Computing Architectural Vulnerability Factors for Address-Based Structures

Arijit Biswas[1], Paul Racunas[1], Razvan Cheveresan[2], Joel Emer[3], Shubhendu S. Mukherjee[1] and Ram Rangan[4]

[1] FACT Group, Intel Corp.
Hudson, MA 01749

[2] Sun Microsystems
Santa Clara, CA 95054

[3] VSSAD, Intel Corp.
Hudson, MA 01749

[4] Dept. of Computer Science,
Princeton University,
Princeton, NJ 08544

Abstract

Processor designers require estimates of the architectural vulnerability factor (AVF) of on-chip structures to make accurate soft error rate estimates. AVF is the fraction of faults from alpha particle and neutron strikes that result in user-visible errors. This paper shows how to use a performance model to calculate the AVF of address-based structures, using a data cache, a data translation buffer, and a store buffer as examples. We describe how to perform a detailed breakdown of lifetime components (e.g., fill-to-read, read-to-evict) of bits in these structures into ACE (required for architecturally correct execution), un-ACE (unnecessary for ACE), and unknown components.

This lifetime analysis produces best estimate AVFs for these three structures' data arrays of 6%, 36%, and 4%, respectively. We then present a new technique, hamming-distance-one analysis, and show that it predicts surprisingly low best estimate AVFs of 0.41%, 3%, and 7.7% for the structures' tag arrays. Finally, using our lifetime analysis framework, we show how two AVF reduction techniques--periodic flushing and incremental scrubbing--can reduce the AVF by converting ACE lifetime components into un-ACE without affecting performance significantly.

1. Introduction

Radiation-induced soft errors—caused by neutrons in cosmic rays or alpha particles in packaging material—are becoming an increasing burden for microprocessor designers. The raw error rate per device (e.g., latch, SRAM cell) in a bulk CMOS process is projected to remain roughly constant or decrease slightly for the next several technology generations [4][5]. Thus, unless we add more extensive error protection mechanisms or use a more robust technology (such as SOI), a processor's error rate will grow with Moore's Law in direct proportion to the number of devices we add to a processor in each succeeding generation.

Error protection mechanisms, such as radiation-hardened circuits or architectural redundancy, however, come with significant penalty in performance, power, and area. Hence, excessive protection may make the resulting product uncompetitive in cost and/or performance. Alternatively, a microprocessor with inadequate protection from soft errors may prove useless due to its unreliability. Consequently, designers must carefully evaluate the soft error rate of a microprocessor to decide on the appropriate amount of protection necessary for a target market.

Computing the architectural vulnerability factor (AVF) for all processor structures is a key aspect of such soft error analysis. A hardware structure's AVF is the probability that a fault in that particular structure will result in an user-visible error. A structure's error rate is the product of its raw error rate, as determined by process and circuit technology, and its AVF. A designer can compute a processor's overall soft error rate by summing the soft error contribution of all processor structures.

Mukherjee, et al. [9] introduced the concept of *architecturally correct execution (ACE)* to compute a structure's AVF. ACE analysis divides a bit's lifetime components into ACE and un-ACE intervals. A bit is un-ACE for any interval where its value can be changed without affecting the program's final outcome. Any interval that cannot be proven un-ACE is assumed to be ACE. The AVF for a single-bit storage cell is simply the fraction of time that it holds ACE state. Assuming that all cells have equal raw fault rates, the AVF for a structure can be computed by averaging the individual AVFs of all of its storage cells. Using this methodology, Mukherjee, et al. computed the AVFs of an instruction queue and execution units of an Itanium®2-like microprocessor.

This paper shows how to compute the AVF of address-based processor structures using a level-one write-through data cache, a data translation buffer, and a store buffer as examples. The data cache and data translation buffer are large enough to potentially have a significant impact on a processor's soft error rate. The store buffer holds modified data, so valid entries are quite vulnerable to errors. Each of these three structures consists of a data RAM (random access memory) array and a tag CAM (content-addressable memory) array, but each has different usage characteristics. We also show how the lifetime analysis of the write-through cache can be altered to accurately approximate the AVF of a write-back cache.

This paper makes four contributions related to the AVF computation of these three processor structures. First, we perform a detailed ACE analysis of the lifetime components of the constituent bits in the three structures' data arrays. Our analysis with selected sections of all SPEC CPU2000 benchmarks running on an Itanium®2-like microprocessor results in a best estimate AVF of 6%, 36%, and 4% for the data arrays of a 16KB 4-way set-associative write-through cache, 128-entry data translation buffer, and an 32-entry store buffer.

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

Second, we use a novel approach that we call *hamming-distance-one analysis* to compute the AVF of the tag arrays by tracking false positive matches. A single bit error in the tag array that results in a miss where there should have been a hit is harmless in the data translation buffer and write-through data cache. This miss will only result in the same data being refetched from elsewhere in the memory hierarchy. Therefore, only false positives, errors that result in a incorrect match between an incoming address and a corrupted tag, are of concern in these structures. A single-bit error can only cause a false positive match in a tag that is of hamming distance one—that is, differs in only one bit—from the incoming bits. Thus, we can track these errors by simply tracking tag entries that differ from the incoming CAM bits in one bit. Our analysis shows a best estimate AVF of 0.41%, 3%, and 7.7% for the tag arrays of the write-through cache, data translation buffer, and store buffer studied in the paper. The AVFs of the tag arrays of the write-through cache and data translation buffer are significantly lower than that of the data arrays because errors are only contributed by the individual bits with the potential to cause false positive matches.

Third, we introduce a new technique called *cooldown* to account for limitations of performance simulators. Because performance models typically do not run benchmarks to completion, we may not know the state of an entry in a structure at the end of the simulation. Cooldown continues the simulation after statistics collection has ended to allow these unknown states to be resolved. A cooldown period of 10 million instructions reduces the unknown AVF of the data cache by over 50%. The cooldown results show that to-end times are generally un-ACE, with the average increase in SDC AVF for all structures under 0.2% absolute.

Finally, we examine two simple AVF reduction techniques for the data cache and data translation buffer. Both of these techniques reduce the AVF by converting an ACE lifetime component to un-ACE. For example, let's assume a specific read-to-read time for a byte in a write-through cache is ACE. If, however, we could evict the byte between the two reads, then we result in the following sequence: read-to-evict, evict-to-fill, and fill-to-read, of which evict-to-fill is un-ACE. Thus, by forcing an early eviction, we can convert ACE time into un-ACE, and, thereby reduce the AVF of both the data cache and the data translation buffer. In this paper, we examine two variations of this scheme: one that flushes the cache and translation buffer periodically and a second one that scrubs the cache incrementally to remove single bit errors.

Our results show that flushing the structures every 100,000 instructions can reduce the AVF by over 50% for every structure. For the data cache, the benchmark with maximum loss in IPC ranges from a 0.3% loss flushing every 5 million instructions to a 1.25% loss flushing every 100,000 instructions. On average, 0.02% loss is seen for the 5 million interval, and 0.19% for the 100,000 instruction interval. For the data translation buffer, the maximum loss ranges from 0.05% to 1.77% and the average loss ranges from 0% to 0.56% for the 100,000 instruction flushing interval. Scrubbing reduces the DUE AVF of a writeback cache by 42%.

