Silent Stores for Free

Kevin M. Lepak and Mikko H. Lipasti
Electrical and Computer Engineering
University of Wisconsin
1415 Engineering Drive
Madison, WI 53706
{lepak,mikko}@ece.wisc.edu

Abstract

Silent store instructions write values that exactly match the values that are already stored at the memory address that is being written. A recent study reveals that significant benefits can be gained by detecting and removing such stores from a program's execution. This paper studies the problem of detecting silent stores and shows that an average of 31% and 50% of silent stores can be detected for very low implementation cost, by exploiting temporal and spatial locality in a processor's load and store queues. We also show that over 83% of all silent stores can be detected using idle cache read access ports. Furthermore, we show that processors that use standard error-correction codes to protect data caches from transient errors can be modified only slightly to detect 100% of silent stores that hit in the cache. Finally, we show that silent store detection via these methods can result in a 11% harmonic mean performance improvement in a two-level store-through on-chip cache hierarchy that is based on a real microprocessor design.

1 .0 Introduction

A recent study of store value locality notes that many store instructions write values that are either trivially predictable or actually match the values that are already stored at the memory address that is being written. Such stores are called *silent stores*, since they have no effect on system state. While surprising at face value, this discovery is logically consistent with the plethora of recent research on the value locality of load instructions and register-writing instructions (e.g. [6,7,10,15,17]). If indeed the input values that are being loaded from memory exhibit significant value locality, and the register-to-register computation itself exhibits value locality, it follows naturally that the output values being stored back to memory also exhibit significant value locality. Source-level analysis presented in [1] indicates that many silent stores are algorithmic in nature. Results reported in [14] demonstrate that 20%-68% of all store instructions are silent.

Detecting and squashing silent stores can have a number of beneficial effects: reducing the pressure on cache write ports, reducing the pressure on store queues or other microarchitectural structures that are used to track pending writes, reducing the need for store forwarding to dependent loads, and reducing both address and data bus traffic outside the processor chip. Many of these benefits are examined and quantified in [14]. However, there is also a complexity and microarchitectural resource utilization cost associated with detecting silent stores. Namely, to detect the fact that a store is silent, the prior value must first be read out from the memory location, compared to the new value, and then conditionally overwritten in a process called *store squashing*. The simple store squashing approach outlined in [14] simply issues each store instruction twice: first as a read followed by a compare, and later as a store if it is not silent. Though beneficial overall, it is clear that such a simplistic approach places additional pressure on cache ports, particularly when running programs with few silent stores.

Meanwhile, concerns over reliability and the increasing susceptibility of current and future semiconductor technologies to soft errors induced by gamma rays [24,25] and alpha particles [16] have forced additional complexity into the store-handling logic of high-end microprocessors. For example, the latest high-end processors from Compaq and IBM (the Alpha 21264 and PowerPC RS64-III) protect L1 data caches from soft errors with SEC-DED error-correction codes for each aligned 64-bit quantity. Performing sub-64bit stores into SEC-DED-protected caches requires a read-merge-write procedure for recomputing and storing the ECC for the affected 64 bit parcel.

Store handling has also been heavily complicated by the introduction of out-of-order execution in many current processor cores. In order to track pending requests and guarantee that memory ordering rules are not violated, all outstanding uncommitted loads and stores are tracked in complex hardware structures commonly called load queues and store queues. These queues in fact provide a historical and future context for every individual memory reference by surrounding it with memory references that occur near to it in the program order.

The emergence of both SEC-DED protection for transient error recovery and load/store queues to support out-of-order execution create interesting opportunities for a microarchitect searching for low-cost approaches for implementing store squashing. In this paper, we examine some of these opportunities, ranging from embedding silent store detection into the read-merge-write sequence required for subword stores; to read port stealing; to exploiting temporal and spatial locality in the store and load queues; all to perform store squashing for negligible or reasonably low implementation cost. We find that 31% of silent stores can be identified with the simplest approach that exploits temporal locality only, while a more aggres-

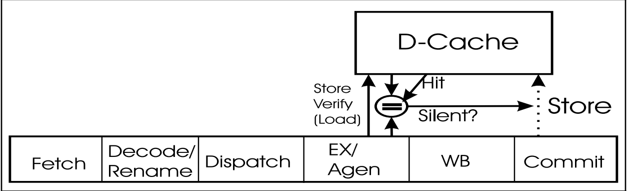

FIGURE 1. A standard store verify consists of load and compare operations in the execute stage..

sive approach that also exploits spatial locality captures an additional 19% on average. Finally, we explore the performance benefits of store squashing in a two-level cache hierarchy with store-through L1 caches that is based on the upcoming IBM Power4 design [13]. In such a configuration, we find that reducing pressure on the memory system can provide up to 56% performance improvement in one benchmark, with a harmonic mean improvement of 11%.

2 .0 Free or Low-Cost Squashing Options

Earlier work showed that performance benefit can be obtained by squashing silent stores for both uniprocessor and multiprocessor systems [14]. Throughout this paper, *store squashing* describes the overall process of suppressing a silent store; *store verification* refers to the subtask of detecting that a store is silent. Further, we assume a weakly consistent memory model when describing the various squashing and verification mechanisms. Of course, some optimizations may or may not be possible with stricter consistency models. Further discussion of consistency model issues is generally omitted for the sake of brevity.

2.1 Explicit Store Verifies for All Stores

In order to understand why we would like to exploit free silent store squashing (FSSS), a review of the original proposed mechanism is necessary to understand its implicit assumptions and potential performance problems. As originally explained in [14], referred to in this work as a *standard store verify*, all store operations are converted to explicit loads, comparisons, and conditional stores. A pipeline diagram is shown in Figure 1.

This implementation has some undesirable characteristics. First, explicitly converting all stores to loads increases pressure on the available cache ports in the system and can potentially delay the issue of loads which are likely on the critical path. Second, having a single instruction perform multiple data cache accesses (and potentially cause many data cache misses) will increase scheduler and control logic complexity. Finally, performing more cache accesses (an additional read for each non-silent store) can increase power consumption. Therefore, we would like to find more efficient ways of squashing silent stores.

