location. The FDD, TDD, and logical masking information can all be computed using this list of producers and consumers. Thus, a list with two consecutive producers for a register R and no intervening consumer for the same register R can be used to mark the first producer of R as a dynamically dead instruction.

We used microbenchmarks written in assembly language to ensure that the analysis window was working correctly. We explicitly engineered a certain number dynamically dead instructions and values in these microbenchmarks. The number of dynamically dead instructions reported by the analysis window matched up with the expected number of dynamically dead instructions coded into the benchmarks.

5. METHODOLOGY FOR EVALUATION

For our evaluation, we use an Itanium2®-like IA64 processor [14] scaled to current technology. This processor was modeled in detail in the Asim [8] performance model framework. Red Hat Linux 7.2 was modeled in detail via an OS simulation front-end. For wrong paths, we fetch the mis-speculated instructions, but do not have the correct memory addresses that a load or store may access. We augmented this processor model with the instrumentation described in Section 4.3.

Table 1 lists the skip interval and input set selected for each of the Spec2000 programs used for our analysis. The benchmarks were compiled with the Intel® electron compiler (version 7.0) with the highest level of optimization. We obtained the number of instructions to skip using Sherwood et al.'s [21] SimPoint analysis modified for the IA64 instruction set architecture. For each benchmark, we obtained a number of simpoints, but here we present numbers only for the first simpoint of each benchmark. We ran each simpoint for 100 million instructions (including NOPs).

6. RESULTS

This section describes our AVF results. Section 6.1 provides a program-level decomposition of various components in the committed stream of instructions. In Sections 6.2 and 6.3, we use these program-level components as well as microarchitectural states to compute the AVFs of the instruction queue and execution units, respectively.

6.1 Program-level Decomposition

Figure 2 shows a decomposition of the dynamic stream of instructions based on whether the instruction's output affects the final output of the benchmark. An instruction whose result may affect the output is an ACE instruction, while an instruction which definitely does not affect the final output is un-ACE. As the figure shows, on average,

Table 2. AVF breakdown using Little's Law. # ACE inst = ACE IPC X ACE Latency. AVF ~= # ACE inst / # instruction queue entries.

Integer Benchmarks	ACE IPC	ACE Latency (cycles)	# ACE Inst	AVF	Floating Point Benchmarks	ACE IPC	ACE Latency (cycles)	#ACE Inst	AVF
bzip2-source	0.55	22	12	19%	ammp	0.23	92	21	33%
cc-200	0.57	18	10	16%	applu	0.82	21	18	27%
crafty	0.37	15	6	9%	apsi	0.31	31	9	15%
eon-kajiya	0.36	20	7	11%	art-110	0.68	37	25	40%
gap	0.78	17	13	21%	equake	0.26	12	3	5%
gzip-graphic	0.60	13	8	12%	facerec	0.41	7	3	5%
mcf	0.25	68	17	26%	fma3d	0.59	11	7	10%
parser	0.49	24	12	19%	galgel	1.10	21	23	35%
perlbmk-makerand	0.38	17	7	10%	lucas	1.23	17	21	33%
twolf	0.30	27	8	13%	mesa	0.47	16	8	12%
vortex_lendian3	0.42	22	9	15%	mgrid	1.28	10	13	21%
vpr-route	0.35	12	4	7%	sixtrack	0.66	20	13	21%
					swim	1.08	16	17	27%
					wupwise	1.60	13	20	31%
average	0.45	23	9	15%	average	0.77	23	14	23%

we get about 45% ACE instructions. The rest—55% of the instructions—are un-ACE instructions. Some of these un-ACE instructions still contain ACE bits, such as the opcode bits of prefetch instructions. Because our analysis is conservative, there may be other opportunities to move instructions from the ACE to un-ACE category.