The rest of the paper is organized as follows. Section 2 and Section 3 provide background on soft errors and three structures we examine in this paper. Section 4 and Section 5 describe the AVF analysis technique for data arrays and tag arrays, respectively. Section 6 describes our methodology and Section 7 describes the results of the base AVF analysis. Section 8 describes the techniques to reduce AVF and their resultant performance. Finally, Section 9 presents our conclusions.

2. Background on Soft Errors

This section describes the background on soft errors. We limit discussions in this paper to single bit errors, which has the first order impact on the FIT rate of a microprocessor [8]. Section 2.1 discusses two different types of errors that arise from particle strikes. Section 2.2 discusses how to compute the AVF of different error types.

2.1. SDC & DUE

Figure 1 illustrates the possible outcomes of a single-bit fault. Outcomes labeled 1-3 indicate non-error conditions. The most insidious form of error is *silent data corruption (SDC)* (outcome 4), where a fault induces the system to generate erroneous outputs. To avoid SDC, designers often employ basic error detection mechanisms, such as parity. This comes at the expense of area, logic and sometimes the lengthening of a critical path.

With the ability to detect a fault but not correct it, we avoid generating incorrect outputs, but often cannot recover when an error occurs. In other words, simple error detection does not reduce the overall error rate, but does provide fail-stop behavior and thereby avoids any data corruption. We call errors in this category *detected unrecoverable errors (DUE)*. Currently, the industry specifies soft error rates in terms of SDC and DUE numbers.

We further subdivide DUE events according to whether the detected error would have affected the final outcome of the execution. We call benign detected errors false DUE events (outcome 5 of Figure 1) and others true DUE events (outcome 6). A conservative system that signals all detected errors as processor failures will unnecessarily raise the DUE rate by failing on false DUE events. Alternatively, if the processor can identify false DUE events (e.g., the error corrupted only the result of a wrong-path instruction), then it can suppress the error signal.

2.2. Architectural Vulnerability Factor (AVF)

As described in the last section, a device's SDC and DUE rates are the product of its device error rate and SDC and DUE AVFs, respectively. In this section we describe the AVF components of this equation and how to compute them.

With a single-bit error model, a device's SDC AVF expresses the probability that a bit flip in that device results in an error in a program's output. A device protected by an error detection or correction mechanism cannot cause an

SDC event, so its SDC AVF—and its contribution to the overall SDC rate—is zero.

The DUE AVF is the probability that a strike will result in a detected unrecoverable error. Only components that have error detection but not error correction (e.g., parity) will have non-zero DUE AVFs. The DUE AVF is the sum of the true DUE and false DUE AVFs (see Section 2.1).

Protecting a structure with an error detection mechanism increases the overall error contribution from the structure. A fault that would have been an SDC event now becomes a true DUE event, so the true DUE rate equals the old SDC rate. However, some faults that would have been benign because the program outcome was unaffected will now be detected, generating false DUE events. Furthermore, error detection schemes generally add extra bits which raise the false DUE rate of the structure as well. Thus, the total DUE AVF of the protected structure will be at least as large as, and probably greater than, the SDC AVF of the unprotected version. Interestingly, an error detection bit, such as parity, is a source of false DUE event.

3. The Three Processor Structures

This paper examines the AVFs of three common address-based structures: a level-one data cache, a data translation buffer, and a store buffer. These structures were chosen because each has very different usage characteristics. The baseline data cache is 4-way set-associative and its AVF is affected by writes, while the data translation buffer is fully associative and read-only. The access patterns of the data cache and data translation buffer are significantly different. The store buffer contains the only valid copy of its data in the memory hierarchy, and can simultaneously contain multiple entries with the same tag.

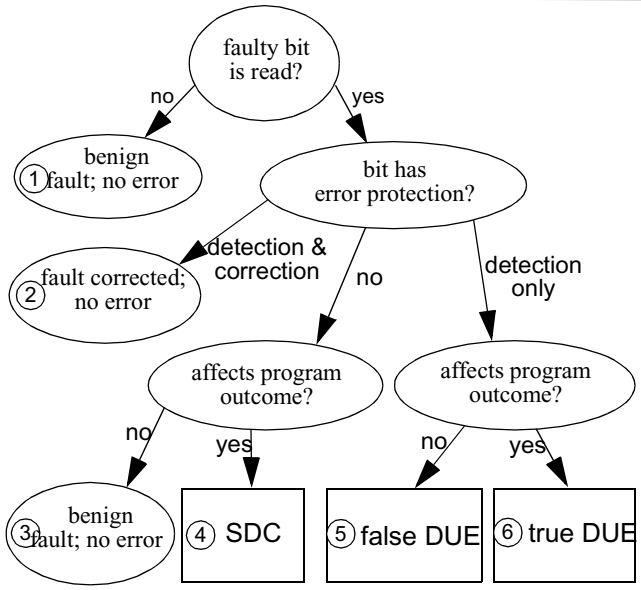

Figure 1. Classification of the possible outcomes of a faulty bit in a microprocessor. SDC = silent data corruption. DUE = detected unrecoverable error.

The processor accesses the data cache on every load and store to read and write data, respectively, from the cache. Our baseline processor model uses a write-through cache in which the store is also written out to the store buffer. We also show how the breakdown of lifetimes into ACE and un-ACE components for the writethrough cache can be modified to closely approximate the AVF of a write-back cache.

The data translation buffer is accessed by every load or store in parallel with the access to the data cache. Unlike the data cache, however, both loads and stores initiate only a read operation on the data translation buffer. Each such read CAMs the tag array with the virtual address. On a CAM hit, a load or a store obtains the corresponding physical address and associated protection information.

Finally, our third structure is a store buffer. Like the data cache, the store buffer is written by store instructions, but read by loads. Unlike the data cache, however, each store creates a new entry in the store buffer. Thus, the store buffer can concurrently hold multiple stores to the same address. Also, the store buffer has per-byte mask bits to identify which bytes have been modified. As soon as a store instruction retires, it becomes a candidate for eviction. When a store is evicted, the pipeline moves it to a coalescing merge buffer from where the data is eventually written into the cache hierarchy. Hence, residency times of entries in the store buffer are much shorter than corresponding ones in the data cache or data translation buffer.

4. Lifetime Analysis for Data Array

Tag-based structures typically have two hardware components: a data array containing the payload and a tag array containing tags corresponding to the payload. For example, in our fully-associative data translation buffer, the tag array holds virtual page numbers and the data array holds the corresponding payload consisting of the physical page number, protection information, etc.

This section examines the SDC AVFs of three RAM arrays—first-level write-through cache, data translation buffer, and a store buffer. Where applicable, we also derive the corresponding DUE AVFs. In Section 4.2 we also show how this analysis is altered to achieve an accurate approximation for the AVF of a write-back cache.

Section 4.1 describes the details of our lifetime analysis. Section 4.2 describes some of the exceptions that must be made to account for the behavior of individual structures. Section 4.3 describes the impact of working set size on lifetime breakdown. Section 4.4 shows why the granularity at which we maintain the lifetime breakdown is critical. Section 4.5 describes the impact of edge effects arising from partial simulation of benchmarks. Section 4.6 describes how to compute the DUE AVF for these structures using the lifetime analysis results.

