Silent Stores and Store Value Locality

Kevin M. Lepak, *Student Member*, *IEEE*, Gordon B. Bell, *Student Member*, *IEEE*, and Mikko H. Lipasti, *Member*, *IEEE*

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, including detailed source-level analysis of the causes of silent stores, 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.

Index Terms—Value locality, value prediction, store optimization, false sharing, cache coherence.

1 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. As a few examples of recent work, 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.

• *The authors are with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706. E-mail: {lepak, gbell}@cae.wisc.edu, mikko@engr.wisc.edu.*

Manuscript received 5 Dec. 2000; revised 25 May 2001; accepted 31 May 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114257.

The work described herein proposes a different approach for exploiting value locality. Rather than focusing on predicting the values that are fetched from memory via load instructions or that are computed within the processor with register-to-register ALU instructions, we instead focus on the values that are generated by all this activity and are ultimately written back to memory as the results of the computation. As one might expect, since input memory operands and intermediate ALU computations are quite predictable, the output values appear to be as well. Initial studies indicate that a considerable degree of value locality exists in this output operand stream and that significant potential exists for streamlining the handling of write operations to reduce memory traffic, simplify store-handling microarchitecture, and gain performance benefit [13], [17], [18], [19]. This paper classifies store instructions that exhibit value locality into two main categories: *update-silent stores* (or simply *silent stores*) and *stochastically silent stores*. The former—*update-silent stores*—are store instructions that truly are silent, that is, they have no effect on the state of the system since they are writing a value that is identical to the one that already exists in memory. The latter—*stochastically silent stores*—are store instructions that are not update silent (that is, they write a value that differs from the value that exists in memory), but we deem them to be interesting since they store a value that is predictable by some well-known value prediction scheme.

The purpose of the work presented in this paper is to examine opportunities for exploiting value locality to accelerate and streamline the processing of store instructions. Specifically, we introduce the notion of update silent stores and examine the value locality of stochastically silent stores from both memory-centric (data-address-based value prediction) and producer-centric (store instruction-based value prediction) viewpoints (Section 2); identify, classify, and characterize silent stores (Section 3); examine the

TABLE 1
Store Value Locality Measurements for Various Benchmarks

Benchmark	Description	Silent Stores (PPC/SS)	PSSVL (PPC) (LastVal/Stride)	MPSVL(PPC) (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%
tomcatv	SPEC95 vectorized mesh generation	47%/33%	40%/50%	45%/55%
swim	SPEC95 shallow water equations	34%/26%	20%/23%	19%/21%
mgrid	SPEC95 3D potential field	23%/7%	24%/26%	17%/19%
applu	SPEC95 partial differential equations	37%/35%	35%/37%	28%/31%
apsi	SPEC95 weather prediction	21%/25%	22%/23%	20%/22%
fpppp	SPEC95 Gaussian quantum chemistry	15%/15%	15%/16%	14%/15%
wave5	SPEC95 Maxwell's equations	25%/22%	30%/32%	20%/24%
oltp	4proc in-memory TPC-B w/RDMBS	56%	52%/56%	45%/51%
tpc-w	4proc TPC-W shopping mix	53%	53%/57%	42%/46%
specweb	4proc SPECWEB99	46%	39%/41%	38%/41%
specjbb	4proc SPECJPP2000 Java benchmark	37%	49%/51%	31%/33%
barnes	4proc SPLASH-2 N-body simulation	40%	27%/27%	38%/43%
ocean	4proc SPLASH-2 Ocean simulation	41%	41%/43%	36%/39%
raytrace	4proc raytrace with teapot input set	30%	34%/37%	40%/45%

implications of update silent stores on microarchitectural techniques for handling store instructions (Section 4); introduce the notion of critical silent stores (Section 5); introduce and quantify two new definitions of multiprocessor true sharing [4] based on store value locality (Section 6); and explore the potential for reducing multiprocessor data and address traffic (Section 7). Our initial results in all of these areas—many of which were published earlier [17], [18], [19]—are promising and should encourage and motivate further study of this topic.

2 The Value Locality of Stores

In order to better understand the value locality of stochastically silent 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), as introduced in [18]. 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 Mendelson and Gabbay's [12], which studies the value locality of instructions based on destination register identifier, and Calder et al.'s [3], which studies load value predictability based on memory address.