In the next sections, we present several alternative implementations of store squashing, each more efficient than the standard store squashing mechanism. We use the term *free* rather loosely to indicate that these mechanisms have a qualitatively lower implementation cost than the standard store verify. Detailed assessment of actual implementation complexity is left to future work. We will use the traditional store verify mechanism as the basis for comparison.

Table 1: ECC data words and required check bits.

Data-word Size (bits)	ECC Check Size (bits)	ECC-word Size (bits)	ECC Check Bit Overhead
8	4	12	50.0%
16	5	21	31.3%
32	6	38	18.8%
64	7	71	10.9%
128	8	136	6.3%
256	9	265	3.5%

2.2 Error Correcting Codes (ECC)

With soft errors in modern microprocessors becoming a larger concern as we move to deeper sub-micron fabrication technologies and higher reliability systems [11,16,21,22,24,25], microprocessor designers are protecting the areas of a chip which are most densely packed with transistors (e.g. caches, memories, etc.) against random alpha-particles and other causes of soft errors. Error checking and correcting (ECC) codes are a very common method for protection against soft errors.

With the incorporation of ECC logic into data caches, even in the L1, as is done in the Alpha 21264 [9] and PowerPC RS64-III [4], silent store squashing becomes much simpler to implement. We return to this point in more detail in Section 3.1 when a possible implementation of squashing in this cache structure is presented, but the basic idea is the following:

ECC using various encoding schemes (we focus on the SEC-DED variety of Hamming based codes [2,20], but the comments made here apply more generally) requires some number of data bits and check bits to enable the correction of errors. The number of check bits is related to the number of data bits by the following function: $n + k \leq 2^k - 1$, where n is the number of data bits and k is the number of check bits. Given the transcendental nature of this function, there is no simple closed form for k, but we illustrate the number of data bits and check bits required for various ECC-word sizes in Table 1.

There is an obvious trade-off between the granularity on which we keep ECC (data-word size) and the overhead of the check bits. In the case of 12 bit ECC-words (8 data bits), there is a 50% increase in storage space as overhead for ECC. For progressively larger ECC-words, the overhead is reduced--down to 3.5% in the case of 265 bit ECC-words (256 data bits). However, this lower overhead does not come without penalty. We can only correct a single bit error and detect a double bit error within the entire ECC-word. Of course, as ECC-word size increases, the probability of multiple errors within a word increases, so ECC is less effective for larger words and a design com-

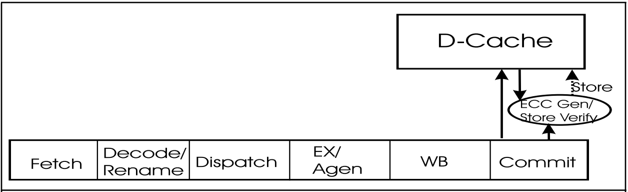

FIGURE 2. ECC store verify occurs at commit.

promise must be reached. In general, fairly large ECC-word sizes are chosen to minimize overhead and obtain acceptable error coverage. In many modern microprocessors and system busses, 64 bit data-word ECC or larger is used for ease of implementation and because of the configuration of memory systems [9,11]. As a point of reference, the Alpha 21264 and the PowerPC RS64-III implement L1 data cache ECC on quadword (64 bit) data quantities.

The check bits for a data-word are generated when a value is stored into the cache and compared when the value is later read (more detail in Section 3.1). In order to generate correct check bits, all bits in the ECC-word must be available as input to the ECC generation logic. Therefore, if we perform a write operation that is either improperly aligned on ECC-word boundaries, or is a sub-ECC-word write, we must first fetch the rest of the original ECC-word stored at the location, merge in the changes (from the current write), calculate the new check bits, and store the ECC-word.

We can see that in many cases the store operation into an ECC-protected cache really consists of four operations: read original ECC-word, store merge, ECC check bit generation, and new ECC-word store. This realization illuminates the possibility of one type of free silent store squashing (FSSS). Since we are reading the original ECC-word anyway, we can perform a comparison of the new store value to the original value and squash the silent stores. This store verify can be performed in parallel with the store merge and new ECC check bit generation, adding very little delay to the store logic, as will be examined in more detail in Section 3.1.

In comparison to standard store verifies (Section 2.1), we can see that store verifies carried out in ECC logic require no explicit load operation, but rather can simply be performed at commit, as illustrated in Figure 2. The drawbacks of this approach are that a store is squashed relatively late in the pipeline (at commit instead of during the execute stage) so it may not reduce pressure on write buffers; it cannot be removed early from the LSQ; and finally that it cannot capture ECC-word-aligned stores.

2.3 Read Port Stealing

It is well known that programs are non-uniform in the usage of system resources. Therefore, in many cases, some available idle resources can be used for other purposes. We propose an additional use of idle resources; namely, exploiting free cache read ports to implement store verifies. This mechanism is a simple extension of the standard store verify explained in Section 2.1. Since stores must commit in order, it is possible that due to a pipeline stall a store can wait in the LSQ for a long period of time before it completes. If a load port becomes free while the store is waiting to commit, we can use the load port to perform a store verify operation. Because we are using resources that are idle and available, these store verifies are free. If a load port never becomes available before the store is ready to commit, we forego attempting to squash the store and assume it is non-silent.

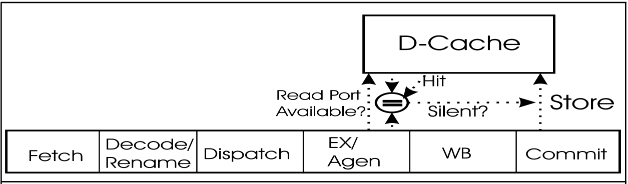

FIGURE 3. Read port stealing performs a load and compare only if a cache port is idle.

Relative to standard store verifies, this method has the benefit of not delaying execution of load operations due to resource conflicts. However, it can create additional instruction scheduling difficulties because the policy for issuing a store verify is dependent on resource usage and not just program order or another static scheduling policy. This technique of FSSS is shown in Figure 3.

2.4 Load/Store Queue

In order to obtain high performance, many processors implement aggressive memory systems which require load/store queues (LSQs) to perform store to load forwarding and monitor speculative load operations which may be violations of the architected consistency model. We can exploit locality in the LSQ to obtain FSSS as outlined in the following sections.