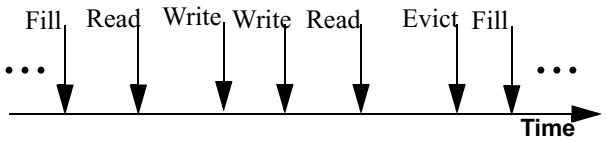

Figure 2. Example activities during a bit's lifetime.

4.1. LifeTime Analysis

Mukherjee, et al. [9] introduced *lifetime analysis* to compute the AVF of a processor's instruction queue and execution units. Lifetime analysis involves dividing up a bit's lifetime during a program execution into ACE and un-ACE components. The AVF is the fraction of the bit's lifetime during which the bit contained ACE state. To compute the AVF we use Mukherjee, et al.'s conservative assumption that the entire lifetime is ACE and then systematically prove which portion of the bit's lifetime is un-ACE. Then, the fraction that cannot be proven un-ACE (ACE and unknown in Table 1), by definition, is an upper bound of ACE time. Also, instead of reporting a per-bit AVF, we report a per-structure AVF, which is the average AVF of all its bits.

To compute the un-ACE fraction of a bit's lifetime, we identify activities during a bit's lifetime that contribute towards un-ACE state. Figure 2 shows example activities occurring during the lifetime of a bit in a data array. The bit begins in "idle" state, but is eventually filled with either ACE or un-ACE state. The bit is written, read, and eventually the state contained in the bit is evicted and re-filled. Reads from or writes to a bit in the data array occur on a tag match in the corresponding tag array. Thus, the lifetime of a bit can be divided up into several non-overlapping components: idle, fill-to-read, read-to-write, write-to-write, write-to-read, read-to-evict, etc. Table 1 shows a detailed classification of lifetimes into ACE and un-ACE components.

By definition idle, read-to-write, and write-to-write are un-ACE. Whether fill-to-read and write-to-read are un-ACE depends on the read itself. For example, if the read is dynamically dead (its value will never be used in future and, therefore, will not affect the final outcome of a program), then the write-to-read time is un-ACE. Other examples of un-ACE reads include those on the wrong-path or those falsely predicated.

4.2. Structure Differences

The store buffer is somewhat unique in that a write to the data bit of one entry can change the ACE status of a data bit in a completely different entry. Consider two stores that write to the same byte of a data address. In a single-processor system, the bits representing this byte in the store buffer entry associated with the older store become un-ACE as soon as the younger store is entered into the store buffer. This is because any subsequent loads to this address will receive their value from the younger store buffer entry.

The write-back cache needs special consideration also. Consider the following scenario in the data array. Two consecutive bytes A and B in the same cache block are fetched into the data array. If A is only read and B is never read prior to eviction, then the fill-to-evict time for B is un-ACE. In contrast, if A is written into, then the fill-to-evict time for B becomes potentially ACE because the entire block (including B) will now be written back into the next level of the cache hierarchy. Thus, an error in B will get propagated. To handle a write-back cache, the lifetime breakdown must be modified. Any bytes of a modified line that have not themselves been modified are ACE from fill-to-evict, regardless of what else may happen to them in the interim. The bytes that have been modified are ACE from last write-to-evict, plus any earlier write-to-read time. The bytes of an unmodified line work identically to those of a write-through cache. Hence, two of the un-ACE components for a write-back cache (as shown in Table 1), fill-to-evict and read-to-evict, can be conditionally ACE at certain times. This extra ACE component could potentially be reduced by adding multiple modified bits, each representing a portion of the cache line.

4.3. Working Set Size

The working set size can have a big impact on a structure's AVF. For example, if in an 128-entry data translation buffer, only one entry is ever used, then the AVF will never exceed 1 / 128. Similarly, if a structure's miss rate is very high, then the AVF is likely to be low because part of the ACE lifetime gets converted to un-ACE time. For example, an intervening eviction between a write and a read—arising possibly from reduction in a structure's size or forced evic-

Table 1: Classification of lifetimes into non-overlapping ACE or un-ACE components. Dynamically dead reads or writes (not shown explicitly) convert ACE into un-ACE components.

Processor Structure	Lifetime Classification		
	ACE	un-ACE	Unknown
Write-through data cache	fill-to-read, read-to-read, write-to-read	idle, fill-to-write, fill-to-evict, read-to-write, read-to-evict, write-to-write, write-to-evict, evict-to-fill	fill-to-end, read-to-end, write-to-end
Write-back data cache	fill-to-read, read-to-read, write-to-read, write-to-evict, write-to-end, some of un-ACE components can be conditionally ACE (see prose)	idle, fill-to-write, fill-to-evict, read-to-write, read-to-evict, write-to-write, evict-to-fill	fill-to-end, read-to-end
Data Translation Buffer	fill-to-read, read-to-read	idle, read-to-evict, evict-to-fill	fill-to-end, read-to-end
Store Buffer	fill-to-read, fill-to-evict, fill-to-end, read-to-read, read-to-evict, read-to-end	idle, evict-to-fill	none

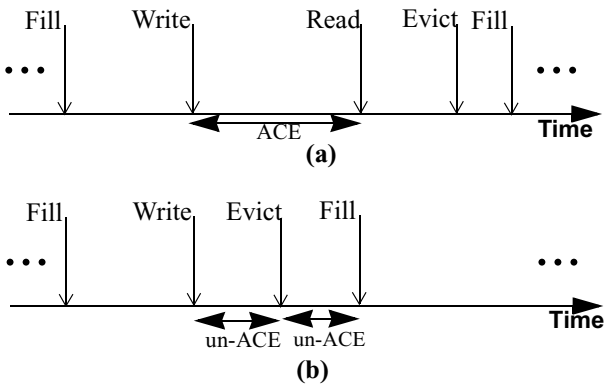

Figure 3. AVF reduction by converting ACE time to un-ACE. (a) shows an example where write-to-read time is ACE. (b) shows how an intervening evict can convert part of the ACE time to un-ACE.

tion—can convert ACE time to un-ACE, thereby reducing that entry's contribution to overall AVF (Figure 3).

4.4. Granularity

The granularity at which we maintain the lifetime information can have a big impact on the lifetime analysis of certain structures, such as a cache. A cache data array is divided up into cache blocks, whose typical size ranges between 32 - 128 bytes. When a byte in a cache block A is accessed, the entire cache block A is fetched into the cache. However, not all the remaining bytes in the block A will be read or written by the processor. When a new cache block B replaces the cache block A, the remaining bytes in block A becomes un-ACE. This is reflected as fill-to-evict time in Table 1, which represents the bytes of data either never used before the line is evicted or used only by the initial access. For the write-through cache, fill-to-evict time constitutes approximately 45% of its total un-ACE time.

For the data translation buffer, we only maintain the ACE and un-ACE components on a per-entry basis. For the store buffer data array, however, we maintain the information on a per-byte basis, just like the data cache.

4.5. Edge Effect

Edge effects arise as an artifact of not running a benchmark to completion in a performance model. For example, in Figure 2, if our simulation ended after the write, then we would not know if the write-to-end time is ACE or un-ACE. If there were an ACE read after the simulation ended, the write-to-end time would be ACE. Conversely, if there were an eviction after the simulation ended, then the write-to-end time would be un-ACE.