We examined the 15 uniprocessor and seven multiprocessor benchmarks described in Table 1. The uniprocessor silent store statistics were gathered under two instruction sets (PowerPC and SimpleScalar) using their respective compilers, runtime environments, and simulators. The store value locality and store silence measurements (columns 3-5) were collected for the PowerPC instruction set with the SimOS-PPC full system simulator [9], while the silence statistics for the uniprocessor benchmarks (column three) were also collected using Simplescalar [28]. The SPLASH-2 benchmarks [1] are widely used and understood and are included for continuity with previous work. The *oltp* benchmark employs 12 concurrent streams of TPC-B transactions [16] operating on an in-memory database stored in a commercial relational database management system (RDBMS), the TPC-W benchmark is modeled after an online bookstore; the SPECWEB99 benchmarks models a web server satisfying static and dynamic http requests [21], and SPECJBB2000 is a new Java server benchmark [22].

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 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

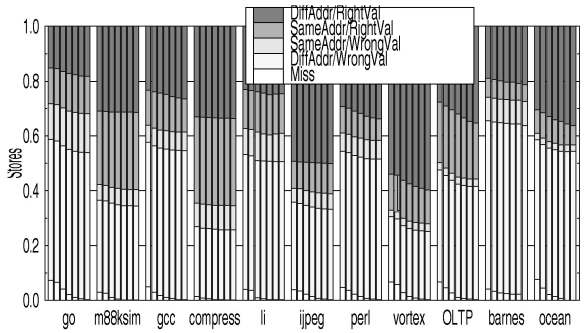


Fig. 1. Program structure store value locality. Dynamic store breakdown is shown for 1K, 2K, 4K, 8K, 16K, 32K, and 64K entry predictor tables for PowerPC.

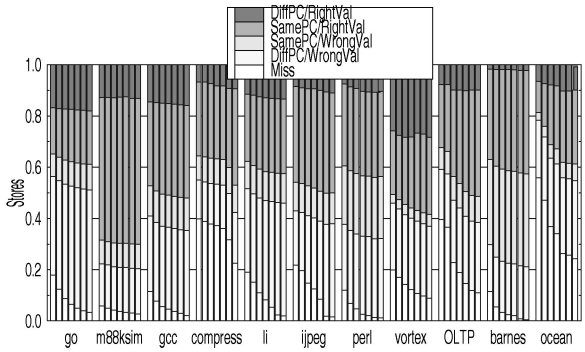


Fig. 2. Message-passing store value locality. Dynamic store breakdown is shown for 1K, 2K, 4K, 8K, 16K, 32K, and 64K entry predictor tables for PowerPC.

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. Because of the existence of these more complex predictors, and also the many possible methods of describing store value locality (PSSVL and MPSVL are two examples), we also define the notion of *stochastically silent stores*. Stochastically silent stores change the system state in some predictable manner. 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 Fig. 1 and Fig. 2 and will be described shortly.

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. We used the gcc and g77 compilers for Simplescalar; the PowerPC benchmarks were compiled with the IBM AIX optimizing C and Fortran compilers. 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.[1] However, it is interesting to note that, with only one exception (*apsi*), the percentage of silent stores in the PowerPC environment is higher than the SimpleScalar environment, leading us to believe that silent stores cannot simply be ascribed to inferior compilers, as in the case of Simplescalar. We do not explore this difference in greater detail in this work.

Fig. 1 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 cumulative predictability of store values ranges from a low of 27 percent for *barnes* to a high of 72 percent for *vortex*. Since there is a significant population of stores in each category, no clear correlation exists between address variability and store value variability.

Fig. 2 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 cumulative predictability ranges from a low of 39 percent for go to a high of 70 percent 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 7, where we explore the effects of store value locality on multiprocessor bus traffic.

1. Of course, the speculative stores are not retired into the memory system; we only indicate here that our statistic gathering code included speculative stores, as is possible in an execution driven simulator.

```
for (i=0; i<32; i++){
   m88ksim.time_left[i] –= MIN(m88000.time_left[i], time_to_kill);
}
```

Fig. 3. *m88ksim*: Frequently, the store in all 32 loop iterations is silent, allowing the entire loop to be bypassed.

