On the Value Locality of Store Instructions

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

Value locality, a recently discovered program attribute that describes the likelihood of the recurrence of previously-seen program values, has been studied enthusiastically in the recent published literature. Much of the energy has focused on refining the initial efforts at predicting load instruction outcomes, with the balance of the effort examining the value locality of either all register-writing instructions, or a focused subset of them. Surprisingly, there has been very little published characterization of or effort to exploit the value locality of data words stored to memory by computer programs. This paper presents such a characterization, proposes both memory-centric (based on message passing) and producer-centric (based on program structure) prediction mechanisms for stored data values, introduces the concept of silent stores and new definitions of multiprocessor false sharing based on these observations, and suggests new techniques for aligning cache coherence protocols and microarchitectural store handling techniques to exploit the value locality of stores. We find that realistic implementations of these techniques can significantly reduce multiprocessor data bus traffic and are more effective at reducing address bus traffic than the addition of Exclusive state to a MSI coherence protocol. We also show that squashing of silent stores can provide uniprocessor speedups greater than the addition of store-to-load forwarding.

1 .0 Introduction

A flurry of recent publications have examined the program attribute of value locality. Value locality describes the likelihood of recurrence of previously-seen program values within computer storage locations. Most of this work has focused on exploiting this property to accelerate the processing of instructions within a superscalar processor, with the goal of exposing greater instruction-level parallelism and improving instruction throughput. In fact, value locality makes it possible to exceed the classical dataflow limit, which is defined as the program performance obtained when machine instructions execute as soon as their operands are available. Indeed, value locality allows instructions to execute before their operands are available by enabling the prediction of the operand values before they are computed. Value prediction has been proposed and examined for the purpose of reducing average memory latency by predicting load instruction outcomes [10], improving throughput of all register-writing instructions by predicting the outcomes of all such instructions [11], as well as focusing prediction on only those computations that help resolve mispredicted branches [7] or occur on some other critical path [2]. All of these proposed uses of value prediction share the common goal of accelerating the processing of instructions within a superscalar processor.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA 00 Vancouver, British Columbia Canada
Copyright (c) 2000 ACM 1-58113-287-5/00/06-182 $5.00

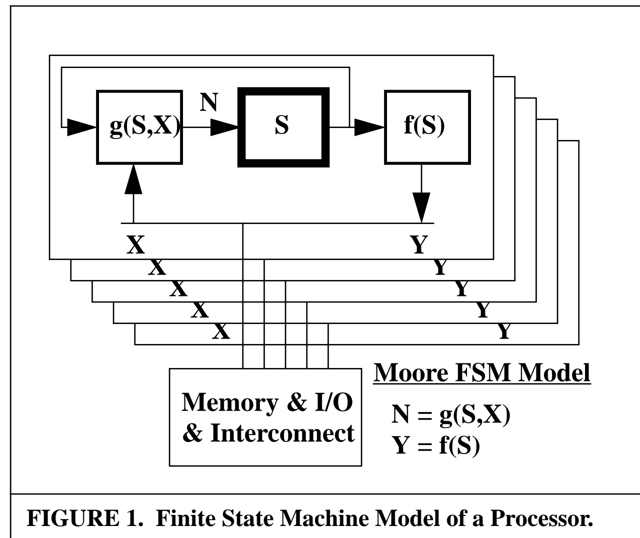

FIGURE 1. Finite State Machine Model of a Processor.

While important and interesting in its own right, this approach to exploiting value locality is in some ways misguided, as it focuses on *process* rather than *outcome*. In other words, it places emphasis on efficient and rapid processing of instructions, which is of course the *means* of modern computing, rather than on timely and correct generation of the result of the computation, which is after all the real *end* or goal of computing. To better understand the distinction, we revisit the useful abstraction of the finite state machine (FSM) model of instruction set processing [17], as illustrated in Figure 1].

When reduced to its most basic form, a modern microprocessor can be viewed as nothing more than a simple Moore finite state machine. The microprocessor has some initial state S, consumes a sequence of inputs X by sequencing through a set of finite states determined by the next state function g(S,X), and generates a sequence of outputs Y defined by the output function f(S). The inputs X are embodied as bit patterns retrieved by instruction fetches and loads from memory and/or I/O devices, and the out-

Benchmark	Description	Silent Stores (PPC/SS)	PSSVL (LastVal/ Stride)	MPSVL (LastVal/ Stride)
go	SPEC95 game	38%/27%	30%/32%	36%/39%
m88ksim	SPEC95 simulator	68%/62%	56%/60%	65%/70%
gcc	SPEC95 compiler	53%/46%	37%/39%	49%/52%
compress	SPEC95 compression	42%/39%	35%/65%	16%/47%
li	SPEC95 lisp interpreter	34%/20%	32%/39%	34%/43%
ijpeg	SPEC95 image compression	43%/33%	52%/61%	46%/50%
perl	SPEC95 language interpreter	49%/36%	39%/41%	42%/44%
vortex	SPEC95 object database	64%/55%	71%/72%	57%/58%
oltp	4proc in-memory TPC-B w/RDBMS	56%	52%/56%	45%/51%
barnes	4proc SPLASH-2 N-body simulation	40%	27%/27%	38%/43%
ocean	4proc SPLASH-2 Ocean simulation	41%	41%/43%	36%/39%

TABLE 1. Store Value Locality Measurements for Various Benchmarks.

puts Y are embodied as fetch and load addresses as well as store address and value tuples presented to memory and/or I/O devices. In its simplest form, the architected register state of an instruction set processor corresponds to the FSM state S, the next state function N=g(S,X) corresponds to the next instruction address computation as well as any register-to-register ALU semantics specified by the current instruction, and the output function f(S) relays fetch, load, and store addresses and store values to the memory interface.

Within the FSM model, the *end* or goal of processor microarchitecture can be described as transforming an input sequence X as quickly and efficiently as possible into its corresponding output sequence Y, particularly into the final state determined by aggregate effect of that output sequence. With this goal in mind, it becomes clear that recently proposed hardware optimizations that exploit value locality are focused exclusively on accelerating the next state function N=g(S,X), i.e. accelerating the processing of instructions, which is merely the *means*, and not the *end*, of a processor's microarchitecture. We contend that focus should be placed on the entire path from input sequence X to output Y, especially on the neglected output function Y=f(S), rather than exclusively on the N=g(S,X) next state function. The stated purpose of our work, then, is to examine opportunities for exploiting value locality to accelerate the output function Y=f(S). Specifically, we introduce the notion of silent stores and examine the value locality of stores from both memory-centric (data-address based) and producer-centric (store instruction-based) viewpoints (Section 2); examine the implications of store value locality on microarchitectural techniques for handling store instructions (Section 3); introduce and quantify two new definitions of multiprocessor true sharing [4] based on store value locality (Section 4); and explore the potential for reducing multiprocessor data and address traffic (Section 5). Our initial results in all of these areas are promising, and should encourage and motivate further study of this neglected area of computer architecture.