Similarly, in Figure 3a, if the simulation ended after the read, we would not know if the read could have been dynamically dead or not. The read could become dynamically dead, if there were a corresponding write to the same address after the simulation ended.

To tackle these edge effects, we introduce the concept of *cooldown*, which is complementary to the concept of warmup in a performance model. A processor model faces a problem at startup in that initially all cache blocks will be empty. If simulation begins immediately, the simulator will show an artificially high number of cache misses. This problem can be solved by warming up the caches before activating full simulation. In the warmup period, no statistics are gathered, but the caches and other structures are warmed up to reflect the steady state behavior of a processor.

Cooldown is the dual of warmup and follows the actual statistics gathering phase in a simulation. During the cooldown interval, we only track events that determine if specific lifetime components, such as write-to-end or read-to-end should be ACE or un-ACE. If after the end of the cooldown interval, we cannot precisely determine if the specific lifetime components were ACE or un-ACE, we mark them as unknown (Table 1).

Cooldown has a marked impact on reducing the unknown portion of a data cache's array. The effect is less pronounced in the data translation buffer and is negligible for the store buffer. Short cooldown intervals quickly help determine if specific lifetime components in a cache, such as write-to-end, are ACE or un-ACE. But, for benchmarks whose working sets fit in the data translation buffer, we have to simulate till the end of a program to precisely determine when an entry becomes un-ACE. In a store buffer, effect of cooldown is negligible because any valid entry is ACE until eviction.

4.6. Computing the DUE AVF

All prior discussions in Section 4 focused on determining the SDC AVF, which assumes no protection for a specific structure. Instead, if these structures had fault detection (e.g., via parity protection) and no recovery mechanism, then the corresponding AVF is called DUE AVF. As described in Section 2.2, we can derive the DUE AVF by summing the original SDC AVF and the resulting false DUE AVF.

In the structures studied in this paper, false DUE AVF from parity protection arises only for a write-back cache and the store buffer. On detecting a parity error, the write-through cache and data translation buffer can refetch the corresponding entry from either the higher-level cache or page table, respectively. That is, with parity and appropriate recovery mechanism, the DUE AVF of both a write-through data cache and data translation can be reduced to zero.

In both the write-back cache and store buffer RAM, however, false DUE AVF arises from dynamically dead loads. When a dynamically dead load reads an entry in the cache or store buffer RAM array, it can check for errors by recomputing the parity bit. If there is a mismatch between the existing and computed parity bit, then the cache or store buffer will signal an error, resulting in a false DUE event. The π bit can help reduce the false DUE AVF [13], but we do not include the π bit in our evaluation. Section 8.2 examines how scrubbing can help reduce the DUE AVF of a cache.

Table 2: un-ACE CAM lookup scenarios on a single bit error

Should Have	Actual Outcome	Potential Error?			Scenario
		Write-through Cache	Data Translation Buffer	Store Buffer	
Mis-matched	Matched	Yes	Yes	Yes	False Positive
Matched	Mis-matched	No	No	Yes	False Negative

5. Lifetime Analysis for Tag Array

The lifetime analysis for tags in cache arrays has both similarities to and differences from that for the data payload. Like in Figure 2, the lifetime analysis for tags also involves monitoring activities on the tag array and identifying the un-ACE portion of a tag's lifetime. Again, read and evict operations can contribute ACE time to bits of the tag. Nevertheless, there are two key differences between the lifetime analysis for tags and data payload: false positive and negative cases (Section 5.1) and hamming-distance-one analysis (Section 5.2).

5.1. False Positive & False Negative Match

A content-addressable memory, such as the tag store, operates by simultaneously comparing the incoming address against the contents of each of several memory entries. As Table 2 illustrates, a single bit error in the tag array results in two scenarios that can cause incorrect execution. In the first scenario the incoming bits would match against a tag entry when it should really have mismatched. We refer to this as the false positive case. This will cause the corresponding data array to deliver the incorrect entry, potentially causing incorrect execution.

In the second scenario, the incoming bits do not match any tag entry, even though they should have really matched. For a write-through cache or a data translation buffer, this would result in a miss, causing the entry to be refetched without causing incorrect execution.

Structures that hold modified data, such as a write-back cache or store buffer, must be handled differently. In these structures, the tag must be correct at eviction time. An incorrect tag at eviction will cause the data to be written to an incorrect memory location. We conservatively assume that any such write will corrupt ACE data. Therefore, all bits of a tag associated with a modified entry are ACE from the time that any byte in that entry was first modified until the time that entry is evicted. This is true even if the store that modified the entry is dynamically dead.

5.2. Hamming-Distance-One Analysis

To track false positives in the tag array, we use a new technique called *hamming-distance-one* analysis. Assuming a single bit error model, an incoming set of bits can cause a false positive match in the tag array if and only if there exists an entry in the tag array that differs from the incoming set of bits in one bit position. In other words, false positives are introduced in those tag entries that are at hamming distance one from the incoming set of bits.

Because the false positive case is caused by one particular bit in a tag entry, the ACE analysis for the tag array must be done on a per-bit basis, rather than on a per-entry or per-byte basis as in the data arrays (Section 4.4). That is, when we match, we mark the bit as potentially ACE. All other bits in the same entry remain un-ACE.

The false negative case is easier to track. The false negative case—mismatch when it should have really matched—occurs when the incoming bits match a tag entry. A single bit error in any bit of the tag entry would force a mismatch. On a false negative match, therefore, all bits in the tag entry are marked either ACE or un-ACE depending on whether a false miss in the structure would cause incorrect execution.

Interestingly, there is a subtle difference between the data and tag analyses. On a single bit error in the data array, the actual execution does not necessarily change because the effect of the single bit error is localized. In contrast, on a false negative match in the CAM array, we may not get an actual error (e.g., as in the write-through cache or data translation buffer). Nevertheless, the error can alter the flow of execution because the hardware would potentially bring in a new entry in the tag array. In our simulation model, we do not track this effect.

To gauge the importance of this effect, we have done limited statistical fault injection experiments with a data translation buffer in a corresponding commercial-grade RTL model. Our experiments with microbenchmarks show that this effect is negligible and does not alter the AVF in any significant way.