2.1 Identifying Causes of Silent Stores

In order to develop a better understanding of the causes for store value locality, we shift our attention to a program source analysis of update-silent stores. This study was conducted using a profiling infrastructure based on Simplescalar [28]. As described earlier, update-silent stores are stores that write a value to a memory location that already exists there. Given the frequency of silent stores shown in Table 1, a primary question might be "Why do silent stores exist at all?" What causes a compiler or programmer to insert stores that have no effect on the state of the processor? One explanation lies in the nature of program generality. Programs are often designed to operate on a variety of data input sets and, while stores may update memory contents when executed with some inputs, they may not with others [25]. Consider an example from the *m88ksim* benchmark (Fig. 3).

This instruction continually decrements an array element by the minimum of itself and the unsigned value *time_to_kill*. Clearly, at some point, the value stored in memory will converge to zero and every subsequent store to that location will continue to store zero. In the input data sets included with SPECINT-95, this convergence occurs quickly and, in over 95 percent of all executions, this store is silent. Because this particular piece of code is heavily executed, this single static store is responsible for over half of the benchmark's five million dynamic silent stores.

Furthermore, not only is this store frequently silent, but 27 percent of the time the stores in all of the 32 loop iterations are silent. Thus, if it is known that every element in the array is zero, the entire loop can be skipped. This could be exploited in a technique similar to memoization [23], [24] or dynamic instruction reuse [25], in which the values of the array are used to index a table whose outcome conditionally determines if the loop should be executed. This leads to a substantial savings as not only would the silent stores be skipped, but also the call to the MIN function, the subtraction of *time_to_kill from m88000.time_left[i]*, the loop induction variable calculations and the backwards branch to the start of the loop. By just skipping the loop when all array elements are zero, 9.6 million fewer instructions need to be executed or about 11 percent of the total dynamic instruction count.

This example leads to four classifications of dynamic silent stores based on their previous execution (though similar to the store value locality classifications presented in the previous section, these are a subset of the general PSSVL classification and consider only silent stores):

Same Location, Same Value: A static silent store stores the same value to the same location as the last time it was executed. This is the case in the eval function from the *perl* benchmark in Fig. 4.

Each iteration through this loop processes an argument and stores the argument's flags in the stack-allocated

```
for (anum=1; anum <= maxarg; anum++){
    argflags = arg[anum].arg_flags;
```

Fig. 4. *perl*: Same Location, Same Value.

temporary *argflags*. Many arguments have *arg_flags* equal to zero, thus the store is silent 71 percent of the time.

Different Location, Same Value: A silent store stores the same value to a different location as the last time it was executed. This situation often occurs when an instruction is storing to an array indexed by a loop induction variable, such as the *m88ksim* example just presented and the Fig. 5 example from *go*. The bottom three stores initialize every location in the arrays that represent the game board to the same value (zero or, equivalently, FALSE). However, the stores' addresses are derived from the loop index and, thus, are different in each iteration. Often the values on the board already contain zero (or FALSE), so these three stores are silent 86 percent, 43 percent, and 77 percent of the time, respectively.

Same Location, Different Value: A silent store stores a different value to the same location as the last time it was executed. The store to *ltrscr* in Fig. 5 is of this type. *ltrscr* is a global variable (thus its location does not change), but stores a different element (and, thus, a potentially different value) of the *ltr2* array. If another instruction updates *ltr2* between successive executions of this function, the store has the potential to be silent in the first loop iteration. Although this store is silent 86 percent of the time, 98 percent of the time that it is silent consecutive elements of *ltr2* are zero and the store is usually Same Location, Same Value.

Although silent stores of this category are generally rare, they usually exist due to one of several circumstances. Many times the silent value was previously stored to memory by another instruction, corresponding to message-passing store value locality (MPSVL). More often, Same Location, Different Value silent stores are caused by compiler and architecture conventions that dictate how stack frames are used. For example, when a function is invoked in a callee-save register convention, it immediately saves the contents of a predefined set of registers onto its stack frame (although it does not necessarily have to if it will not modify those register contents before returning). If these register contents do not change between calls from within the same function, each called function will silently store them onto the same stack frame. Similarly, a function

```
for (x=xmin; x<=xmax; ++x)
  for (y=ymin; y<=ymax; ++y){
     s = y*boardsize+x;
     .....
     ltrscr –= ltr2[s];

     ltr2[s] = 0;
     ltr1[s] = 0;
     ltrgd[s] = FALSE;
  }
```