2 .0 The Value Locality of Stores

In order to better understand the value locality of store data values, we measured the store value locality program property from two different perspectives which we call *program structure store value locality* (PSSVL) and *message-passing store value locality* (MPSVL). Program structure store value locality measures the locality of values written by a particular static store, and is analogous to the value locality reported for loads and other register-writing instructions in prior work (e.g. [10]). In contrast, message-passing store value locality measures the locality of values written to a particular address in data memory (i.e. messages passed through memory). Most of the prior work on value locality has focused on program structure-based prediction, since there is very little to be gained by predicting load values once their addresses are known (it is usually just as fast and nonspeculative to access cache memory directly, with the exception of data items that miss in the cache). Two counterexamples that study message-passing value locality are Gabbay's work [12], which studies the value locality of instructions based on destination register identifier, and Calder's work [3], which studies load value predictability based on memory address.

We examined the eight uniprocessor and three multiprocessor benchmarks described in Table 1. The uniprocessor programs were run under two instruction sets (PowerPC and Simplescalar) using associated compilers, run-time environments, and simulators. The multiprocessor benchmarks were run only under PowerPC, on a 4-processor shared-memory multiprocessor simulated under the SimOS-PPC full system simulator [9]. The SPLASH-2 benchmarks [1] are widely used and understood, and are included for continuity with previous work. The *oltp* benchmark employs twelve concurrent streams of TPC-B transactions [16] operating on an in-memory database stored in a commercial relational database management system (RDBMS).

Table 1 reports the program structure store value locality (PSSVL) and message-passing store value locality (MPSVL) for each benchmark, as well as the fraction of stores executed by each program that are effectively silent. A *silent store* is defined as one that does not change the system state. In other words, the value being written by the store matches the exact value already stored at that memory location. This program characteristic can be viewed as the upper limit for message-passing store value locality that relies on a tagged last value predictor. More complex predictors, such as the stride predictor we use, are able to exceed this limit. The PSSVL and MPSVL results shown in Table 1 are derived with a large stride predictor table with 64K direct-mapped entries; results for smaller predictor tables are shown in Figure 2and Figure 3.

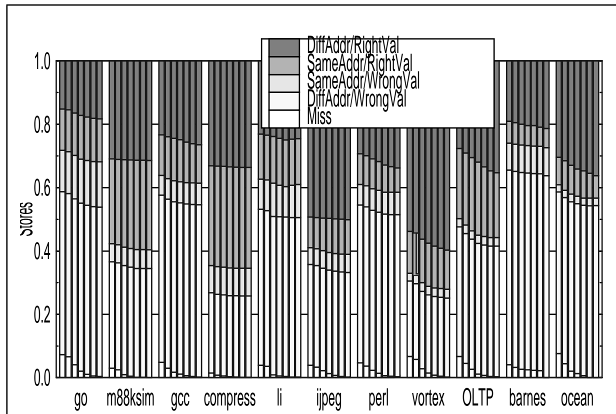

FIGURE 2. Program Structure Store Value Locality. Dynamic store breakdown is shown for 1K, 2K, 4K, 8K, 16K, 32K, and 64K entry predictor tables.

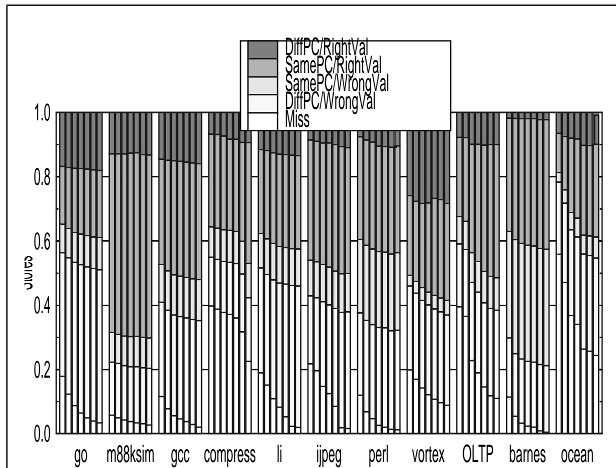

FIGURE 3. Message-passing Store Value Locality. Dynamic store breakdown is shown for 1K, 2K, 4K, 8K, 16K, 32K, and 64K entry predictor tables.

Figure 2 shows PSSVL for various predictor table sizes. For each benchmark, from left to right, the stacked bars account for store value locality for predictor tables of size 1K, 2K, 4K, 8K, 16K, 32K, and 64K entries. The prediction tables are indexed and tagged with the store program counter value, and are capable of capturing last value locality as well as unit stride sequences [8]. Each dynamic store instance is counted in one of five categories, which correspond to the five elements of the stacked bar: Misses, in which case the store does not find a matching entry in the table; DiffAddr/WrongVal, in which case the store is to an address that differs from the address of the previous instance of that static store and the value written does not match the table's prediction; SameAddr/WrongVal, in which case the store is to the same address as the previous instance but writes a value that does not match the prediction; SameAddr/RightVal, in which case the store is to the same address and writes the predicted value; and DiffAddr/RightVal, in which case the store is to a different address but writes the predicted value. The aggregate predictability of store values ranges from a low of 27% for *barnes* to a high of 72% for *vortex*. Since there is a significant population of stores in each category, no clear correlation exists between

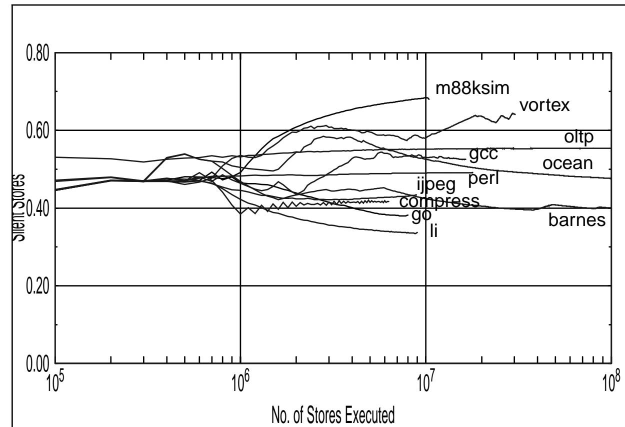

FIGURE 4. Silent Stores vs. Time. Cumulative fraction of silent stores throughout program execution. Note that the x-axis is on a log scale.

address variability and store value variability. The larger table sizes are important only for *oltp*, which is known to have a large instruction working set [11].