Table 3: SPEC2000 benchmarks 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-166	4,700 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	29,000 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

6. Methodology

For our evaluation, we use an Itanium®2-like IA64 processor [6] scaled to current technology. The baseline processor

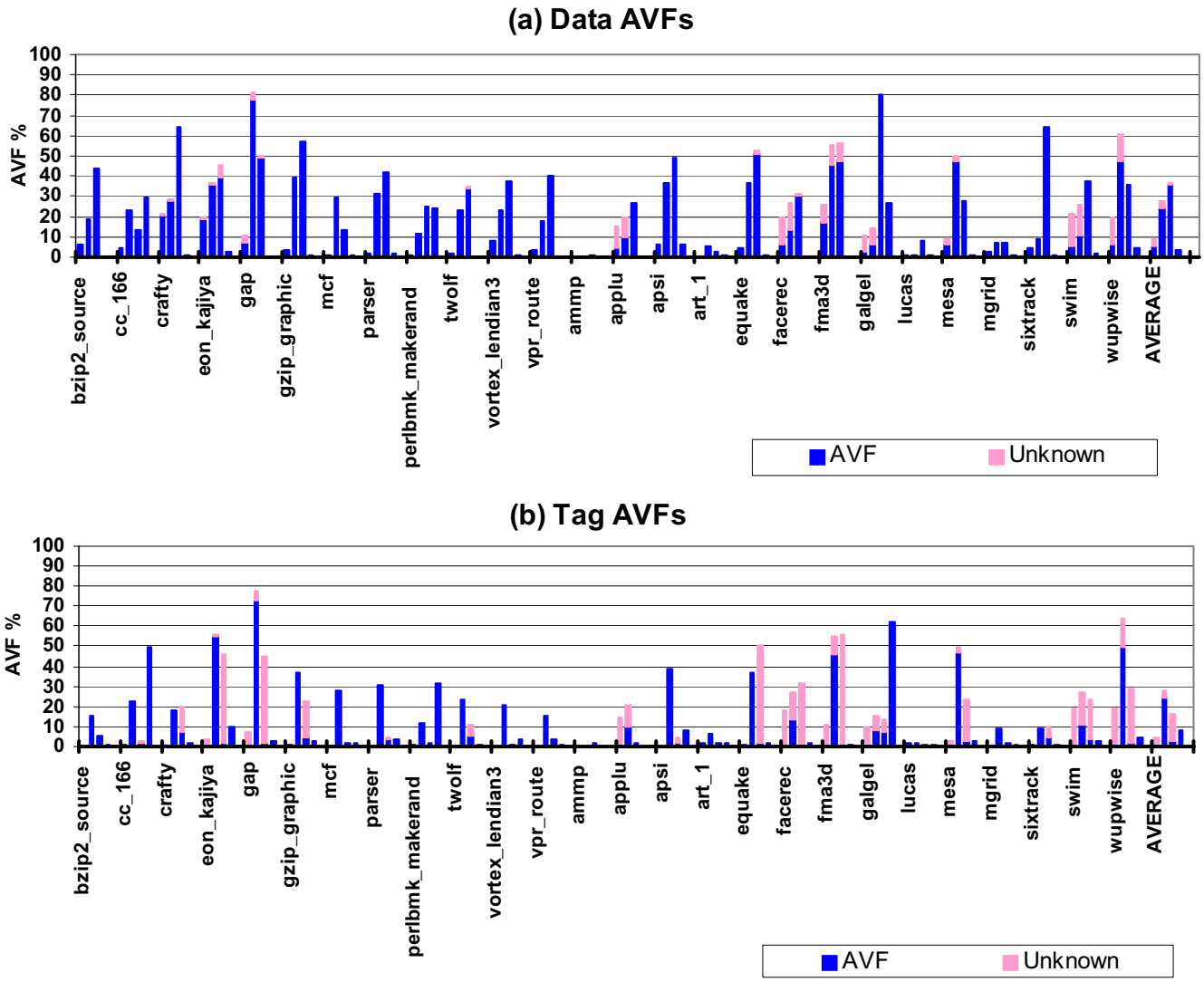

Figure 4. (Left to right: Data cache (write-through), Data cache (write-back), Data Translation Buffer, Store Buffer)

we modeled has a 22-cycle pipeline, runs at 2 GHz, and has an issue width of six instructions. It has three levels of cache: a 16KByte four-way set-associative L1 cache with 32 byte lines, a 512KByte 8-way set-associative L2 cache, and a 4MByte L3 cache. It has a 128-entry fully-associative data translation buffer and a 32-entry store buffer. Each entry in the store buffer can contain up to 16 bytes of data.

The processor is modeled in detail in the Asim framework [3]. The benchmarks are run on Red Hat Linux 7.2 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.

Table 3 lists the skip interval and input set selected for each of the SPEC CPU2000 programs used for our analysis. The benchmarks were compiled with Intel's electron compiler (version 7.0) with the highest level of optimization. We obtained the number of instructions to skip using Sherwood, et al.,'s [11] SimPoint analysis modified for the IA64 instruction set architecture [10]. 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 10 million instructions, which included no-ops.

7. Results

Section 7.1 shows the overall SDC AVF numbers for each structure broken down by benchmark. Section 7.2 shows how much false DUE would be created by adding parity to each of the structures. Section 7.3 examines the results of the cooldown experiments.

7.1. SDC AVF

Figure 4 shows the SDC AVFs for the data and tag arrays of the structures we studied in this paper. From left

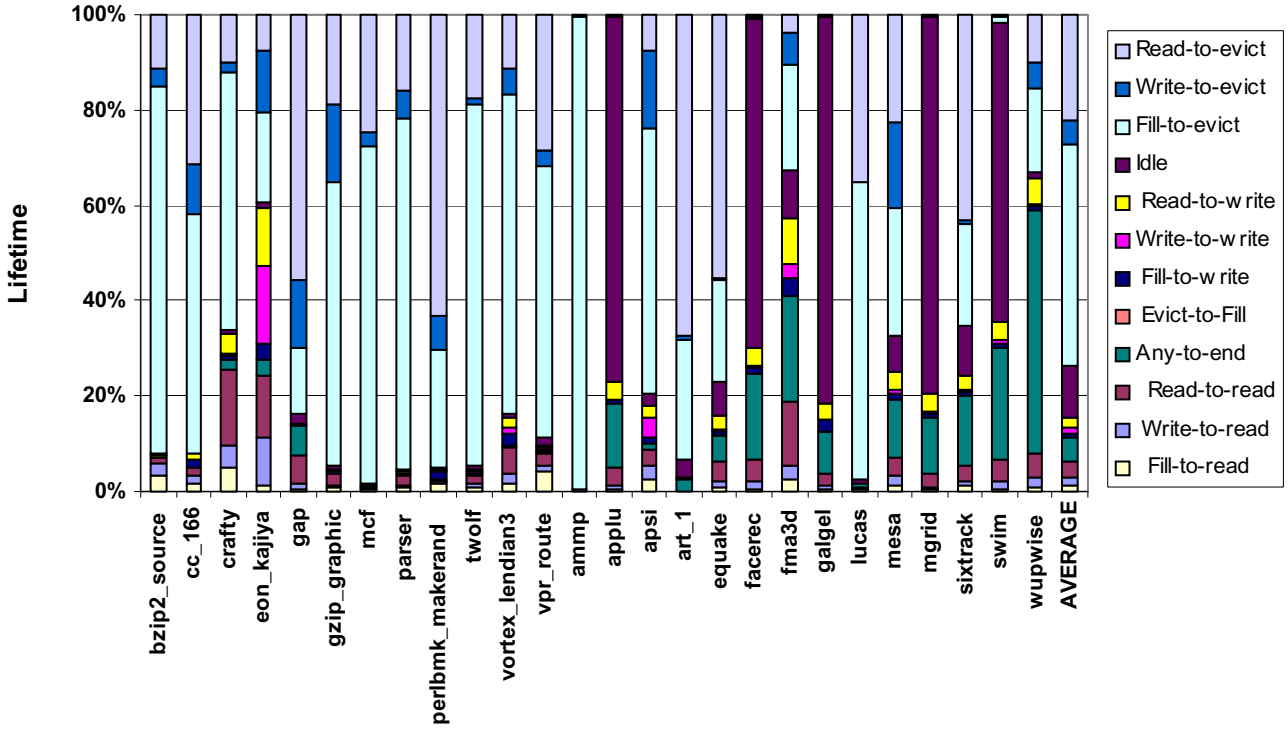

Figure 5. Lifetime Breakdown of a Write-through Data Cache.

to right, the bars associated with each benchmark show the AVF of a 16KB 4-way write-through data cache, a 16KB 4-way write-back data cache, a 128-entry data translation buffer, and a 32-entry store buffer. The y-axis of the graph shows the average SDC AVF of the bits of each structure over the course of the simulation. These experiments were run with a cooldown interval of 10 million instructions and any lifetime component that could not be classified at simulation's end is listed in the graphs as unknown. Section 7.3 will show how cooldown reduces the unknown components in the lifetime analysis of these numbers.