Fig. 5. *go*: Different Location, Same Value (last 3); Same Location Different Value (*ltrscr*).

```
NODE ***xlsave(NODE **nptr, ...){
    ......
    for (; nptr != (NODE **)NULL; nptr = va_arg(pvar, NODE **)){
        ...
        *--xlstack = nptr;
        ...
    }
```

Fig. 6. *li*: Different Location, Different Value.

often stores its caller's frame pointer and its return address onto its stack frame. Because these values do not change between successive function calls from the same caller, they too are likely to be silent. A final case of silent stores of this category involve architectures that use the stack to pass function arguments (either by passing arguments explicitly on the stack or by saving register contents to memory before using them to pass arguments). A variety of static compiler optimizations exist that can remove many such stack stores [30], some of which may be silent. Further discussion of the compiler's role in eliminating silent stores appears in Section 3.3.

Different Location, Different Value: A silent store stores a different value to a different location as the last time it was executed. Examples include nested loops that store a set of values into an array in the inner loop. The *xlsave* procedure that saves a set of nodes onto the stack in the *li* benchmark is one such case (Fig. 6). In the first execution of *xlsave*, each iteration of the loop sets *nptr* equal to the next function argument (different value) and decrements the store address (different location). However, if subsequent calls to *xlsave* store the same set of nodes to the same starting stack address, each store instruction will be silent (as it is 48 percent of the time).

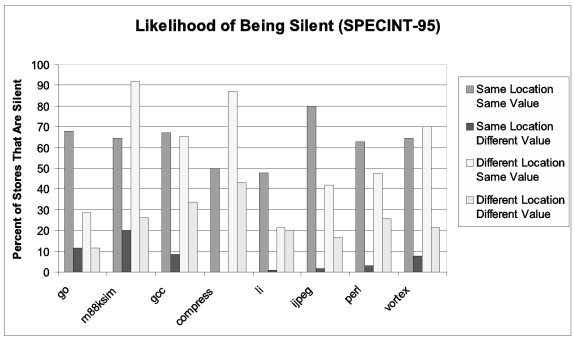
A store's probability of being silent based on these categories is shown in Fig. 7. In all of the benchmarks, instructions that consecutively store the same value are more likely to be silent than if they were storing a different value. An important corollary to this graph is Fig. 8, which adds instruction frequency information by showing each silent store category's contribution to total silent stores. Because stores that have a high likelihood of being silent are also executed frequently (this will be shown in Section 3.2), they represent a significant portion of all dynamic silent stores. One could imagine using such categories as an aid to predict if a store will be silent or not early in the pipeline (and, thus, if a store verify should be performed). Such a mechanism is not discussed here and remains an opportunity for future work.

3 Silent Store Characteristics

Having examined causes for silent stores in source code with numerous examples, in this section, we take a step back and examine high level characterizing data for silent stores in the hope that understanding their characteristics better will aid in exploiting them architecturally.

3.1 Program Phase Characteristics

In order to examine variation due to program phase, we measured the time domain frequency of occurrence of silent



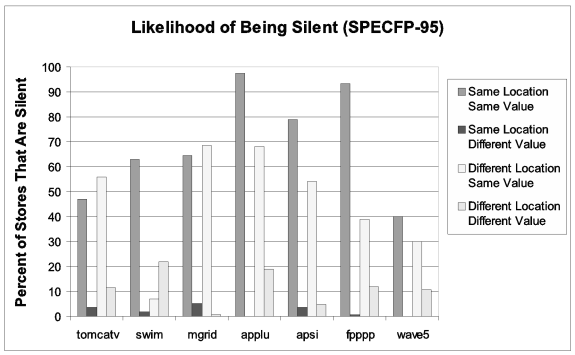
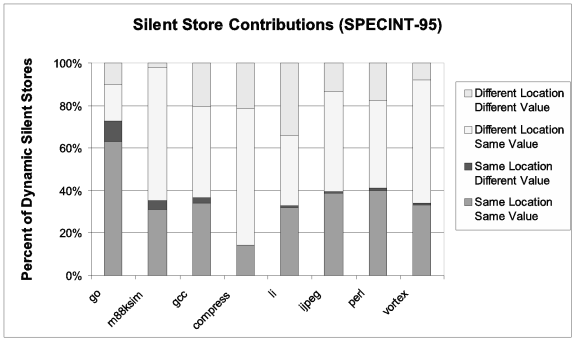