Figure 3 shows the message-passing store value locality for the same predictor table sizes (1K, 2K, 4K, 8K, 16K, 32K, and 64K entries). In this case, the tables are indexed with the physical data address being written by the store, and each dynamic store instance is counted in one of five categories: Miss, as above; DiffPC/WrongVal, in which case a static store that is not the last one to write to this address (a different static store) writes a value that does not match the prediction; SamePC/WrongVal, in which case the same static store writes a value that does not match the prediction; SamePC/RightVal, in which case the same static store writes the predicted value; and DiffPC/RightVal, in which case a different static store writes the predicted value. Here, the aggregate predictability ranges from a low of 39% for *go* to a high of 70% for *m88ksim*. Again, there is no obvious correlation between the identity of the writer (static store) and store value variability, since significant populations of stores exist in each category. Here, the table size is more important for most of the benchmarks as it must capture the data working set--which is larger than the instruction working set for most of these programs--in order to be effective.

From the data presented so far, there is no clear answer as to whether program-structure or message-passing store value locality is the better choice; there are situations in which either will outperform the other. We revisit this question in Section 5, where we explore the effects of store value locality on multiprocessor bus traffic.

In order to examine variation due to program phase, we also measured the time domain frequency of occurrence of silent stores for the PowerPC architecture. This data is plotted in Figure 4. With the exception of *oltp*, which is a snapshot of steady-state execution, all the benchmarks have nearly identical store value locality in the first several hundred thousand stores. We attribute this to the program loader (this data was collected with a full-system simulator that includes all portions of program execution, including program load time). Beyond the initial loader phase, the benchmarks demonstrate noticeable variation in their behavior. The near-monotonic decrease in *li* indicates that the actual work *li* is doing beyond the load time has much less store value locality (as measured by silent stores) than the program loader does. We

attribute the differences in store value locality between The PowerPC and Simplescalar instruction sets (shown in Table 1), to variations in program model, runtime environment, and compilation technology. For example, efficient register allocation can cause a large difference in the frequency and character of store instructions. For lack of an alternative, we used the gcc compiler for Simplescalar; The PowerPC benchmarks were compiled with the IBM AIX optimizing C compiler. An additional explanation for the variations observed is that the PowerPC statistics were collected from a trace, and hence include only retired instructions, while the Simplescalar statistics include speculative instructions on wrong branch paths.

3 .0 Improving Store Handling Microarchitecture

Having introduced the concept of silent stores, can we find a way to exploit them at the microarchitectural level? In this section, we outline some obvious microarchitectural structures that may be enhanced and explore initial experimental results. For the entirety of the microarchitecture discussion, the term "silent store" refers only to update-silent stores--those which seem intuitively easiest to exploit with little microarchitectural complication. We will explore possible benefits of stochastically silent stores in Section 5. For the microarchitectural discussion, we consider only a weakly consistent memory model, and assume a uniprocessor system, to allow aggressive removal of silent stores.

3.1 Load/Store Queue

In many modern microprocessors, memory hierarchy performance is a bottleneck, even with the latency tolerance that extremely out-of-order processors provide. Hence, improving performance of the memory access path is the subject of much research. Enhancements to this path within the processor core itself, including store to load forwarding, hoisting of loads past previous stores, non-blocking caches, and deep load/store buffers have been used to improve system performance [14, 8]. However, these performance enhancements can lead to increased cycle time due to the size and versatility of content addressable memory system required to maintain program correctness due to address ordering requirements in the architecture. We assert that silent store squashing may allow a designer to obtain greater performance from existing structures, or a reduction in size or complexity of this system (which we herein refer to as the load/store queue, or LSQ) because of the relative benefit of squashing.

Squashing should provide performance benefit in this system because squashed silent stores can be physically removed early from the LSQ, effectively making the LSQ larger. Early removal of entries from the LSQ (and possibly the instruction control unit) can also decrease pressure on the commit logic. It also has the obvious effect of easing store unit pressure.

However, squashing may negatively impact the LSQ, due to the necessity of verifying each store before we can squash it (the simple mechanism we employ in our preliminary experiments is covered in Section 3.4).

3.2 Memory Hierarchy

Silent store squashing may benefit a uniprocessor system memory hierarchy by reducing the number of dirty cache lines and hence the number of writebacks. The removal of writebacks can increase system performance and lower pressure on writeback buffers.

Squashing may also allow a designer to have fewer store ports into the memory system (in our current implementation, we trade store ports for load ports, but other implementations may be possible) for a desired amount of store bandwidth. This has the potential to decrease circuit complexity, as in general, load bandwidth is easier to obtain than store bandwidth by resource duplication or other methods [8].

3.3 Machine model

To determine the performance effect, if any, of an initial implementation of squashing, we used an execution driven simulator of the Simplescalar architecture. In order to model the increasing demands on a memory subsystem, we used a very 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; 2 integer multipliers. The cache configurations are 64KB split I/D L1 and 1MB unified L2 with pipelined access and latencies of 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 way associative with line sizes of 32 and 64 bytes, respectively. When store to load forwarding is enabled it has a latency of 2 clocks to match the L1 cache latency.

The memory access configuration is a four-load-wide version of the two-wide DEC Alpha 21164 [5]. We allow up to 4 loads or 1 store to issue per cycle (loads and stores are mutually exclusive, with no address restrictions on parallel loads). All stores wait for their associated store verify (described in Section 3.4) before committing into the memory subsystem subject to latency constraints (i.e the level in the memory hierarchy where the store hits). Waiting for store verifies allows maximal reduction of writebacks, but also can hurt instruction throughput because the store is not allowed to leave the instruction window until the store verify completes (as opposed to letting the store complete immediately and just buffering the write appropriately.) It is possible to do this squashing at some other level in the memory hierarchy (or have special hardware outside of the instruction window to do the verifies) but this was not implemented in our simulator. Therefore, the IPC results when verifying cache missing stores are slightly pessimistic and do not reflect the best possible performance.

It is also important to note that our simulator does not model contention for writeback buffers (or rather, it supports unlimited writeback buffers), hence we do not gain any IPC from removing the writebacks. However, in real systems the number of writeback buffers is finite so there is some IPC benefit which is not reflected in our results.