7.1.1. Unknown Components

In the graphs in Figure 4, the unknown component on average shows a maximum at 3% for the data arrays and 4% for the data cache and store buffer tag arrays. The data translation buffer tag array has a significantly higher unknown component of 13%. This is because the translation buffer has a significantly lower turnover rate than the data cache. All of the bits of a tag that do not Hamming-distance-one match with a memory operation will remain in the unknown state until that entry is evicted from the translation buffer. However, nearly all of these bits will eventually resolve to the un-ACE state. Hamming-distance-one matches are rare, and each match only adds ACE time to a single bit of the matched tag. The unknown lifetime components for the data arrays also resolve to un-ACE at a very high rate. This will be discussed in Section 7.3. Throughout this paper, when we refer to the best estimate AVF for a structure, the number we quote does not include the unknown component for this reason.

7.1.2. Data Arrays

The best estimate of SDC AVFs vary widely across the data arrays: from 4% for the store buffer and 6% for the write-through data cache to 25% for the write-back cache and 36% for the data translation buffer. If unknown time is included, these rise to 4%, 9%, 28%, and 38%, respectively. The store buffer's low SDC AVF arises from its bursty behavior and lower average utilization in most benchmarks. Additionally, the store buffer has per-byte mask bits that identify which of the 16 bytes of an entry are written. Entries that are not written remain un-ACE and do not contribute towards the AVF. In the average in-use store buffer entry, only 6 out of the 16 bytes are written.

The data translation buffer's data array has an SDC AVF of 36%, the highest among the data arrays we studied. This is due to its read-only status and relatively low turnover rate. However, as we will see in Section 8.1, flushing the data translation buffer helps increase the turnover rate, thereby reducing the SDC AVF.

The write-through cache's SDC AVF is relatively low for several reasons. These can be best illustrated by looking at a breakdown of the lifetime components for the cache. Figure 5 shows the lifetime component breakdown for a 16KB 4-way set-associative L1 data cache. The analysis is

done on a byte granularity, and the y-axis shows on average, what percent of execution time a byte in the cache spends in each lifetime component. For example, if a cache line is filled and one of its bytes is read 5 cycles later, a contribution of 5 cycles is made to the fill-to-read lifetime component for that byte. This graph does not account for dynamically dead loads, so, in the above example, a fill-to-read lifetime component would be added even if the read were from a dynamically dead load. The graph also does not use a cooldown interval. Doing so would further resolve the any-to-end time into one of the other components.

The graph shows several things about the write-through cache that imply that its SDC AVF should be relatively low. First, on average over 45% of the bytes read into the cache are accessed only on that initial access or not at all. This can be seen from the fill-to-evict bar in Figure 5. A read or write to a specific byte or word brings in a whole cache block full of bytes, but many of these other bytes are never accessed. Second, read-to-evict constitutes a significant fraction of the average overall lifetime (over 20%). These results agree with observations made in previous research [7][14]. Read-to-evict is un-ACE for a write-through cache. It is also un-ACE for a write-back cache assuming that no write preceded the read. Lastly, unlike the data translation buffer, a data cache line is modified by stores. This means that any bytes overwritten by a store are un-ACE in the time period from the last useful read until the write. On the graph, this component of un-ACE time can be generated by adding together the read-to-write, fill-to-write, and write-to-write times. The combination of the three ends up accounting for a little less than 5% of the overall lifetime.

The write-back cache's SDC AVF is 25% compared to 6% for the write-through cache. This is because a write to a byte of a cache line makes all unmodified bytes in the cache line ACE from the time of the initial fill until the eviction of the line. This is true regardless of the intervening access pattern to that line. Since this number cannot be generated from the components in Figure 5, the write-through lifetime analysis was modified to allow recording of this interval. The write-back numbers were then generated from the modified lifetime analysis. The modified bytes of the write-back cache line will be ACE for write-to-evict plus any preceding write-to-read time for those bytes. Per-byte mask bits (as in the store buffer) would help avoid the write-backs of bytes never modified, thereby reducing the write-back cache's AVF.

7.1.3. Tag Arrays

The best estimate SDC AVFs for the tag arrays of the write-through cache and the data translation buffer, which do not include unknown time, are quite low: 0.41% and 3%, respectively. These values are considerably lower than the same values for the corresponding data arrays: 6% and 36%, respectively. The low AVF in these tag arrays arises because the false negative case (Section 5.1)—mismatch when there should have been a match—forces a miss and refetch in these structures, but does not cause an error. In contrast, the false positive case—match when there should be a mismatch—can cause an error, but it only affects the ACE state of the single bit that causes the difference. Consequently, there are significantly fewer ACE bits on average in the tag arrays of these structures compared to the data arrays. The write-through tag AVF is particularly low because a Hamming-distance-one match would have to occur between the four members of a set to contribute to ACE time.

When including unknown time, the tag array SDC AVFs of the write-through cache and data translation buffer increase to 4.3% and 16% respectively. We argue that the lower numbers are more representative of the actual AVFs for these two structures because Hamming-distance-one matches are so rare and each Hamming-distance-one match makes only one bit ACE. If the higher number were used, we would be classifying all bits of the tag as ACE at the end point of the simulation. Also, it is likely that these structures would be effectively flushed by a context switch before the end of the cooldown phase. We will show the results of flushing on AVF in Section 8.1.

In contrast, the SDC AVF of the store buffer is 7.7%, which is, as per our expectation, higher than that of the corresponding data arrays. This is because the store buffer tags are always ACE from fill to evict. The only contributor to un-ACE time for the store buffer tags is the idle lifetime component. The low AVF implies that the store buffer utilization is on average quite low. The tag AVF is higher than the data AVF for the store buffer because all of the bytes in the store buffer entry are not written by each store. On average, an in-use store buffer entry contains only 6 valid bytes out of 16 total. Only the valid bytes contribute ACE time.

The SDC AVF of the tag array of the write-back cache is 25%, but is not directly correlated with that of its data array. This is because two of the un-ACE components for a write-through cache—fill-to-evict and read-to-evict—may become potentially ACE in a write-back cache (see Section 4.1). A write-back cache's tag is always ACE from the time of first modification of its data entry until that entry is evicted.