2.4.1 Temporal Locality in the LSQ

Store to load forwarding of memory dependences is an optimization commonly implemented in modern microprocessors. In the case of store squashing, a store verify operation necessitates a read. If store forwarding is implemented, we can extend it to squash later stores to the same address as an earlier store in the LSQ (WAW dependence). We can do so without using a cache read port, hence making the squash free.

In a similar fashion, we can also squash stores to memory addresses for which an outstanding load exists in the LSQ. This is possible because the cache access for the load will be performed, obtaining the data value for the store verify. In some sense, we can consider the store verify for the store to be "piggy-backed" on the explicit load operation to the same memory address (WAR dependence). Note that this optimization is also possible for loads which occur later in program order, which generally would have their load value forwarded from the previous store we're trying to squash. This is possible because the usage of the cache port is usually scheduled before it is known whether the value will be forwarded from an earlier store in the LSQ [8,13]. Therefore, since we have scheduled the load for cache access anyway, the load can

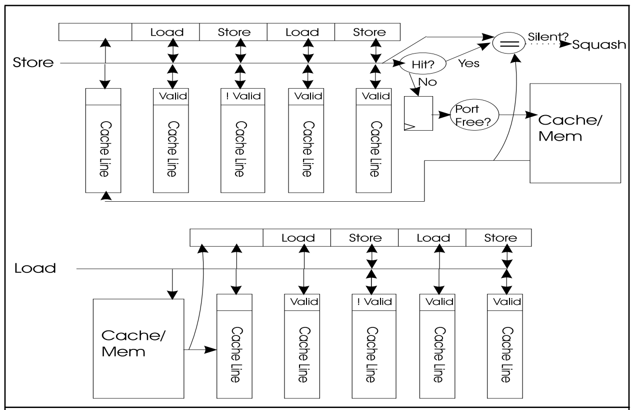

FIGURE 4. Block level LSQ cache design. The temporal and spatial LSQ squashing operations, data allocation, and store forwarding are illustrated for memory operations.

still be performed at no cost. Hence, the store verify is again free in the case of a RAW memory dependence.

2.4.2 Spatial Locality in the LSQ

In a similar fashion, we can expand the scope of stores squashable within the LSQ to addresses that inhabit the same cache line. Given that L1 data caches are on-chip, obtaining wide access to these caches is relatively easy. Therefore, one may imagine each memory operation reading an entire cacheline on any reference because of the high bandwidth available from the L1 cache. Assuming that a memory access reads the entire line from the cache into a *LSQ cache* (shown in Figure 4), we can use the spatially local data to perform additional squashing.

In the case of a WAW dependence, a previous store to the line reads the line into the LSQ cache, and all subsequent stores to that line can be squashed from the LSQ cache. In the case of a WAR and RAW dependences, a similar process occurs--the load operation allocates the line in the LSQ cache, and stores to the same line are squashed from it. We will show in Section 4.4 that a small LSQ cache is especially effective in the case of WAW dependences.

The LSQ cache is similar to the *write cache* proposed in [12], except it contains entire cache lines as opposed to 8 byte quantities and it buffers both load-allocated and store-allocated lines. Also note that since issuing stores is generally not as time critical as issuing loads (because the stores can be buffered at commit) we serialize the lookup in the LSQ cache and the access to the memory system (shown in Figure 4) to avoid unnecessary usage of the data cache port. We can also exploit read port stealing (Section 2.3) and only read data for stores into the LSQ cache if a memory read port is available. Assuming we do so, we also need a separate valid bit both for the LSQ cacheline data and for the entries in the LSQ themselves (shown in the Figure 4) because a store may fail to acquire a free read port, leaving the data invalid. When an access allocates a line into the LSQ cache, we may choose to verify stores already present in the LSQ with the newly allocated data, but this may add complexity to the LSQ and LSQ cache for additional data paths. We will discuss this further in Section 4.4.

If we assume that the LSQ cache is FIFO allocated and is operated in lock-step with the entries in the LSQ such that when an entry leaves the LSQ its LSQ cache line is also deallocated, we can avoid having explicit tags and dirty bits in the entries (because all necessary address and dirty value forwarding is already available in the LSQ entries for store-forwarding). In the case of a weakly consistent system, it is sufficient for correctness to flush the LSQ cache on memory barriers (and this is most likely very effective because of the small LSQ cache size) and avoid snooping it for invalidates. In more strict consistency models, snooping the LSQ may already be required to detect consistency model violations, so snooping the LSQ cache as well adds no additional complexity [23].

The benefits of squashing in the LSQ relative to standard store verifies are apparent. No additional cache access is required for the load portion of the store verify and squashing stores is free.

3 .0 ECC Free Silent Store Squashing

As touched on briefly in Section 2.2, soft errors are a growing concern for microprocessor architects in order to provide highly reliable systems and because of manufacturing concerns [11,16,21,22,24,25]. Having discussed in Section 2 the opportunities for FSSS, in this section we elaborate on the ECC method of FSSS in greater detail. We show three possible mechanisms for protecting L1 data caches from soft errors and illustrate under what circumstances the FSSS techniques can be exploited. We also explore which of the techniques we expect to be most effective for different cache architectures.

3.1 L1 Data Cache with ECC

Soft error protection can be performed in the L1 data cache directly, as is done in the Alpha 21264 [9] and the PowerPC RS64-III [4]. The 21264 and PowerPC RS64-III use 64-bit ECC data words. As shown in Section 2.2, this provides error coverage for relatively low space overhead of approximately 11%. As also outlined in that Section, FSSS is trivially implementable as part of ECC check bit generation for subword writes. In order to illustrate the argument made in Section 2.2, Figure 5 shows a datapath with which ECC may be implemented on a sub-ECC-word store operation in an Alpha-like system. Note that we use 72-bit ECC words (instead of the 71 used in standard Hamming-based codes) because the Alpha uses a slightly modified coding scheme with 72-bit words [9].