In Figure 2, UNKNOWN denotes the instructions for which we could not determine if the destination register was live or dead because there was not enough information in the analysis window to make this determination (Section 0). NOT_PROCESSED is always 20,000 instructions at the end of the simulation. UNKNOWN and NOT_PROCESSED instructions account for about 1% of the total instructions, so they should not affect our analysis significantly.

NOPs, predicated false instructions, and prefetch instructions account for 26%, 6.7%, and 1.5%, respectively. NOPs are introduced in the IA64 instruction stream to align instructions on three-instruction bundle boundaries. These NOPs are carried through the Itanium2® pipeline.

Finally, FDD and TDD show the first-level and transitively dynamic dead instructions. FDD_reg and FDD_mem denote results that are written back to registers and memory, respectively. On average, FDD_reg and FDD_mem account for about 9.4% and 2% of the dynamic instructions. IA64 has a large number of registers relative to other instruction sets. Consequently, it produces a significantly higher fraction of FDD_reg instructions compared to FDD_mem. Similarly, the TDD_reg and TDD_mem account for 6.6% and 1.6% of the dynamic instructions.

6.2 AVF for Instruction Queue

Figure 3 shows what percentage of cycles a storage cell in the instruction queue contains ACE and un-ACE bits. This calculation assumes each entry of the instruction queue is approximately 100 bits. An IA64 instruction is 41 bits, but the number of bits required in the entry is higher because a large number of bits are required to capture the in-flight state of an instruction in the machine. Of these 100 bits, we estimate that five bits are control bits and cannot be derated. Of the remaining 95 bits, we do not derate the seven opcode bits for any instruction. Additionally, we do not derate the six predicate specifier bits of falsely predicated instructions or the seven destination register specifier bits for FDD and TDD instructions.

Figure 3 shows that on average, a storage cell in the instruction queue contains an ACE bit about 28% of the time. Thus, the AVF of the instruction queue is 28%. On average a cell is idle 30% of the cycles and contains an un-ACE bit about 42% of the cycles. Across the simulated portions of our benchmark suite, the AVF number ranges between 14% and 47% for the instruction queue.

Figure 33 also shows that the floating point programs, in general, have higher AVFs compared to integer programs (31% vs. 25%, respectively). Floating-point programs usually have many long-latency instructions and few branch mispredictions. Hence, they use the instruction queue more effectively than integer programs, leading to a higher AVF.

Table 2 shows how to approximate AVFs (at the instruction level) for the instruction queue using Little's Law. As explained in Section 4.2, the AVF of the instruction queue can be approximated as the ratio of the average number of ACE instructions in the instruction queue to the total number of instruction entries in the instruction queue, which is 64 in our machine. The number of ACE instructions in the instruction queue, as given by Little's Law, is the product of the bandwidth or ACE IPC and the average number of cycles an instruction in the instruction queue can be considered to be in ACE state or ACE latency. Note that an instruction can persist even after it is issued for the last time. Thus, after an ACE instruction is issued for the last time, the ACE bits holding the ACE instruction become un-ACE. We obtained the ACE IPC and ACE latency from our performance model.