7.2. DUE AVF

As explained in Section 2.1, a DUE (detected unrecoverable error) occurs when we can detect a fault in a structure, but cannot recover from it. Putting parity on a write-back cache or a store buffer allows us to detect an error in a dirty block, but not recover from it. To compute the DUE AVF we can use the original SDC AVF (same as the true DUE AVF with parity) from Section 7.1 and false DUE AVF arising from dynamically dead loads. Our analysis shows that the false DUE AVF is, on average, an additional 0.2% and 0.5% arising out of dynamically dead loads and stores. Hence, the total DUE AVF for the data arrays of a store buffer and write-back cache are 4.2% and 25.5% respectively.

In contrast, putting parity on a write-through cache or data translation buffer's data array allows us to recover from a parity error by refetching the corresponding entry

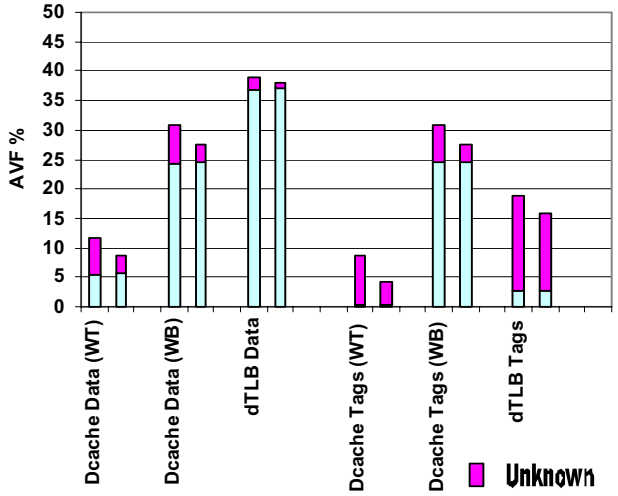

Figure 6. Effect of cooldown with a 10 million instruction cooldown interval.

from the higher-level cache or page table respectively. Consequently, DUE AVF of these two data arrays can be reduced to zero.

The DUE AVF of the tag arrays of both structures is the same as their SDC AVF, since the tag bits are required to be correct even if the store is dynamically dead. An incorrect tag could result in data being written to a random memory location when the entry was evicted.

7.3. Effect of Cooldown

As explained in Section 4.5, edge effects increase the unknown component (Table 1) of lifetime components, thereby increasing the overall SDC AVF. Figure 6 shows the effect cooldown has in reducing the unknown component. The y-axis of the graph shows the average AVF for each structure over all benchmarks. There are two bars associated with each structure. The first bar represents that structure's AVF without cooldown, and the second represents the AVF with cooldown. The gray section of each bar represents the fraction of AVF that is unknown at simulation end. For every structure other than the tags of the data translation buffer, the cooldown period reduces the unknown component by over 50%. Less effect is seen in the data translation buffer because an unknown entry can only be classified after it is evicted, and the data translation buffer has a much lower turnover rate than the other structures. The cooldown graph shows that most entries that reach end of simulation in the unknown state are resolved to be un-ACE. The unknown component of the tags of the data cache and data translation buffer resolves at a ratio greater than 60 units un-ACE to 1 unit ACE. The unknown component of the data portions of all structures resolves at a ratio greater than 10 units un-ACE to 1 unit ACE. The average SDC AVF for all structures increases less than 0.2% absolute, during the cooldown phase.

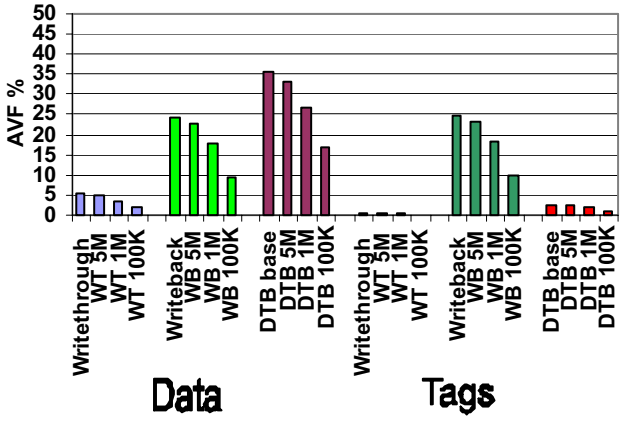

Figure 7. Effect of flushing (with different intervals) on the SDC AVF.

8. AVF Reduction Techniques

As explained in Section 4.3, a high turnover rate in a structure can increase the un-ACE fraction of a structure's lifetime, thereby reducing the AVF. We use the same insight to design AVF reduction techniques. Section 8.1 describes how flushing can reduce the SDC AVF of a data cache and data translation buffer. Section 8.2 describes how scrubbing can reduce the DUE AVF of a data cache.

8.1. Flushing

The first technique simply flushes (or invalidates) the structure periodically. Flushing effectively evicts one or more entries from a structure, thereby achieving the same effect shown in Figure 3b.

Flushing helps reduce the AVF for only two of the three structures—cache and data translation buffer—we examine in this paper. Flushing cannot be used to reduce the AVF of the store buffer because the data is aggressively written out as soon as a store retires.

The basic insight behind reducing the AVF is to convert ACE lifetime components into un-ACE. Flushing a structure periodically achieves this, for example, by converting part of the write-to-read (ACE) time in a write-through cache to write-to-evict (un-ACE). Figure 7 shows the overall SDC AVF reduction from flushing structures at regular intervals. For this experiment, we assume that a flush operation is instantaneous. The y-axis on the graph represents the AVF of each structure averaged over all benchmarks. Each structure has 4 entries on the x-axis, one for the baseline AVF, and one each for flushing at a 5 million instruction granularity, a 1 million instruction granularity, and a 100,000 instruction granularity. Flushing mirrors the cache data displacement that is caused by operating system context switches

Our results show that flushing the structures every 100,000 instructions can reduce the AVF by over 50% for every structure except for the write-through data cache tags. For the data cache, the benchmark with maximum loss in IPC ranges from a 0.3% loss flushing every 5 million

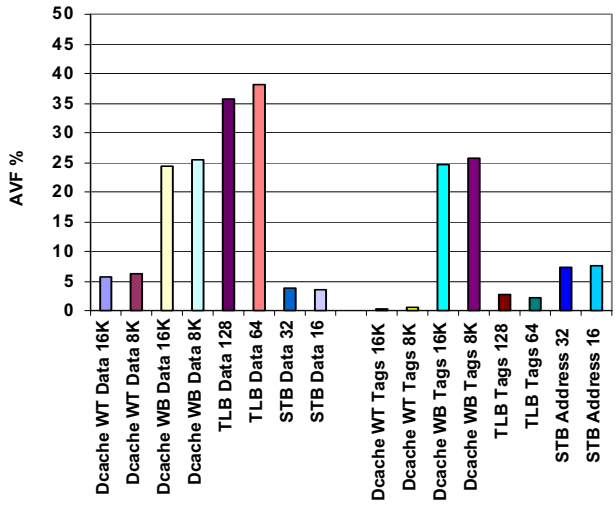

Figure 8. Relative SDC AVFs with halved structure size.

instructions to a 1.25% loss flushing every 100,000 instructions. On average, 0.02% loss is seen for the 5 million interval, and 0.19% for the 100,000 instruction interval. For the data translation buffer, the maximum loss ranges from 0.05% to 1.77% and the average loss ranges from 0% to 0.56%.