3.4 Silent Store Removal Mechanisms

We implement two store squashing mechanisms to evaluate a realistic implementation and also a theoretical limit for our machine. We refer to these as *realistic* and *perfect*.

Realistic Method: Each store is converted into a store verify, which is effectively three operations--a load, a comparison, and the subsequent store (if the store is non-silent). The store verify is initiated after the effective address has been computed in the execution engine and all previous store addresses are known, so that possible store address unknown dependencies need not be considered [14]. When the data returns from the memory subsystem, it is compared to the new value to be written. If the data values are equal, the store is update-silent and it is removed from the

Benchmark	Number of writebacks (% reduction relative to baseline case)				
	Baseline	Realistic/L1	Realistic/L1+L2	Realistic/L1+L2+M	Perfect
go	34175	33656 (1.5%)	30611 (10.4%)	29308 (14.2%)	29354 (14.1%)
m88ksim	23877	23863 (0.0%)	23768 (0.4%)	10134 (57.6%)	10133 (57.6%)
gcc	68833	61240 (11.0%)	56934 (17.3%)	53693 (21.6%)	53628 (22.1%)
compress	370569	353172 (4.7%)	152953 (58.7%)	147557 (60.2%)	147582 (60.2%)
li	1896	1953 (-3.0%)	1913 (-0.9%)	1830 (3.5%)	1852 (2.3%)
ijpeg	51853	51848 (0.0%)	49702 (4.1%)	44548 (14.1%)	44337 (14.5%)
perl	8723	8517 (2.4%)	8374 (4.0%)	7990 (8.4%)	7949 (8.9%)
vortex	377852	375470 (0.6%)	312052 (17.4%)	71742 (81.0%)	71712 (81.0%)

TABLE 2. UFS Store Squashing Effect on L1 Writebacks. The "realistic" columns show results of squashing store hits to different levels of memory ("L1" indicates we verify only L1 store hits, "L1+L2" indicates we verify L1 and L2 store hits, etc.)**.**

LSQ and the store entry in the RUU is flagged to indicate the store is silent. When the store reaches commit, if it is not flagged as silent, the store port is obtained and the write occurs to the memory system as it would normally. If the store is silent, the store retires with no memory access and no side effects, except that it consumes a commit slot.

Perfect Method: Store squashing occurs in the same manner as above, except it is known by some mechanism that the store is silent and hence the verification is performed only for the known silent stores. Non-silent stores execute as normal with no store verify. This method is meant to illustrate the performance obtained with a perfect prediction mechanism for update-silent stores. The store verify is still carried out for the predicted silent stores because in reality, no confidence mechanism can ever be perfect--hence validation of the prediction must still be done.

3.5 Results

3.5.1 Writeback Reduction

In Table 2, we show the writeback reduction obtained by squashing and allowing store verifies to complete to different levels of memory hierarchy for both realistic and perfect squashing.

We see from Table 2 that squashing can yield a significant reduction in writebacks depending on the benchmark and the memory hierarchy level to which we allow verifies. We see a range in reduction from 81% (in *vortex*) to 0% (or small negative values in *li* which we attribute to second-order LRU policy effects). The average for all benchmarks is a 33% reduction.

We also see that squashing in L1 cache only doesn't significantly reduce the number of writebacks (the maximum reduction is 11% in *gcc*, all others are less than 5%). This indicates that lines in the L1 cache are sufficiently active such that they are stored to at least once non-silently, necessitating a writeback anyway.

However, there is a substantial reduction in writebacks when squashing to other levels in the memory hierarchy--all reductions are greater than 14% when verifying into memory (with the exception of *li* and *perl* which have very few writebacks to start with.) This can be partially explained by the observation that the probability of a store miss creating a dirty line in the L1 cache if we don't verify it is 100%, but if we verify the miss, the probability is less than 100%.

Finally, it is worthwhile to note that the realistic results with L1, L2, and memory squashing are essentially the same as the perfect results, with the minor differences attributable to second order LRU policy effects, agreeing with the intuition that performing the store verify for non-silent stores (which is done in the realistic case) should not affect the number of writebacks. Some similar results for writeback reduction were presented in a study by Molina [13].

3.5.2 Instruction Throughput

We now turn our attention to the effect of squashing on instruction throughput and compare it to store to load forwarding as a performance enhancing mechanism. Note that these results only perform store verifies on L1 cache hits so that the LSQ and RUU do not get backed up waiting for store verifies. As mentioned in Section 3.4, ideally a machine would squash misses at some other layer of the memory hierarchy (to obtain both the writeback reduction and instruction throughput shown), yielding a potential reduction in processing core complexity (to handle fewer stores) and also the writeback reduction. However, it should be noted that the results in Table 2 and Figure 5 can be considered orthogonal in this respect since squashing at some other level in the memory hierarchy would not affect the processing core substantially.

In Figure 5, the three left-hand bars in each group represent performance with no store forwarding (SF) and the right-hand group represent performance with store forwarding. Within each group, the bars represent (from left to right) baseline performance, realistic squashing performance, and perfect squashing performance. The subdivisions within bars indicate different LSQ sizes. We can see in Figure 5 that in no case does realistic silent store squashing decrease performance in our processor model, even with the added store verify operations. More interestingly, without SF in *m88ksim* we see an IPC of 3.32 with an LSQ size of 16 and squashing vs. an IPC of 3.12 with LSQ size 32 and no squashing--better performance for half the LSQ size. (This can be attributed to the high percentage of silent stores in this benchmark: 62%.) In the other benchmarks, the effect of squashing is less dramatic, but realistic squashing generally performs better than SF for equivalent LSQ sizes (except for *go*, where the IPC is fairly constant and not memory limited, *compress* which obtains 8% speedup from SF alone with an LSQ size of 16, and *li*/*perl* for LSQ size of 32). Perfect squashing always outperforms SF (except in *compress* and *go* for the reasons above) because of the removal of unnecessary store verify operations.