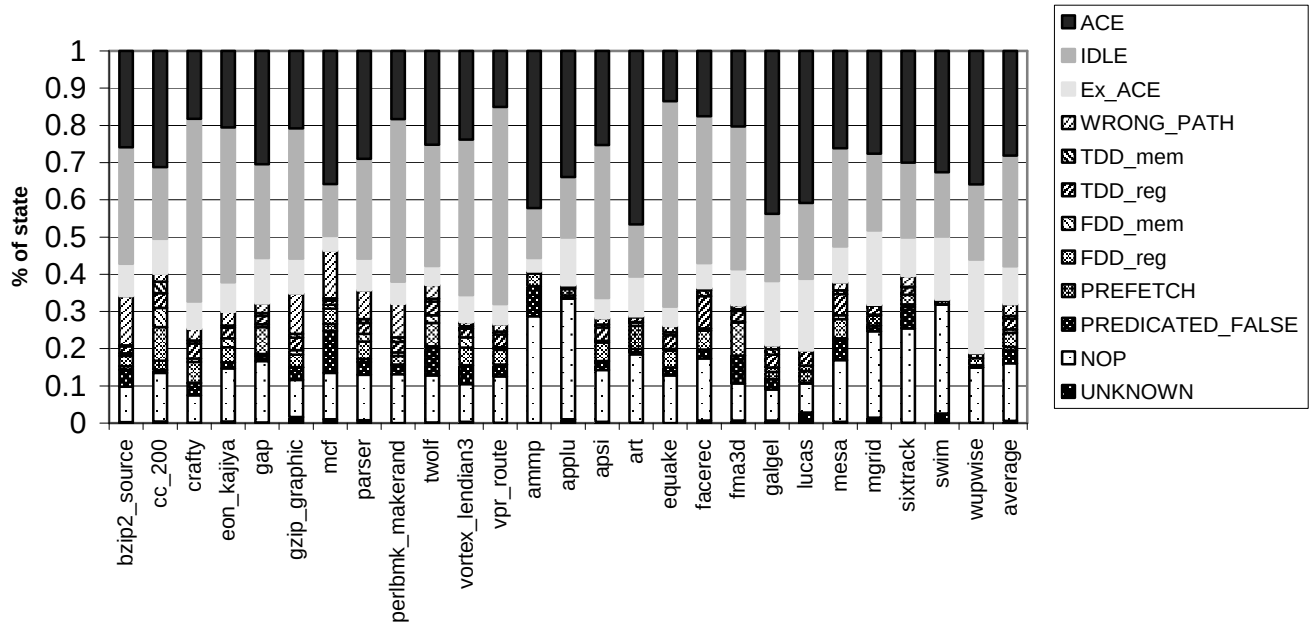

Figure 3. Breakdown of different architectural and microarchitectural states for the instruction queue. AVF is the % of cycles the instruction queue contains ACE bits.

Using the ACE IPC and ACE latency, we compute the AVFs in Table 2. Using this method, we compute an average AVF of 19%, which is 9% lower than that of actual AVF for the instruction queue reported earlier. This difference can be attributed to the ACE bits of un-ACE instructions, such as prefetch and dynamically dead instructions, whose results do not affect the final output of a program (Section 3). Table 2 does not account for these ACE bits, but Figure 3's ACE number includes them. If we did our Little's Law analysis at the bit-level, instead of instruction-level as in Table 2, then we would have matched Figure 3's average AVF of 28%.

Table 2 also explains why lucas has an AVF similar to ammp even though lucas has one of the highest ACE IPCs. This is because the AVF depends on both the ACE IPC as well as ACE latency. Although lucas has a high ACE IPC, it has relatively low ACE latency. Consequently, the product of these two terms results in an AVF similar to ammp's, which has a low ACE IPC but a high ACE latency.

6.3 AVFs for the Execution Units

This section describes the AVF numbers of the execution units in our simulated machine model. In our six-issue machine, we have four integer pipes and two floating point pipes. Integer multiplication is, however, processed in the floating point pipeline. When integer programs execute the floating point pipes lie idle.

We assume that the execution units overall have about 50% control latches and 50% datapath latches. First, we show how to derate the entire execution unit, so that the results would apply to both the control and datapath latches. Then, we will show how to further derate the datapath latches only.

Figure 4 shows that the execution units on average spend 11% of the cycles processing ACE instructions (with a range of 4% to 27%). Thus, the average AVF of a latch in the execution units is 11%. Interestingly, the execution units' AVF is significantly lower than that of the instruction queue. This is due to three effects. First, instructions must wait in the instruction queue until they execute and retire. Thus, ACE instructions persist longer in the instruction queue than they take to execute in the execution units.