Implementation will be slightly different to handle smaller bit width stores, but for ease of illustration, only a 32-bit store is shown. We see the four major operations as discussed in Section 2.2: read the original quadword from the data cache, merge the store data into the input side of the ECC Data Register, generate ECC check bits, and store the quadword and ECC bits. Note that if ECC-word generation takes multiple cycles (as one might expect for

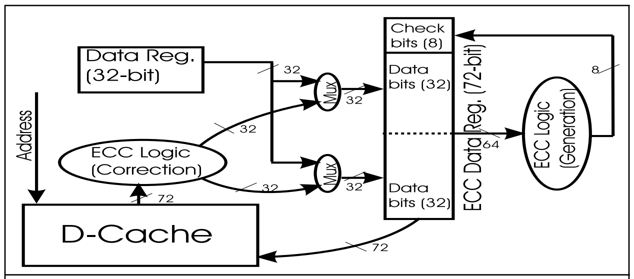

FIGURE 5. L1 data cache ECC-word generation on a sub-ECC-word store.

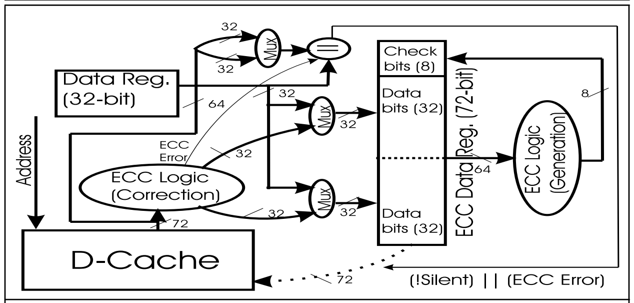

FIGURE 6. L1 data cache ECC-word generation on a sub-ECC-word store with free silent store squashing.

essentially a read-modify-write sequence), we must maintain atomicity of the sequence either through design of the write buffer feeding the ECC logic, or in the logic itself. We have ignored this detail to simplify the diagram.

In Figure 6, we show the implementation of FSSS in the same ECC logic structure as shown in Figure 5. We can see that the changes to the datapath are relatively simple; the addition of an extra multiplexor and a comparator. Figure 6 also illustrates that we cannot perform silent store squashing if an ECC error is encountered on the read of the data value from memory. This is because the corrected value is obtained from the ECC correction logic and therefore must be written back to the memory system. The logic implements the same four steps as described previously. However, the store merge, ECC check bit generation, and new ECC-word store operations may be aborted if it is determined that the store is silent and there is no ECC error. The abort operation can be as simple as not re-acquiring the cache port for the write of the (silent) ECC-word from the ECC Data Register.

The most important aspect of Figure 6 is when the silent store comparison can be performed. From the datapath shown, we can see that the comparison can be performed in parallel with the ECC check bit correction and generation. In general, ECC correction and generation logic consists of trees of exclusive-or gates [20] which have delay on the same order as the 32-bit comparison for squashing. Therefore, FSSS for sub-ECC-word stores can be implemented in an ECC-protected L1 data cache for simply the cost of a few extra gates which should not increase the ECC logic's critical path.

3.2 Store-through L1 Cache with ECC L2

Implementing ECC protection directly is not the only way to combat soft errors in the L1 data cache. In fact, adding ECC protection to the L1 directly can contribute negatively to cycle time because the ECC correction logic is now added to the critical path on load operations to assure usage of corrected values from the cache. Of course, speculation can be used in order to move the ECC check/correction logic off the critical path by speculating that all load values are correct and recovering if the ECC logic reports an error. Of course, this adds control complexity to trigger the recovery [9].

An alternative is to use an L1 cache with simple parity protection and a store-through policy backed up with an ECC L2 cache. The L1 parity protection has a few advantages when compared with ECC in the L1. First, parity can easily be kept on a byte basis with the same overhead as the 72-bit ECC-word as in the 21264 (in both cases the overhead is approximately 13%.) With byte parity in the L1, there are no merging issues with store operations because the smallest atom for memory operations is a byte--therefore stores into the L1 do not require a read-modify-write sequence. The parity for each byte can be calculated very early in the pipeline when the store value is known and can simply be written into the cache. The single bit of parity for each byte provides single error detection on the byte level, as opposed to double error detection over 64 data bits as provided by 64-bit SEC-DED. If an error is detected in the L1 data cache via parity, the correct value is fetched from the ECC L2 cache.

Of course, a major caveat of this approach is the additional bus traffic generated by implementing a store through L1 cache [12]. This traffic can be reduced with techniques like aggressive write combining and other buffering techniques, but special care must be taken to handle the extra L1 to L2 bandwidth requirements. Weaker consistency models allow greater freedom for store combining than stricter models.

In the case of a store-through L1 cache, silent store squashing can have a noticeable performance benefit. To further improve performance, we can use ECC squashing for sub-ECC-word writes in an ECC-protected L2 cache. However, this will not reduce store-through traffic on the L1 to L2 interface. Instead, we rely on the other methods of FSSS--squashing in the LSQ and stealing read ports--in order to reduce store-through traffic. Performance results of the different methods of FSSS for such a memory system configuration are given in Section 4.

3.3 Duplication of L1 Data Cache

We can also obtain single error detection and correction capability in the L1 cache by duplicating it and protecting both copies with parity. If we encounter a parity error on the read of any byte, we can fetch the correct byte from the other copy of the cache to recover from the error. This scheme avoids a read-modify-write sequence for sub-word stores. It also provides effectively double the read-port bandwidth into the L1 data cache because each copy of the data cache can be accessed with loads to arbitrary addresses.

However, this scheme is not without its flaws. First, this scheme has high overhead of 100% compared to a cache with only parity. Second, this scheme does not allow easy scaling of store bandwidth because both copies must be consistent, requiring stores to write both copies.

FSSS can still provide performance benefit in this cache structure because it is biased towards more read ports than write ports. Therefore, we would expect the performance improvement of FSSS in this cache configuration to be similar to results reported in [14] and do not explore this configuration further in this work.

4 .0 FSSS Performance Benefit

We have shown in Section 2 and Section 3 that many opportunities exist for FSSS. In this Section, we quantify the performance benefit of the mechanisms compared with a standard architecture. As we have stated previously, the squashing mechanisms we evaluate are *free* (as defined in Section 2.1) so any non-negligible performance benefit is proof that these methods are effective.

We perform only uniprocessor simulations to show proof-of-concept for the proposed mechanisms. Of course, as shown in [14], there are additional savings for communication misses in multiprocessors that are not considered in these results.