Over the range of benchmarks, we see speedups of 6.3% and 6.9% for realistic and perfect squashing with SF over the baseline with SF (5.1% and 6.9% respectively for each without SF over

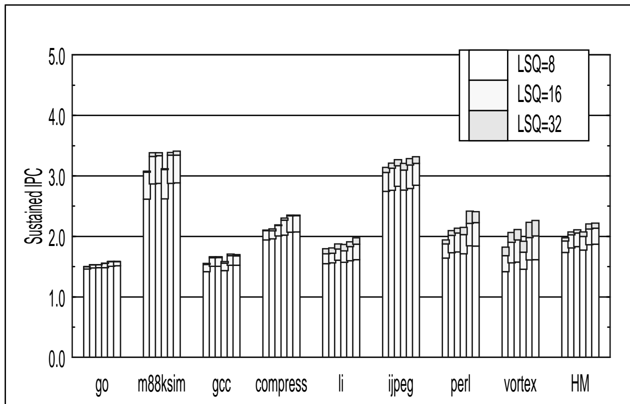

FIGURE 5. SPECINT Performance with Store Squashing. For each benchmark, the left three bars show performance without store forwarding, while the right three bars show performance with store forwarding. The three bars in each group show baseline IPC, IPC with realistic store squashing, and IPC with perfect store squashing.

the baseline without SF.) In the benchmarks with the highest percentage of silent stores (*vortex* and *m88ksim*) we see speedups of 15% and 8.9%, respectively--definitely non-trivial.

Of course, as evidenced in Figure 5, SF always provides some additional benefit along with squashing because not **all** stores are silent.

If we compare perfect squashing with realistic squashing, the largest difference appears in *compress* with an improvement of 3.3%. In general, the difference is small (1.8% over all the benchmarks) implying that this machine model would not benefit much from a good silence prediction mechanism.

As a separate issue, it is also interesting to note how little effect SF has in our processor model. Over the range of benchmarks, the speedup gained using SF is only 3.8% (contrasted with the 5.1% and 6.9% of realistic and perfect squashing with SF, respectively, shown earlier). This result supports our assertion that store squashing supplies more performance than store forwarding, and is not unexpected in light of the results of Moshovos [14], which explore the temporal locality of memory operations in the context of store to load forwarding. In light of these results, on load-store RISC architectures with sufficient general registers (32 in our machine), given equal hardware costs, silent store squashing provides greater benefit than store forwarding.

4 .0 New Definitions of False Sharing

We shift our focus to multiprocessor applications of store value locality by introducing new definitions of false sharing. Prior work in defining false sharing focuses on the address of potentially shared data. All of the previous definitions rely on tracking invalidates to specific addresses or words in the same block. However, no attempt is made to determine when the invalidation of a block is unnecessary because the *value* stored in the line does not change. The fact that many stores are silent, and even more are stochastically silent requires new definitions of true and false sharing.

4.1 Address-based Definitions of Sharing

In order to describe how our definitions differ from the previous, a review of the prior work is necessary. Throughout the discussion, we imagine sharing as defined in a multiprocessor system with an update based protocol, and for ease of discussion, a sequentially consistent machine with infinite sized caches is implied (so that capacity and conflict misses can be ignored.) All of the definitions are similar in their recognition of “cold” misses (CM), true sharing misses (TSM), and false sharing misses (FSM). We focus our discussion on the definition of Dubois [7] as it provides the most accurate definition of address-based sharing. For a review of other definitions developed prior to Dubois, refer to [15] and [6].

Dubois’ Definition:

Cold Miss: The first miss to a given block by a processor.

Essential Miss: A cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) a value defined by another processor since the last essential miss to that block, it is an essential miss.

Pure True Sharing miss (PTS): An essential miss that is not cold.

Pure False Sharing miss (PFS): A non-essential miss.

Essential misses constitute all misses which bring in a truly shared word either directly, or as a side effect (for example, when a truly shared value is brought in as the non-critical word in a cache refill). Note that the use of the word “value” in the above definition means value in the invalidation sense only, i.e, a store instruction has occurred to that address. It is not implying anything about the data value at that address.

In general, Dubois contributed the insight that merely tracking the address that invalidates a cache block or only comparing the address that causes a miss to the immediately previous invalidating addresses of that block is not sufficient. To be more precise, we must examine all previous invalidations of a block and the side-effects of loading a cache line to be sure that PTS and PFS misses are not incorrectly counted.

4.2 Update-based False Sharing (UFS)

In our definition of update-based false sharing (UFS), we will keep the same definitions as Dubois with extensions covering the value locality of stores. Intuitively, we extend the definition of Essential Miss to exclude those stores which are silent, i.e, those that do not change the machine state because they are attempting to store the value that was previously available at that location in the system memory hierarchy. Rigorously, we propose the following, modified, definition of an essential miss (our changes are in *italics*):

Essential Miss: A cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) *an address which has had a different data value* defined by another processor since the last essential miss to that block, it is an essential miss.

While the wording of this definition is almost the same as the one proposed by Dubois, we have made a slight change to make clear that we are interested in the data value at a memory location. The other definitions remain accurate with no modification.

4.3 Stochastic False Sharing (SFS)

In light of the work of Lipasti [10,11] and others, we have seen that many data values are trivially predictable. We would also like to extend our definition of false sharing to cover data values that are trivially predictable with any well known method. It

seems intuitive that if we can define false sharing to compensate for the effect of silent stores that we could also define it in the presence of stochastically silent stores (values that are trivially predictable via some mechanism, the details of which are beyond the scope of this work). Of course, with value prediction we need a mechanism for verifying the prediction. Efficient mechanisms of communicating/verifying the prediction with the actual owner of the updated value are necessary. This will be the subject of future work and will not be covered here.

In value prediction, a distinction must also be made in how we're predicting a memory value. We can predict the data value based on effective address of the operation (as in the MPSVL case in Section 2) or on the PC of memory operation (as in the PSSVL case in Section 2) which can potentially have a different effective address. To completely enumerate these conditions, we define the following types of SFS:

Message-passing Stochastic False Sharing: (MSFS) is SFS based on the predicted data value located at the effective address generated by any instruction (multiple PCs could generate this EA). This terminology is used because data at the same EA can generally be thought of as being used for inter-process communication.

Program structure Stochastic False Sharing: (PSFS) is SFS based on the predicted data value of an instruction located at a specific PC (multiple data addresses could be targets of this prediction). This terminology is used because the value generated/consumed at a specific program location can generally be thought of as being a characteristic of the program structure.

Note that the definitions of MSFS and PSFS are not mutually exclusive. Formally, we extend the definition of an essential miss again to create the basic definition of stochastic false sharing (SFS) with the distinction pointed out above being implicit. We must also modify the definition of cold misses in the Dubois approach due to the possibility of statically predicting a value with no history (this modification is unnecessary for Update-based False Sharing).

Essential Miss: A stochastic cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) *an address which has had a new data value which is not trivially predictable* defined by another processor since the last essential miss to that block, it is an essential miss.

Stochastic Cold Miss: (SCM) A cold miss on a store which has a data value which is not trivially predictable.