Second, speculatively issued instructions succeeding cache-miss loads must replay through the instruction queue. However, only the last pass through the execution units matters for correct execution. The execution unit state for all prior executions is counted as un-ACE. Note that this is possible in our processor model because a corrupted bit in one of the instructions designated for a replay does not affect the decision to replay. The information necessary to make this decision resides elsewhere in the instruction queue.

Third, the floating point pipes are mostly idle while executing integer code, greatly reducing their AVFs.

As mentioned earlier, the above analysis computes a single AVF for both the control and datapath latches in the execution units. However, the datapath latches themselves can be further derated based on whether specific datapath bits are logically masked (Section 3.2) or are simply idle (Section 3.1). We apply logical masking to data values (and, hence, datapaths) only; analyzing logical masking for control latches would be a complex task.

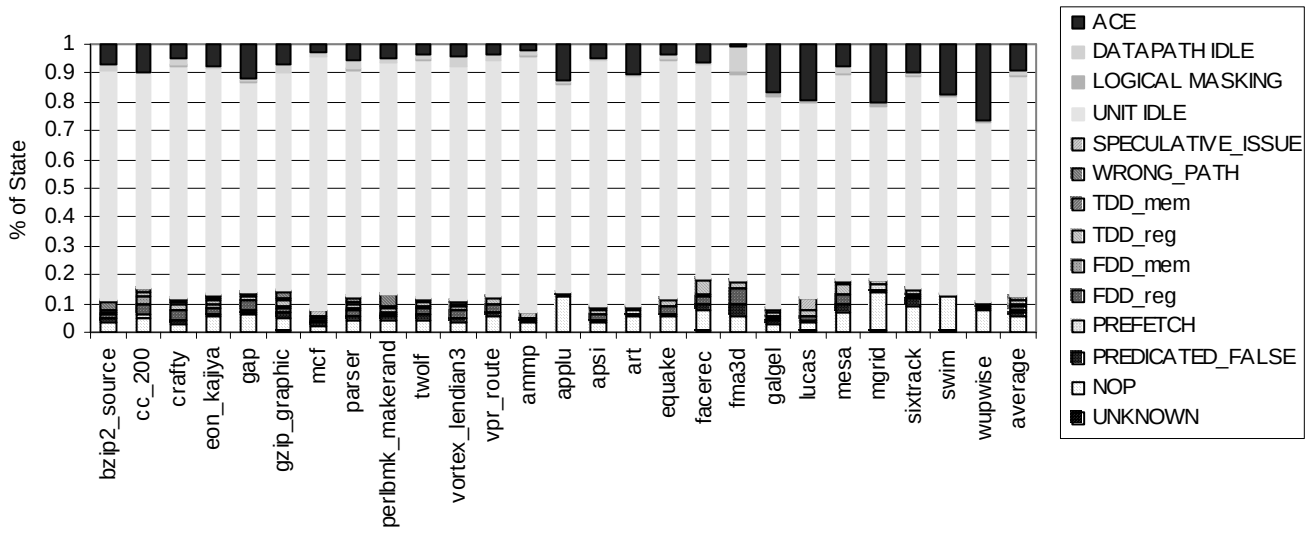

Figure 4. Breakdown of different architectural and microarchitectural states for the execution units. ACE above denotes the % of cycles the execution units contain ACE instructions.

We implemented logical masking functions for a small but important subset of the roughly 2000 static internal instruction types that we have in our processor model. This subset contains a variety of functions, including logical OR, AND, etc. We also estimate that another 20% of the instructions (including loads, stores, and branches) will not have any direct logical masking effect. The combination of these two categories covers the vast majority of dynamically executed instructions. Figure 4 shows that this logical masking analysis further reduces the AVF by 0.5% (averaged across both control and data latches, even though this does not apply to control latches). We expect that the incremental decrease in AVF due to the remaining unanalyzed instruction types will be small. We did not consider transitive logical masking (see Section 3.2), which would further reduce the AVF number for the datapath latches.