8.2. Scrubbing

Parity-protecting a write-back cache brings the SDC AVF down to zero, but introduces the DUE AVF component. One way to reduce the DUE AVF is to use ECC. However, ECC requires extra logic in the critical path of a load's access to a cache for inline correction. To avoid this extra overhead, a processor may choose to avoid the logic for inline correction of errors. Instead, a hardware scrubber can wake up periodically and correct single bit errors in the cache. However, it is impractical for the scrubber to correct single bit errors in all cache blocks in one cycle (this would make it equivalent of inline ECC correction). Hence, a scrubber must periodically wake up, sequence through the cache blocks one at a time, and correct single bit errors in the cache block it examined. This should eliminate a fraction of the errors a load would have seen in the processor's cache. AMD's Opteron™ processor uses such a scheme [1].

We demonstrate that our lifetime analysis for AVF calculation can also easily compute the impact of scrubbing on the DUE AVF of such a write-back cache. Lets assume that a certain fill-to-read component is ACE. If we introduce a scrub operation in between, then the fill-to-scrub becomes un-ACE, whereas the scrub-to-read remains ACE. Thus, scrubbing can reduce the AVF by converting ACE time into un-ACE component. Our analysis shows that the DUE AVF is reduced by 42% with scrubbing with a 16KB cache, 2 GHz processor, and a scrubbing interval of 40ns (which is the best case scrubbing interval for AMD's Opteron™ processor). Our analysis scrubs only on idle cache cycles, to minimize any disruption in processor performance.

8.3. Sensitivity Analysis

Figure 8, shows the effect of halving the size of each structure on its AVF. In general, halving the size of a structure tends to increase its AVF slightly. This is due to the set of frequently accessed lines in the structure. As the structure size decreases, this set of frequently accessed lines makes up a greater fraction of the structures's capacity. These lines are less likely to be evicted and more likely to spend time in ACE lifetime components, such as read-to-read. The exceptions are the store buffer data and translation buffer tag arrays. The AVFs of the data translation buffer tags decrease because the opportunity for Hamming-distance-one matches is reduced when considering 64 other tags instead of 128 other tags. While in general the AVF of the store buffer data array increases slightly when its size is halved, this effect is not seen in the average above. This is because the three benchmarks with high store buffer utilization and AVFs see their AVFs decrease enough to offset the general trend. Reducing the size of the store buffer in a benchmark with high utilization will increase stalls and reduce the average lifetime of entries in the store buffer. The benchmark with maximum IPC loss from halving structure size ranges from a loss of 7.5% for the data cache to 28% for the data translation buffer to 6.3% for the store buffer.

9. Summary

This paper made four contributions towards computing the AVFs of three critical address-based structures--a write-through data cache, a data translation buffer, and a store buffer--each with distinctive hardware characteristics. First, we described how to perform a detailed breakdown of lifetime components (e.g., fill-to-read, read-to-evict) of bits in these structures into ACE (required for architecturally correct execution), un-ACE (unnecessary for ACE), and unknown components. Then, we computed the AVF for the data arrays. Our analysis of a detailed IA64 processor simulator shows best estimate AVFs of 6%, 36%, and 4%, respectively, for the data arrays of the three structures (with realistic sizes). The AVF of the store buffer's data array is particularly low because of low average utilization.

Second, we extended the lifetime analysis for tag arrays to identify false positive (match when there should have been a mismatch) and false negative (mismatch when there should have been a match) cases. To identify false positive matches, we introduced a novel technique called hamming-distance-one analysis, which identifies tag entries that differ by one bit (potentially due to a single bit error) from the incoming match bits. Our analysis shows that the tag arrays of these structures have surprisingly low best estimate AVFs of less than 0.41%, 3%, and 7.7%, respectively. For the data cache and translation buffer, the low AVF arises because a false negative match will force a miss and refetch sequence, but not cause an error. For the store buffer tag, the low AVF arises from low average utilization of the store buffer itself.

Third, we introduced a new technique called *cooldown* to account for limitations of performance simulators.

Because performance models typically do not run benchmarks to completion, we may not know the state of an entry in a structure at the end of the simulation. Running the simulation a little longer through the cooldown interval may allow us to see the eviction of the entry and resolve the original read-to-end time as un-ACE. In the absence of such a cooldown analysis the read-to-end time would be unknown and would increase the AVF in a conservative analysis. A cooldown period of 10 million instructions reduces the unknown AVF of the data cache by over 50%. The cooldown results show that unknown times are generally un-ACE, with the average increase in SDC AVF for all structures under 0.2% absolute.

Finally, we examined two simple AVF reduction techniques for the data cache and data translation buffer. Both of these techniques reduce the AVF by converting an ACE lifetime component to un-ACE. In this paper, we examined two variations of this scheme: one that flushes the cache and translation buffer periodically and a second one that scrubs the cache incrementally to remove single bit errors. We show scrubbing to reduce the AVF of a write-back cache by about 42%. Our results show that flushing the structures every 100,000 instructions can reduce the AVF by over 50% for every structure except for the write-through data cache tags, which essentially have a zero AVF. For the data cache, the benchmark with maximum loss in IPC ranges from a 0.3% loss flushing every 5 million instructions to a 1.25% loss flushing every 100,000 instructions. On average, 0.02% loss is seen for the 5 million interval, and 0.19% for the 100,000 instruction interval. For the data translation buffer, the maximum loss ranges from 0.05% to 1.77% and the average loss ranges from 0% to 0.56% for a 100,000 instruction flushing interval.s evicted.

Acknowledgments

We would like to thank John Crawford and Nelson Tam for their help with the analysis of cache scrubbing, Nick Wang for his work on RTL fault injection, Chris Weaver, Eric Borch and Intel's Asim group for their help with the Asim infrastructure and Harish Patil and Robert Cohn for providing PinPoints for the benchmark set. We would also like to thank Bill Herrick, Steve Raasch, and the other members of the FACT group for their comments on early drafts of the paper and the anonymous reviewers for their useful feedback.

References

[1] AMD, "BIOS and Kernel Developer's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors." Publication #26094, Revision 3.14, April 2004, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF.

[2] D. Bossen, "CMOS Soft Errors and Server Design," 2002 *IRPS Tutorial Notes - Reliability Fundamentals,* April 7, 2002.

[3] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace, "Asim: A Performance Model Framework," *IEEE Computer*, 35(2):68-76, Feb. 2002.

[4] 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.

[5] 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.

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

[7] A. Lai, C. Fide and B. Falsafi. "Dead-Block Prediction and Dead-Block Correlating Prefethers," *28th International Symposium on Computer Architecture*, June 2001.

[8] S.S.Mukherjee, J. Emer, and S.K.Reinhardt, "The Soft Error Problem: An Architectural Perspective," *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feburary 2005.

[9] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *36th Annual International Symposium on Microarchitecture (MICRO)*, December 2003.

[10] H. Patil, R. Cohn, M. Charney, R.Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation," *37th Annual International Symposium on Microarchitecture (MICRO)*, 2004.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[12] N. Wang, M. Fertig, and S. Patel, "Y-Branches: When You Come to a Fork in the Road, Take It," *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[13] C.Weaver, J. Emer, S.S. Mukherjee, and S.K.Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[14] D. Wood, M, Hill and R. Kessler. "A model for estimating trace-sample miss ratios," 1*991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.