4.1 Machine Model

To determine the performance impact of FSSS, we used an execution driven simulator of the SimpleScalar architecture with an enhanced memory system model [5]. The default SimpleScalar does not accurately (or in some cases at all) model finite memory system components such as write buffers, writeback buffers, scheduling of write/writeback traffic over the L1 to L2 interface, etc. Since FSSS focuses on improving memory system performance, modelling these resources accurately is necessary for our results to reflect true performance.

In order to model the increasing demands on a memory subsystem, we used an aggressive out of order design. The configuration of the execution engine is 8 issue; 64 entry RUU; GShare branch predictor with 64K entries, 16 bit global history; 6 integer ALUs, and 2 integer multipliers. The cache configurations are 64KB each split I/D L1 and 512KB unified L2 with latencies 2, 8, and 50 clocks for the L1, L2, and main memory, respectively. The I-cache is 2 way associative with a line size of 64 bytes; The D-caches are 4 and 8 way associative with line sizes of 32 and 64 bytes, respectively. Store to load forwarding is implemented in the simulator with a latency of 2 clocks to match the L1 hit latency. All binaries are SimpleScalar PISA and compiled with SimpleScalar gcc at -O3.

The machine has two fully pipelined general memory access ports each of which can handle either one load or one store per cycle with no address restrictions. If a store has begun verification, we count this store as verified in the percentages reported, but we do not force verification to finish before committing the store. If a store has not finished verifying when it reaches commit, it is assumed to be non-silent and enters the memory system. Read port stealing for squashing occurs regardless of where an address hits in the memory hierarchy. The simulator implements two write buffers outside of the instruction window (i.e. only for committed stores) where committed stores are held until their completion. Aggressive write-combining is implemented in the write buffer so that any store to the same L1 cacheline can be combined with other stores to the same line in the buffer. The LSQ cache only allocates for stores when it can steal a read port.

The L1 cache has a write-through, write-allocate policy backed by a writeback L2. In all cases (except Section 4.5 where we consider this bandwidth specifically) we make the very aggressive assumption that there is a full L1 cache line width interface between L1 and L2 that can begin a new transaction every clock cycle, as might be possible with on-die L2 caches. Both store-through bandwidth and L1 fill bandwidth are modeled over this interface. Fill transactions (i.e. demand misses) take precedence over store-through traffic on this interface.

The memory access configuration of this machine model is similar, though not identical, to the Power4 [13] which implements a store-through L1 and writeback L2 for ECC protection (as outlined in Section 3.2). It is not our goal in this Section to advocate a specific method of error correction, but rather to show how FSSS can be exploited for performance benefit in one possible configuration.

4.2 ECC Squashing

We do not show performance results for this method of FSSS because it does not make sense to complicate the results discussion with two incomparable machine models. As discussed in Section 3, if a store-through L1 cache is implemented for the purposes of error protection, we have no need for ECC in the L1, since the store-through to an ECC-protected L2 provides adequate reliability. In order to meaningfully demonstrate the performance of ECC squashing, we need a writeback L1. Results for a machine model similar to this were published in [14].

However, it should be noted that the key assumption of Section 2.2 and Section 3.1, namely that store operations must be sub-ECC-word for ECC FSSS, is realistic given commercially available processors today. One would expect that no architect would design a system with the maximal store atom size being smaller than the ECC-word-size so that every store incurs a read-modify-write for ECC generation. However, this occurs frequently. The IBM RS64-III (Pulsar) processor, in use in IBM S80 servers and other machines, executes exactly this way when running 32-bit code. In the RS64-III, the L1 cache is ECC protected directly (similar to the manner discussed in Section 3.1) using an ECC-data-word size of 64 bits. In 32 bit mode, the largest integer store atom is 32 bits, hence incurring the read-modify-write on every store [3,4]. Therefore, we expect ECC squashing to provide significant performance benefit in this and similar systems.

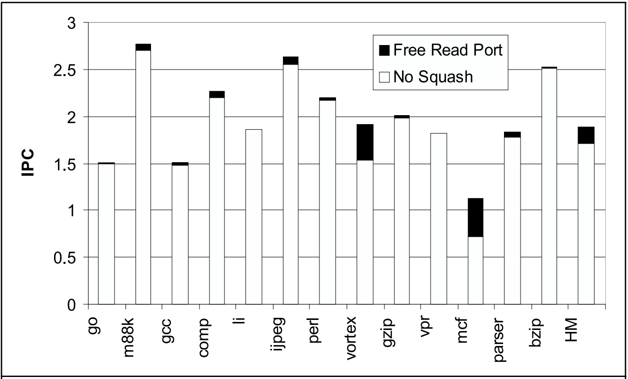

FIGURE 7. Performance improvement of read port stealing vs. no squashing.

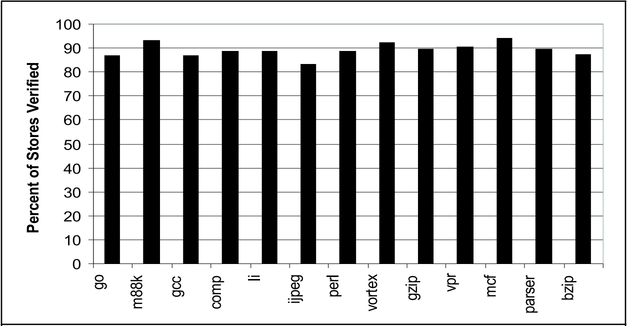

FIGURE 8. Percentage of all dynamic stores verified using only available cache read ports.

4.3 Available Read Port Squashing

Figure 7 shows the performance improvement of read port stealing over the baseline performance with no squashing. We see improvements ranging from a low of 0% in *li* and *vpr* to a high of 56% in *mcf*. The harmonic mean across all benchmarks shows a 10.3% improvement.

It is worthwhile to note that we do not see a performance decrease in any benchmark. This occurs because we are only using cache read ports available after all other ready loads and stores have had a chance to issue/commit. The performance benefit comes primarily from two factors: a) a reduction of bandwidth required between L1 and L2 caches by eliminating store traffic on the interface, and b) reduced pressure on write buffers.