Datapath latches can be further derated by identifying the fraction of time they remain idle. For example, an IA64 compare instruction produces two predicate values—a predicate value and its complement. However, in our simulated implementation, these two result bits are sent over a 64-bit result bus, leaving 62 of the datapath lanes idle. This effect further reduces the AVF by 1.5%, as shown by DATAPATH_IDLE in Figure 4. Depending on the implementation, however, the DATATPATH_IDLE portion can also be viewed as bits that get logically masked at the implementation level. In contrast, UNIT_IDLE in Figure 4 refers to the whole execution unit being idle because of the lack of any instruction issued to that unit.

Overall, factoring in logical masking and idle latches in datapaths, the average AVF for the execution units is reduced to 9%. Across the simulated portions of our benchmark suite, the AVF for the execution units ranges from 4% to 27%.

7. RELATED WORK

Prior work in estimating AVFs has used statistical fault injection into hardware RTL models. Kim and Somani [13] did a systematic fault injection study of Sun Microsystem's publicly disclosed picoJava II RTL model and reported wide variation of AVFs for a variety of picoJava II's hardware structures. Wang and Patel [25] injected faults into an RTL model of the Alpha 21164 processor and reported AVFs of less than 10% for the pipeline latches. The biggest advantage of using an RTL model is that usually the RTL model has all the hardware structures necessary to create a processor. In contrast, a performance model, which we use in our paper, has only components that affect the performance of a processor. Consequently, we can only report the AVFs of components that are modeled. Nevertheless, the performance models used in industry are extremely detailed and capture significant portions of the processor under design.

We improve upon statistical fault injection into RTL models in four ways. First, statistical fault injection requires simulating a large number of fault cases to provide adequate statistical significance. Using ACE analysis, our technique provides reasonably tight, deterministic AVF estimates in a single experiment. As a result, designers can not only generate AVF estimates much more quickly, but perform more sophisticated analyses—for example, determine the impact of benchmark selection on AVF across most of SPEC CPU2000 using multi-million-instruction samples, as we have done.

Second, ACE analysis provides a more comprehensive determination of whether faults impact processor opera-

tion. Statistical fault injection methods typically compare the architectural state of the system with a known good copy after some fixed number of cycles. In contrast, we track state through architectural registers and memory locations to properly categorize faults in values that are dead or masked.

Third, ACE analysis gives useful insight into system behavior, such as a breakdown of why un-ACE bits do not contribute to the program result. The application of Little's Law to rough AVF estimation, as described in Section 4.3, exemplifies the usefulness of ACE analysis's more abstract nature.

Finally, fault injection into an RTL model requires an RTL model, which is generally not available during the architectural exploration phase of a microprocessor design project. This paper outlines a technique that can be used to compute AVFs using a performance model, which is typically used for architectural exploration for a processor design. Consequently, using this technique an architect can anticipate the error protection techniques needed for specific structures much ahead of the RTL development. Although statistical fault injection could be applied to a performance model, it would continue to suffer from the shortcomings listed above.

Nevertheless, we expect that analysis of RTL models will continue to be necessary for more accurate fault-rate estimates later in the design process. The application of ACE analysis to RTL appears promising, and is an area of future work.

Additionally, we would like to point out that we draw heavily upon prior work in measuring the architectural behavior of processors. Several papers (e.g., [5], [9], [19], [7]) have quantified the extent of dynamically dead instructions, prefetches, and NOPs used in modern microprocessors. Finally, Gomaa, et al.'s CRTR design [10] relies on logical masking to avoid fault detection checks on certain logical instructions.

8. CONCLUSIONS

Single-event upsets from particle strikes have become a key challenge in microprocessor design. Techniques to deal with these transient faults exist, but come at a cost. Designers require accurate estimates of processor error rates to make appropriate cost/reliability trade-offs. This paper described a method for generating these estimates.