In order to illustrate our new definitions, an example is given in Table 2. Eight word cache blocks are assumed. The numbers in parenthesis are the data values. The notation (x)+1 means a trivially predictable stride pattern in the data. We also assume, for the sake of stochastic cold misses, that the static prediction of a previously unaccessed data word is zero. This example is very similar to those used by Dubois and is fairly straightforward. Areas of particular interest include:

- T0: We have no cold miss because of the definition of a SCM (we would have predicted this store to be 0).
- T3: This miss is update-false because the stored value didn't change, hence the invalidate and subsequent miss were unnecessary.
- T6: The store instruction in Proc. 2 at PC 200 is program stochastically silent (because the last time a store at PC 200 executed it stored the value 10) and requires no invalidate under PSFS--hence this load is PSFS.

Time	PC	Proc. 1	Proc. 2	Dubois	Ours
T0	0	ST 0(0)	INV	PCM	
T1	100		LD 1(10)	PCM	CM
T2	200	INV	ST 1(10)		
T3	300	LD 1(10)		PTS	UFS
T4	400	LD 8(2)		PCM	CM
T5	200	INV	ST 8(10)	PCM	CM
T6	500	LD 8 (10)		TSM	PSFS
T7	600	ST 16(4)+1	INV	PCM	CM
T8	700		LD 16(5)	CTS	MSFS
T9	800	ST 24(4)		PCM	SCM

TABLE 3. Data Sharing Classification Example.

- T8: The load to address 16 is assumed to be trivially predictable and hence the load is MSFS (assume the processor had some history to predict this message value, even though there is no history in this example to detect the stride).
- T9: The store here is a Dubois CM and is storing a non-zero (not statically predicted) value, hence it is SCM.

4.4 UFS and SFS Results

In order to characterize the degree to which these new definitions of false sharing affect true and false sharing in multiprocessor systems, we implement the measurement algorithm of Dubois [7] and exercise it with our multiprocessor benchmarks under six different scenarios:

- The baseline scenario corresponds to the Dubois definition of false sharing and treats stores just as Dubois' mechanism [7], measuring the relative number of cold misses, false sharing misses, and true sharing misses during each benchmark's execution.
- The second scenario corresponds to our definition of update-based false sharing (UFS). It implements *store squashing*, which effectively converts silent stores into loads. A realistic implementation of store squashing is described in greater detail in Section 3; suffice it to say that from a multiprocessor cache perspective, a squashed silent store requires neither exclusive ownership of the cache line (as in an invalidation-based cache protocol) nor remote propagation of the updated store value (as in an update-based coherence protocol), since the value being stored has not in fact changed. This scenario is consistent with the results in Section 3 in that only stores that hit in the data cache are squashed.
- The third scenario (UFS-P) measures the potential of UFS with perfect knowledge of store silence by squashing all stores that are silent, whether or not they hit in the data cache. This allows us to avoid sending a read (for the store verify) followed by an upgrade (S->M) for non-silent stores, and sending instead a read-with-intent-to-modify.
- The fourth scenario corresponds to our definition of message-passing stochastic false sharing (MSFS), in which stores that write values that are correctly predicted by an MPSVL-based predictor are eliminated from the cache hierarchy. We use a 4K entry stride predictor identical to that modeled for Figure 3.
- The fifth scenario corresponds to our definition of program structure stochastic false sharing (PSFS), in which stores that write values that are correctly predicted by a PSSVL-based predictor are eliminated from the cache hierarchy (i.e. they are observed as neither store nor load references). Here we also use a 4K entry stride predictor.

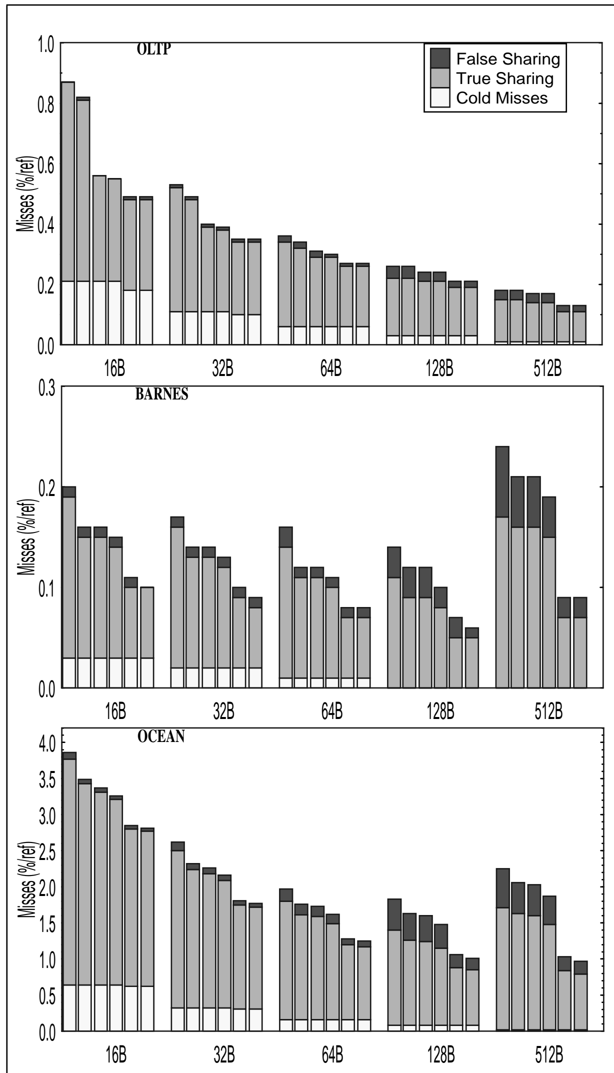

FIGURE 6. Multiprocessor sharing. Left to right, the stacked bars show cold, true sharing, and false sharing misses of the baseline, UFS, UFS-P, MSFS, PSFS, and MSFS+PSFS scenarios.

- The final scenario (M/PSFS) is an optimistic combination of MSFS and PSFS, in which stores that write values that are correctly predicted by either the MPSVL predictor of scenario three, or the PSSVL predictor of scenario four, are eliminated from the cache hierarchy. We assume an ideal mechanism for selecting the correct predictor in the case where only one produces the right prediction.

The final three scenarios correspond to our earlier definitions of stochastic false sharing, and are included to demonstrate the potential of store value locality for reducing multiprocessor bus traffic, as well as to provide some guidance for future research in this area. We do not describe an exact hardware mechanism for exploiting this type of locality in a multiprocessor system. The exact design of such a mechanism is beyond the scope of this initial paper, and is left instead to future work.