It is also interesting to note how few store squashing opportunities we miss by only using available cache read ports as opposed to trying to squash all stores. In Figure 8 we show the percentage of store operations we are able to store verify for free using read port stealing.

We can see that in all cases, we are able to verify over 83% of store operations using available cache read ports with an average of 89%. This indicates that we are achieving almost all available benefit from squashing that uses the standard store verify, but without impacting performance of critical load and store operations.

4.4 LSQ Squashing

In Figure 9, we show the performance improvement of

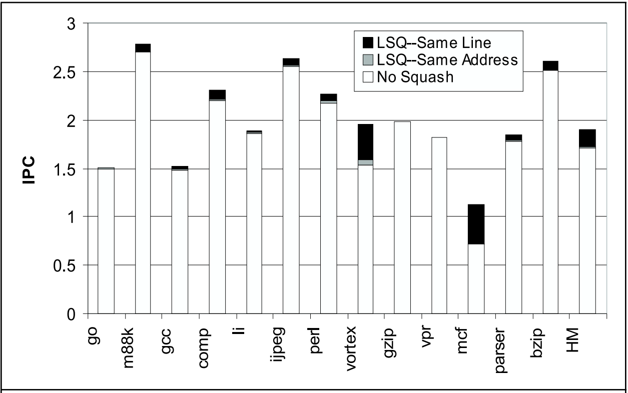

FIGURE 9. Performance of LSQ squashing. The stacked bars indicate the performance of the baseline system (without squashing), same address (temporal) LSQ squashing, and same cacheline (spatial) LSQ squashing, respectively.

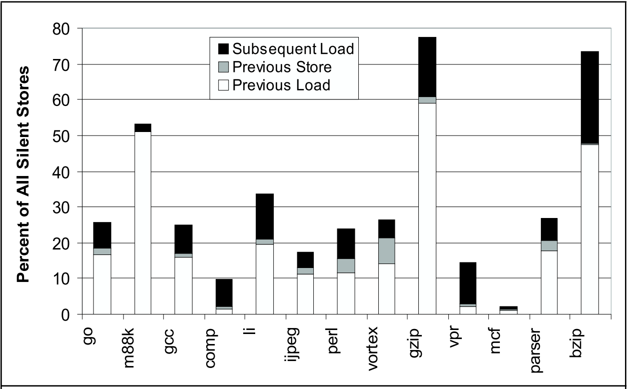

FIGURE 10. Temporal LSQ squashing provided by WAR, WAW, and RAW dependences.

temporal and spatial LSQ squashing over the baseline performance with no squashing (as discussed in Section 2.4.1 and Section 2.4.2, respectively). The stacked bars show the contribution of each mechanism to overall performance. For temporal LSQ squashing, we see improvements in IPC ranging from a low of 0% in *gzip* and *mcf* to a high of 3% in *vortex* with overall performance improved by 0.6% as indicated by the harmonic mean over all benchmarks. When we add spatial LSQ squashing, we see total improvements over the baseline from a low of 0% in *gzip* to a high of 56% in *mcf* with the harmonic mean improving by 11.3%.

When examining temporal squashing, it is interesting to note that most of the stores are squashed by preceding or subsequent load operations (the RAW and WAR dependences discussed in Section 2.4.1), as opposed to previous store operations (WAW dependences), as illustrated in Figure 10. In most benchmarks (except *compress*, *ijpeg*, *vpr*, and *mcf*), temporal LSQ squashing captures over 25% of all silent stores within the dynamic program execution. Some possible explanations for this are provided in [1], and could include program model considerations like stack frame usage. In the results presented in Figure 10, each dynamic silent store is counted at most once (it is present in only one section of the

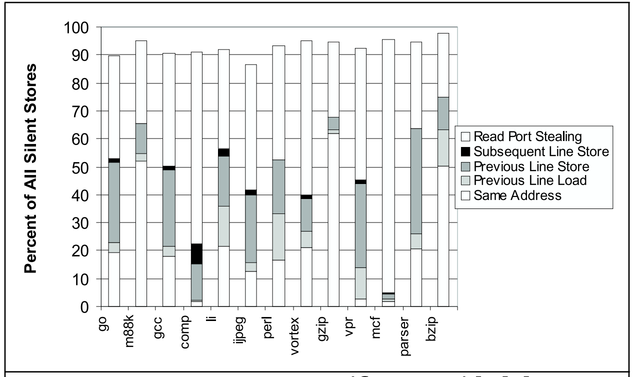

FIGURE 11. LSQ store verifies provided by same address (temporal locality), previous load to line, previous store to line, subsequent store to line, and read port stealing.

stacked bars), with the following priority counting on multiple aliases: previous load (WAR), previous store (WAW), subsequent load (RAW).

In the case of spatial LSQ squashing, the same statement regarding counting of squashable stores holds (a dynamic silent store is only counted once). However, the priority of counting changes slightly due to simulator implementation issues. In this case, the counting precedence is: WAR, WAW, cache line previous load, cache line previous store, RAW, cacheline subsequent load, and read port stealing. We show the results of this method of counting in Figure 11 (results from all same address squashing methods are combined in the Same Address bar for readability and the subsequent line load section is removed because it did not contribute meaningfully). Note that the total percentage of silent stores captured by this mechanism is greater than the results presented in Figure 8 (simple read port stealing) because the LSQ cache is store-allocating using free read ports, as well as exploiting locality in the LSQ. Because LSQ store verifies do not consume a cache port, a port tends to be free more often for additional read port stealing store verifies.

We see that the percentage of same address store verifies decreases over Figure 10, mainly due to counting precedence. Also, substantial previous line store verifies are observed, indicating that the LSQ cache proposed in Section 2.4.2 is useful. These results also indicate, due to the small fraction of subsequent line verifies, that verification from a line allocated by a subsequent access to previous stores in the LSQ is unnecessary for squashing purposes, potentially saving some complexity in the LSQ cache.

Finally, we see that in all benchmarks (except *compress* and *mcf*), over 40% of all silent stores are captured by exploiting locality in the LSQ. Read port stealing for LSQ cache line allocation brings the total percentage of silent stores captured to over 90% (except for *ijpeg*).