A key aspect of this analysis is that some single-bit faults (such as those occurring in the branch predictor) will not produce an error in a program's output. We call the probability that a fault in a particular structure will result in an error the structure's *architectural vulnerability factor* (AVF). A structure's error rate is the product of its raw error rate, as determined by process and circuit technology, and the AVF.

This paper estimated AVFs using a novel approach that tracks the subset of processor state bits required for architecturally correct execution (*ACE*). Any fault in a storage cell that contained one of these bits, which we called *ACE bits*, would cause a visible error in the final output of a program in the absence of error correction techniques. We called the remaining processor state bits *un-ACE bits*, as their specific values are unnecessary for architecturally correct execution. A fault that affected only un-ACE bits would not cause an error. The AVF for a single-bit storage cell is simply the fraction of time that it held ACE bits. Assuming that all cells have equal raw fault rates, the AVF for a structure is the average AVF of its storage cells, or the average fraction of its cells that held ACE bits at any point in time.

In this paper we conservatively assumed that every bit is an ACE bit unless we could prove otherwise. We identified un-ACE bits at both the architectural and microarchitectural levels. We identified five classes of architectural un-ACE bits. These un-ACE bits come from NOP instructions, performance-enhancing instructions (e.g., prefetches), predicated-false instructions, dynamically dead code, and logically masked bits. Similarly, we identify four classes of microarchitectural un-ACE bits. These are idle or invalid bits, mis-speculated bits, predictor structure bits, and microarchitecturally dead bits.

Using the above methodology, dynamic slices of the SPEC 2000 benchmark suite, and the Asim performance model, we computed the AVF for the instruction queue and execution units of an Itanium2®-like IA64 processor. We found that the AVF of the instruction queue ranges between 14% and 40%, whereas the AVF for the execution units range between 2% and 17%. Because our methodology is conservative, these fractions are upper bounds on the AVF numbers. Further refinement of our analysis (e.g., derating the hint bits in an IA64 load instruction) could further lower the AVF estimates. However, we believe we have captured most of the dominant AVF effects for the IA-64 architecture and Itanium2®-like microarchitecture we examined and, hence, we expect the contribution from further refinement to be small.

Per-structure AVF estimates such as these should help microprocessor designers to estimate the FIT rate of an entire processor early in the design cycle. If the processor does not meet its target FIT rate, then these estimates can help designers choose the appropriate error detection or correction schemes to make specific structures less vulnerable to single-bit upsets. Large structures with high AVFs, for example, would be obvious candidates for such error protection. Typically, parity- or ECC-protected bits are not vulnerable to single bit upsets and, therefore, do not contribute to the FIT rate of a chip. Thus, microprocessor designers can lower the FIT rate of the chip iteratively by adding more and more error protection, using AVF estimates as a guide.

ACKNOWLEDGMENTS

We would like to thank VSSAD members, Intel's reliability experts, and the anonymous referees for useful feedback on this work. We would also like to thank Intel's Asim group for help

with Asim and Intel's SoftSDV group for helping us boot an OS image on the Asim model.

REFERENCES

[1] H. Ando, et al., "A 1.3 GHz Fifth Generation SPARC64 Microprocessor," *International Solid-State Circuits Conference*, 2003.

[2] Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *32nd Annual International Symposium on Microarchitecture (MICRO)*, November 1999.

[3] Robert Baumann, "Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pp. 121_01.1 – 121_01.14, April 7, 2002.

[4] D.C.Bossen, "CMOS Soft Errors and Server Design," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pp. 121_07.1 – 121_07.6, April 7, 2002.

[5] J.A.Butts and G.Sohi, "Dynamic Dead-Instruction Detection and Elimination," *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 199 – 210, October 2002.

[6] T.Calin, M.Nicolaidis, and R.Velazco, "Upset Hardened Memory Design for Submicron CMOS Technology," *IEEE Transactions on Nuclear Science*, Vol. 43, No. 6, December 1996.