We measure true and false sharing for each of these six scenarios for various line sizes; our results for line sizes of 16B, 32B, 64B, 128B, and 512B are plotted in Figure 6. For *oltp* we observe measurable reductions in true and false sharing for UFS. For UFS-P and the stochastic sharing cases, the reductions (including some reduction in cold misses) are more dramatic. For *barnes* and *ocean*, the trends are the same, although more pronounced, since even simple UFS provides considerable reductions in overall miss rate due to a combination of reduced false sharing and reduced true sharing. For *oltp*, squashing silent stores that miss the cache (UFS-P) is very important for reducing the miss rate. This indicates that most of the shared data is written before it is read. This is less true for *barnes* and *ocean*, indicating that update-silent shared data (or at least spatially local data in the same line) are read by a processor before they are written, resulting in a silent store hit that canbe squashed.

5 .0 Reducing Multiprocessor Data and Address Traffic

In order to evaluate potential reduction in multiprocessor data and address traffic achievable through exploiting store value locality, we model a multiprocessor cache that implements the standard MESI (Modified, Exclusive, Shared, Invalid) coherence protocol [14]. Briefly, this protocol requires a processor to acquire exclusive ownership (M or E state) of a cache line before writing to it. Exclusive ownership is acquired through an invalidate mechanism that removes the line from other caches in the system. This protocol is widely used in modern shared-memory multiprocessors.

We exercise our cache model with the six scenarios described in Section 4: baseline, UFS store squashing, UFS-P, MSFS, PSFS, and M/PSFS. We present data for a 1MB 4-way set associative data cache with 16B, 32B, 64B, 128B, and 512B lines. We also collected data for smaller and larger caches, but restrict our presentation to the 1MB case, which reflects the general trends seen for other sizes as well. Figure 7 plots the miss rates for these cache configurations for each of our benchmarks. Misses are classified as cold, true sharing, false sharing, and capacity/conflict according to the method described in Section 4 [7].

Once again, we find measurable reductions in miss rates even with the simple UFS scenario, particularly for smaller lines. However, an interesting phenomenon occurs for *oltp*: as the sharing misses decrease due to UFS store squashing, conflict misses increase, holding the overall miss rate nearly steady. This is due to the increased working set brought about by fewer invalidates. Without the available invalidated lines to fill, the LRU replacement policy makes less than optimal replacement decisions. This suggests a need for a better replacement policy or perhaps greater associativity, or simply a larger cache. Dramatic miss rate reductions do not occur until program structure-based store elimination is applied. PSFS has a clear miss rate reduction advantage over MPFS, even though the two have comparable prediction accuracy (see Section 2). Intuitively, this agrees with the results presented by Kaxiras [15], which argue for program structure based predictors for identifying multiprocessor data sharing patterns.

The total data bus traffic is reduced by more than the ratios indicated by the miss rates plotted in Figure 7, since the frequency of writebacks of dirty lines is also reduced. With UFS-P store squashing, we observed 5%-82% reductions (depending on line size) in the writeback rates for *oltp*, 16%-17% reductions for *ocean*, and 5%-16% for *barnes*. For the most aggressive SFS case (scenario 5), we observed writeback rate reductions of 8%-85%, 25%-26%, and 16%-29% for *oltp, ocean,* and *barnes*, respectively.

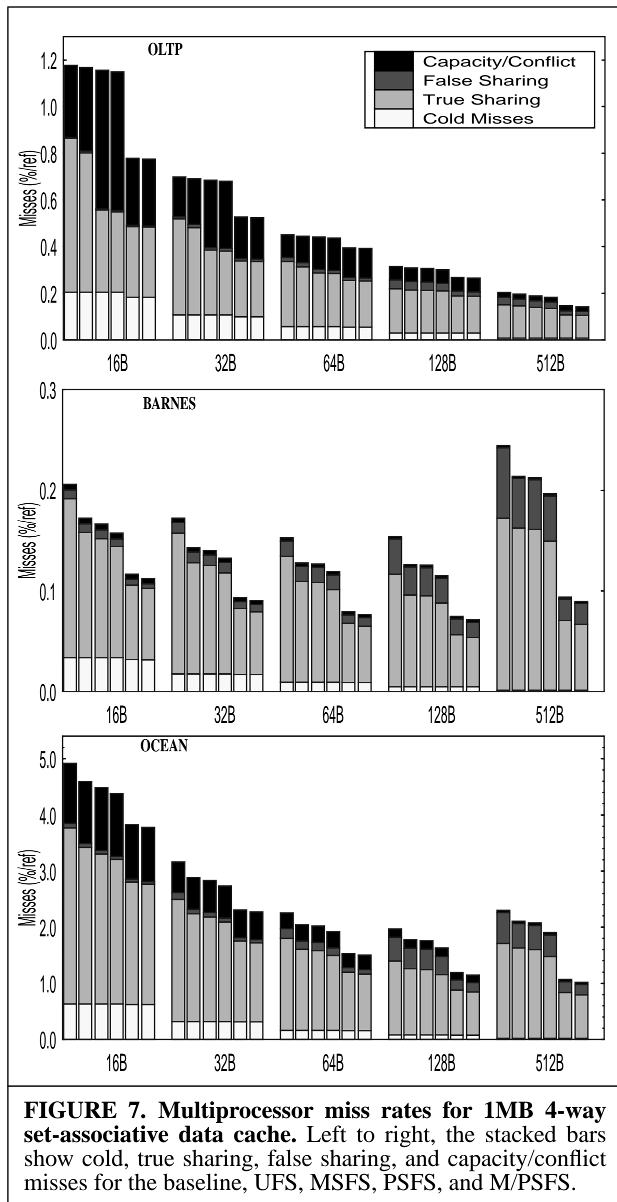

FIGURE 7. Multiprocessor miss rates for 1MB 4-way set-associative data cache. Left to right, the stacked bars show cold, true sharing, false sharing, and capacity/conflict misses for the baseline, UFS, MSFS, PSFS, and M/PSFS.

For UFS-P store squashing the total data bus traffic reductions observed were (depending on line size) 3%-23% for *oltp*, 10%-11% for *ocean*, and 13%-19% for *barnes*. For the most aggressive SFS case (scenario 5), we observed data bus traffic reductions of 15%-48% for *oltp*, 24%-55% for *ocean*, and 45%-63% for *barnes*. A detailed analysis of the variations in writeback reduction and data bus traffic is left to future work.