In comparing temporal to spatial LSQ squashing, we see only two benchmarks that benefit from temporal squashing (*perl* gains 1.5% and *vortex* 3.3%). It is not until spatial LSQ squashing is applied that we see noticeable improvements in instruction throughput. This occurs because the overall percentage of silent stores detected by the spatial scheme (including free read port squashing) is much higher.

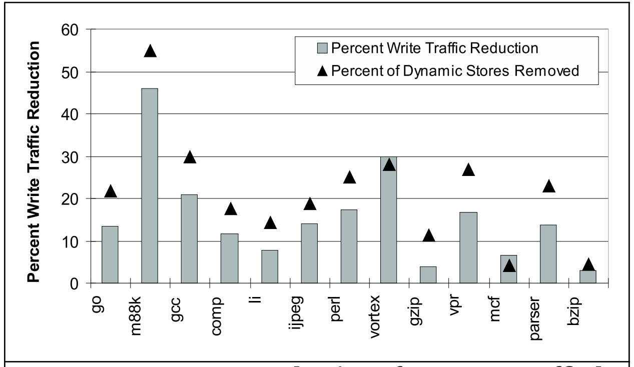

FIGURE 12. Percent reduction of L1 to L2 traffic by performing FSSS. The bars and bullets indicate the percentage of write through traffic reduction and the percentage of total dynamic stores removed, respectively, by FSSS.

4.5 Increasing Effective Write-Through Bandwidth via FSSS

Given that FSSS can squash many silent stores, it is interesting to examine what kind of trade-offs we can make as an architect with this type of memory system to obtain sufficient throughput between the L1 and L2 caches. We can use the "brute force" method and implement a fully-pipelined, write-combining, cache-line-width interface between L1 and L2 (as used in all results presented so far) which can induce significant circuit design complexity. Or, we can exploit FSSS to obtain "effective" throughput over the L1 to L2 interface with less physical throughput. In order to illustrate this, we present Figure 12 which shows the store-through traffic reduction over the L1 to L2 interface as well as the percentage of dynamic stores removed by FSSS. We see an average traffic reduction of 15% across all benchmarks and up to 45% in *m88ksim*. Since this interface is wide (32B) and fast (single cycle pipelined), it is reasonable to assume that this traffic reduction would lead to a savings in chip power.

Note that, as we would expect, the percentage of write through traffic reduction closely mirrors the percentage of successfully removed, squashed, stores. In the case of *vortex* and *mcf*, the traffic reduction is slightly greater than the percentage of removed stores, which we attribute to second-order increase in write combining efficiency. Because squashed stores do not allocate a write buffer, there are more buffers available for combining non-silent stores. The percentage of removed stores is lower than the overall percentage of silent stores (and also the percentages of squashed stores presented previously) because we do not wait for store verifies to complete before committing stores (explained in Section 4.1). In further experiments not detailed here, we found that although we could decrease traffic by waiting for stores that hit in the L1 to finish verifying, because commit of some stores is stalled

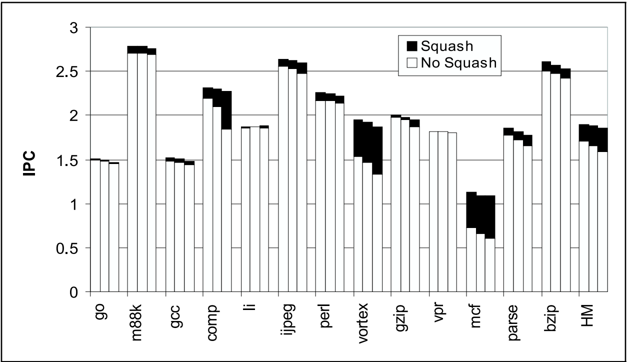

FIGURE 13. Performance comparison between most aggressive FSSS and no squashing for narrower L1 to L2 interfaces. The stacked bars indicate the performance obtained by squashing with 32B, 16B, and 8B wide L1 to L2 interfaces, respectively**.**

in this case, overall instruction throughput is lower. There is a potential performance vs. power consumption tradeoff here that could be exploited in power-aware designs.

In order to determine how effective this bandwidth reduction is on instruction throughput, we present Figure 13, which shows the performance across all benchmarks with varying width interfaces between L1 and L2, with and without FSSS squashing in its most aggressive form (spatial LSQ squashing with read port stealing). We keep the L1 cacheline size at 32B in all simulations, but illustrate the performance of 32B, 16B, and 8B wide interfaces between the L1 and L2. In each case, we are progressively lowering the physical bandwidth of the L1 to L2 interface because in the case of 16B and 8B widths more transactions across the interface are required for a cacheline transfer (two and four transactions for 16B and 8B, respectively). However, we change the write combining width to match the physical interface width so that flushing a write buffer takes only a single cycle.

If we compare FSSS with an interface width of 8B to no squashing with an interface width of 32B, we see that the effective bandwidth (as evidenced by IPC) of FSSS with the 75% lower physical bandwidth interface is more effective than the higher physical bandwidth interface without FSSS (the only exceptions to this are *go* and *gzip;* in these benchmarks, the percentages of silent stores are low, 27% and 16% respectively, leading us to expect less benefit from FSSS). In fact, as evidenced by the harmonic mean, the FSSS low physical bandwidth interface actually provides 9% greater effective bandwidth on average than the fastest physical interface we model. Therefore, we can potentially trade the implementation of FSSS for physical bandwidth. Of course, as also shown, FSSS still provides benefit no matter what physical bandwidth is available. Note that even though the actual reduction in physical bandwidth for the narrower interfaces (50% and 75% for 16B and 8B wide interfaces, respectively) is larger than the percent reductions shown in Figure 12, FSSS also decreases pressure on other hardware structures, such as write buffers, so the performance improvement is not solely due to the reduced L2 bandwidth.

We also observe in Figure 13 that the performance degradation from the widest (32B) to the narrowest (8B) interface is lower in the case of FSSS than for the baseline system with no squashing (40% lower according to the harmonic mean). This occurs because squashing is relatively more effective as the write-combining width narrows. With respect only to physical interface bandwidth, combining and squashing are equivalent. We can either save a transaction over the L1 to L2 interface by combining with a previous store or by squashing the store. However, there is some overlap between the methods (i.e. some stores that are squashed could also have been combined, and vice-versa, as can bee seen in Figure 12 in the difference between removed dynamic stores and reduced write through traffic). Of course, the combining width directly affects the number of stores that can be combined, but does not directly affect the number of squashed silent stores. Therefore, FSSS will capture some stores that can no longer be combined (but can still be squashed at any combining width), so the relative benefit of squashing increases as the combining width decreases along with the width of the L1 to L2 interface.