[7] Y.Choi, A.Knies, L.Gerke, and T-F Ngai, "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor," *34th Annual International Symposium on Microarchitecture (MICRO*), pp. 182 – 191, Dec. 2001.

[8] Joel Emer, Pritpal Ahuja, Nathan Binkert, Eric Borch, Roger Espasa, Toni Juan, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, and Steven Wallace, "Asim: A Performance Model Framework", *IEEE Computer*, 35(2):68-76, Feb. 2002.

[9] B.Fahs, S.Bose, M.Crum, B.Slechta, F.Spadini, T.Tung, S.J.Patel, and S.S.Lumetta, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization," *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO*), pp. 16 – 27, December 2001.

[10] M.Gomaa, C.Scarbrough, T.N.Vijaykumar, and I.Pomeranz, "Transient Fault Recovery for Chip Multiprocessors," *30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.

[11] S.Hareland, J. Maiz, M.Alavi, K.Mistry, S.Walstra, and C.Dai, "Impact of CMOS Scaling and SOI on soft error rates of logic processes," *VLSI Technology Digest of Technical Papers*, 2001.

[12] T.Karnik, B.Bloechel, K.Soumyanath, V.De, and S.Borkar, "Scaling trends of Cosmic Rays induced Soft Errors in static latches beyond 0.18μ," *Symposium on VLSI Circuits Digest of Technical Papers*, 2001.

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

[14] Kevin Krewell, "Intel's McKinley Comes Into View," *Microprocessor Report*, Volume 15, Archive 10, October 2001.

[15] E.D.Lazowska, J.Zahorjan, G.S.Graham, and K.C.Sevcik, "Quantitative System Performance," *Prentice-Hall, Inc., E140nglewood Cliffs*, New Jersey, 1984.

[16] Shubhendu S. Mukherjee, Mike Kontz, and Steven K. Reinhardt, "Detailed Design and Implementation of Redundant Multithreading Alternatives," *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002.

[17] Eugene Normand, "Single Event Upset at Ground Level," *IEEE Transactions on Nuclear Science*, Vol. 43, No. 6, December 1996.

[18] Steven K. Reinhardt and Shubhendu S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000.

[19] Eric Rotenberg, "Exploiting large ineffectual instruction sequences," *Technical Report, North Carolina State University*, November 1999.

[20] Eric Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," *Proceedings of Fault-Tolerant Computing Systems (FTCS)*, 1999.

[21] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder, "Automatically Characterizing Large Scale Program Behavior," *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[22] P.Shivakumar, M.Kistler, S.W.Keckler, D.Burger, and L.Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic," *Dependable Systems and Networks*, 2002.

[23] T.J.Slegel, et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*, pp 12–23, March/April, 1999.

[24] Y.Tosaka, S.Satoh, K.Suzuki, T.Suguii, H.Ehara, G.A.Woffinden, and S.A.Wender, "Impact of Cosmic Ray Neutron Induced Soft Errors, on Advanced Submicron CMOS circuits," *VLSI Symposium on VLSI Technology Digest of Technical Papers*, 1996.

[25] Nicholas Wang and Sanjay Patel, "Modeling the Effect of Transient Errors on High Performance Microprocessors," *Center for Circuits, Systems, and Software (C2S2), 2nd Annual Review, Berkeley*, March 18-19, 2003.

[26] Alan Wood, "Data Integrity Concepts, Features, and Technology," *White paper, Tandem Division, Compaq Computer Corporation.*

[27] J.F.Ziegler, et al., "IBM experiments in soft fails in computer electronics (1978 – 1994)," *IBM Journal of Research and Development*, pp. 3 – 18, Volume 40, Number 1, January 1996.

[28] J.F.Ziegler, "Terrestrial cosmic rays," *IBM* Journal *of Research and Development*, pp. 19 – 39, Volume 40, Number 1, January 1996.