We also collected data on the address transactions needed to support coherence in the MESI protocol. Figure 8 shows both the outgoing (sent) invalidate rate and the incoming (received) invalidate rates (both hit and miss) for the six sharing scenarios and five line sizes. The stacked bar charts show the rate at which invalidates (including invalidates triggered by both store clean hits and store misses) hit in a remote cache, miss in a remote cache, and miss in a remote cache if E state is not implemented. The two plotted data points indicate the rate at which invalidates are sent out both with and without E state (recall that E state identifies a line as being exclusive in the local cache, hence a store clean hit only requires a silent E->M upgrade and not a broadcast invalidate, resulting in fewer total invalidates). For all three benchmarks we record measurable reductions in address traffic, even with just the simple UFS store squashing. Furthermore, there is a marked decrease in incoming invalidates that miss the local cache, indicating that the UFS and SFS approaches are most effective at eliminating useless invalidates (i.e. invalidates that consume address bus bandwidth but do not communicate any useful information). Since address bus bandwidth is a precious commodity in large-scale snoop-based shared-memory multiprocessors, this is a very useful and desirable property.

We also observe that the for this benchmark set, the address bus traffic reduction obtained by simple UFS store squashing is higher than the reduction obtained with the addition of E state, which is an optimization that is commonly implemented in real systems. In fact, we observe that UFS combined with a MSI coherence protocol that omits the extra complexity of the E state always generates less address bus traffic than a MESI protocol without UFS store squashing. Of course, combining both E state and UFS store squashing provides the lowest address bus traffic of all.

In summary, our data clearly show that measurable, even significant, reductions in address and data bus traffic in shared-memory multiprocessors can be achieved with simple UFS store squashing, and dramatic reductions can be achieved with the program structure-based approach to stochastic false sharing reduction. Of course, some of these gains will be countered by the traffic generated by the hypothetical mechanism used to enable SFS. As previously mentioned, the details of that design are left to future work.

6 .0 Conclusion

In this work, we explore various aspects of the value locality of store instructions. In doing so, we make five main contributions. The first of these is an overall characterization of store value locality from memory-centric (message-passing) and producer-centric (program structure) points of view; we find, not surprisingly, that significant value locality exists in both dimensions. Second, we introduce the notion of silent stores and quantify their frequency for many real programs. Silent stores are stores that do not affect the state of the machine they are executed on. Third, we describe how to enhance the performance of uniprocessor programs by squashing silent stores. Fourth, we define and quantify the concepts of update-based false sharing (UFS) and stochastic false sharing (SFS) in multiprocessor systems. Finally, we show how to exploit UFS to reduce address and data bus traffic on shared memory multiprocessors, and also examine the significant potential of hypothetical SFS-based mechanisms for reducing bus traffic. Our initial results in all of these areas are quite promising, and serve to motivate future work.

References

[1] SPLASH Benchmarks. http://www-flash.stanford.edu/apps/SPLASH/.

[2] 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.

[3] Brad Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International*

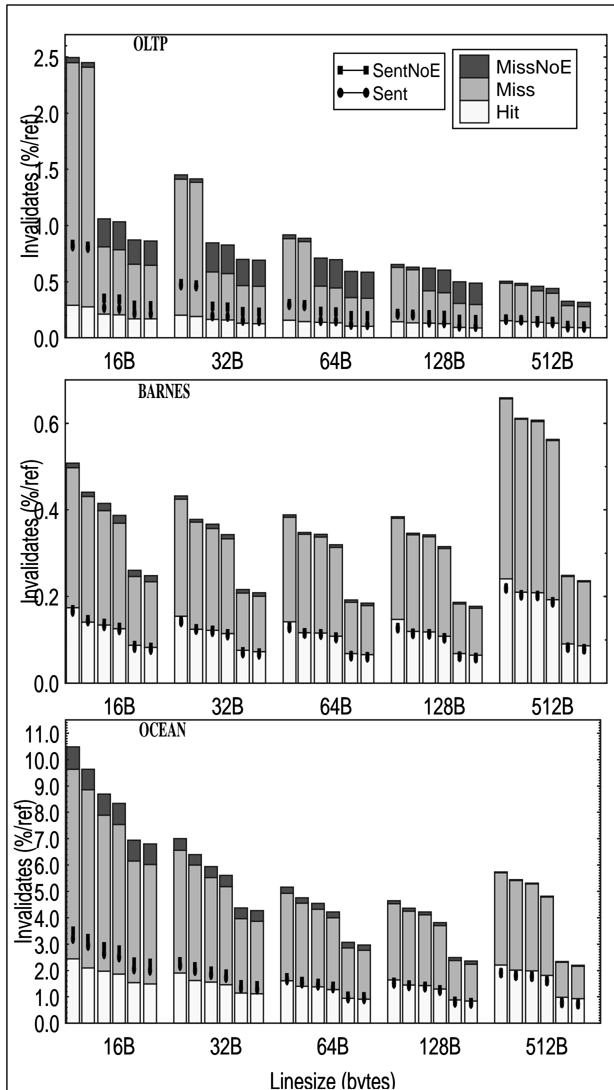

FIGURE 8. Multiprocessor invalidates for 1MB 4-way set-associative data cache. Left to right, the stacked bars and data points show invalidate traffic for the baseline, UFS, MSFS, PSFS, and M/PSFS scenarios.

Symposium on Microarchitecture, December 1997.

[4] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *20th Annual International Symposium on Computer Architecture*, May 1993. To appear.

[5] John H. Edmondson et al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1), 1995.

[6] Susan J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *Proceedings of ICPP-1991*, August 1991.

[7] J. Gonzalez and A. Gonzalez. Control-flow speculation through value prediction for superscalar processors. In *Proceedings of PACT-99*, October 1999.

[8] J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[9] Tom Keller, Ann Marie Maynard, Rick Simpson, and Pat Bohrer. Simos-ppc full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[10] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[11] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[12] Avi Mendelson and Freddy Gabbay. Speculative execution based on value prediction. Technical report, Technion, 1997. (http://www-ee.technion.ac.il/.

[13] Carlos Molina, Antonio Gonzalez, and Jordi Tubella. Reducing memory traffic via redundant store instructions. In *Proc. of Int. Conf. on High Perf. Computing and Networking*, pages 1246–1249, April 1999.

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

[15] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache misses. In *Proceedings of ICPP-1990*, August 1990.

[16] Transaction Processing Performance Council. TPC benchmarks. http://www.tpc.org.

[17] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, volume 26:4, pages 2–14, April 1991.