4.6 Results Discussion

Comparing the performance results for the three FSSS methods simulated for our machine model, we see that read port stealing and aggressive LSQ squashing exploiting both temporal and spatial locality in the LSQ provide nearly equivalent performance, with harmonic mean speedups of 10% and 11%, respectively. This occurs because both methods capture greater than 83% of all silent stores and close to 90% on average, so both methods are suitable for exploiting FSSS for IPC benefit. However, aggressive LSQ squashing reduces the number of store verifies issued to the memory system by 50%, on average (comparing the read port stealing percentages from Figure 8 and Figure 11). Therefore, in a machine model with relatively fewer memory ports, aggressive LSQ squashing may have greater benefit because of reduced port contention. Reducing data cache accesses may also reduce overall power consumption.

Temporal LSQ squashing by itself provides only modest speedup in these benchmarks, less than 1%, because of the low percentage of silent stores captured (31% on average) and the corresponding 9% average reduction in total committed dynamic stores. Therefore, while temporal LSQ squashing has the benefit of never stealing a cache read port, in our machine, solely implementing this mechanism does not seem worthwhile.

5 .0 Conclusion

We make four contributions in this paper. First, we explain why standard store verifies, as initially proposed in [14] have some undesirable characteristics, and introduce the concept of free silent store squashing, which uses existing resources as-is or with slight modification to squash a significant portion of all silent stores. Second, we explain three ways in which we can implement free silent store squashing: i) using pre-existing ECC logic that is present, and will become more prevalent, in current and future microarchitectures; ii) using idle read port stealing to perform store verifies; iii) enhancing an existing load/store queue to exploit temporal and spatial locality for store squashing. We show that in current and next generation microarchitectures that opportunities exist to exploit these mechanisms using real examples from the Alpha 21264 [9], IBM RS64-III [4], and IBM Power4 [13]. We further show that substantial performance benefit can be obtained by exploiting free silent store squashing and that two of the three mechanisms--read port stealing and aggressive LSQ squashing--capture a significant portion of silent stores detected by the standard store verify mechanism (greater than 83% and 89% on average). They do so at a substantially reduced cost for performance benefits averaging 10% and 11% across the SPECINT95 and a subset of the SPECINT-2000 benchmarks for each method respectively. Third, we illustrate that “effective” bandwidth between the L1 and L2 data caches can be increased by free silent store squashing, and indicate that a substantially lower bandwidth physical interface between the two caches provides the same or better performance when performing free silent store squashing. Finally, we provide additional characterizing information about silent stores by showing that, in most benchmarks, greater than 40% of all silent stores can be squashed by simply examining data that are temporally or spatially local to data already existing in the LSQ. We also illustrate that, on average, 31% of silent stores are detected with temporal locality, and an additional 19% are detected with spatial locality, in the LSQ. The work reiterates that silent stores can be exploited for performance improvement and illustrates that taking advantage of the majority of silent stores is relatively easy given current microarchitectures, lowering the barriers to exploiting them in the future.

6 .0 Acknowledgements

This work was supported by generous equipment donations from IBM and Intel. Financial support was provided through a graduate student fellowship from Intel and also funds from the University of Wisconsin Graduate School. We would also like to thank the anonymous reviewers for their many helpful comments.

References

[1] G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of Silent Stores. To appear in *International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[2] R. E. Blahut. *Theory and Practice of Error Control Codes.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[3] J. Borkenhagen. Personal Communication. IBM Server Development. Rochester, MN, June 2000.

[4] J. Borkenhagen and S. Storino. *5th Generation 64-bit PowerPC-Compatible Commercial Processor Design.* IBM Whitepaper, http://www.rs6000.ibm.com, 1999.

[5] D. C. Burger and T. M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] B. Calder, G. Reinman, and D. Tullsen. Selective Value Prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA’99)*, volume 27, 2 of *Computer Architecture News*, pages 64–75, New York, N.Y., May 1–5 1999. ACM Press.

[7] B. Calder, P. Feller, and A. Eustace. Value Profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[8] R. P. Colwell and R. Steck. A 0.6um BiCMOS Processor with Dynamic Execution. In *Proceedings of ISSCC.* 1995.

[9] Compaq Computer Corp. *Alpha 21264 Hardware Reference Manual DS-0027A-TE.* http://www1.support.compaq.com/alpha-tools/documentation/current/chip-docs.html. February, 2000.

[10]J. Gonzalez and A. Gonzalez. Control-flow Speculation Through Value Prediction for Superscalar Processors. In *Proceedings of PACT-99*, October 1999.

[11]IBM Corporation. *Fault Tolerance Decision in DRAM Applications.* Application Note, http://www.chips.ibm.com/products/memory/fault/fault.html. July, 1997.

[12]Norman P. Jouppi. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* , 1993

[13]J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum.* October 1999.

[14]K. M. Lepak and M. H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[15]M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[16]T. May and M. Woods. Alpha-particle-induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electronic Devices*, 26(2), 1979.

[17]A. Mendelson and F. Gabbay. Speculative Execution Based on Value Prediction. Technical report, Technion, 1997. (http://www-ee.technion.ac.il/.

[18]C. Molina, A. Gonzalez, and J. Tubella. Reducing Memory Traffic via Redundant Store Instructions. In *Proc. of Int. Conf. on High Perf. Computing and Networking*, pages 1246–1249, April 1999.

[19]A. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, December 1998.

[20]T.R.N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[21]E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th Fault-Tolerant Computing Symposium*, June 1999.

[22]P. Rubinfeld. Managing Problems at High Speed. *IEEE Computer*, pages 47-48, January 1998.

[23]Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.

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

[25]J. Ziegler et al. IBM Experiments in Soft Fails in Computer Electronics. *IBM Journal of Research and Development*, 40(1):3-18, January 1996.