Fig. 7. Probability of a store being silent as a function of store category.

stores for the PowerPC architecture. This data is plotted in Fig. 9. With the exception of *oltp*, which is a snapshot of steady-state execution, all the benchmarks have a nearly identical cumulative fraction of silent stores in the first several hundred thousand stores. We attribute this to the program loader (this data was collected with a full-system



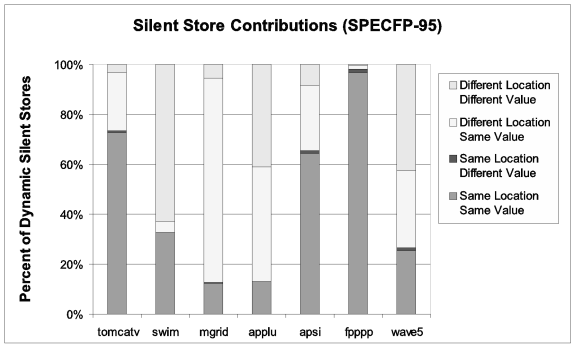


Fig. 8. Total silent store contributions as a function of store category.

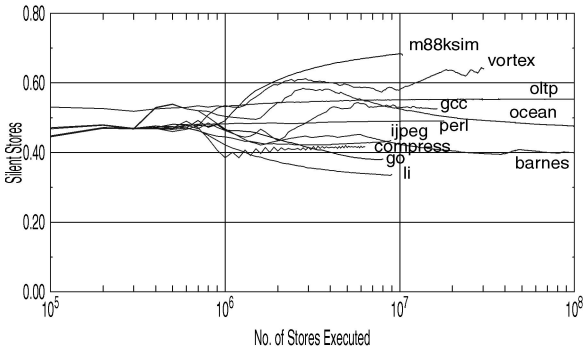


Fig. 9. Silent Stores vs. Time. Cumulative fraction of silent stores throughout program execution. Note that the x-axis is on a log scale.

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 many fewer silent stores than the program loader does.

3.2 Frequency of Execution Related to SVL

Fig. 10 examines a store's probability of being silent as a function of how many times it is dynamically executed under SimpleScalar. It graphs the contribution of additional static silent stores (in order of decreasing number of total dynamic silent stores) toward the total number of dynamic

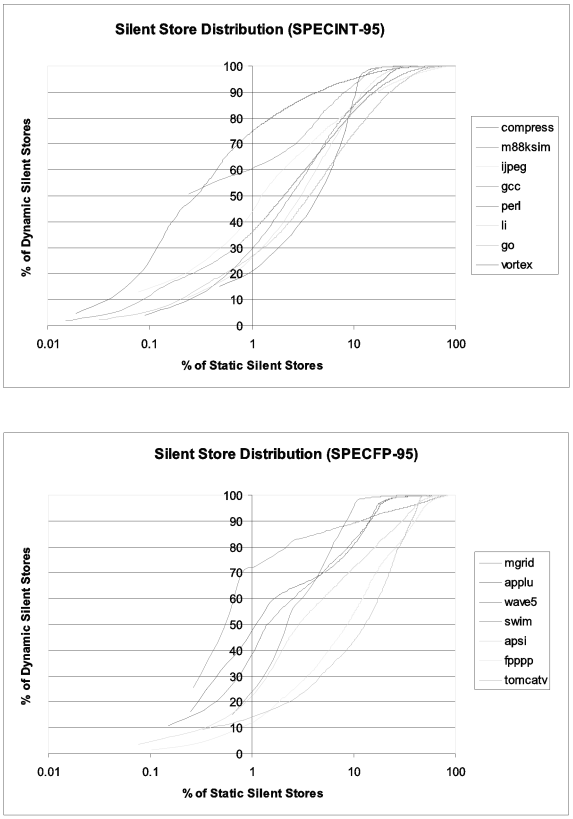


Fig. 10. Total silent stores as a percentage of static silent stores (note: the benchmark order in the legend matches the order when the x-axis is at 1 percent.

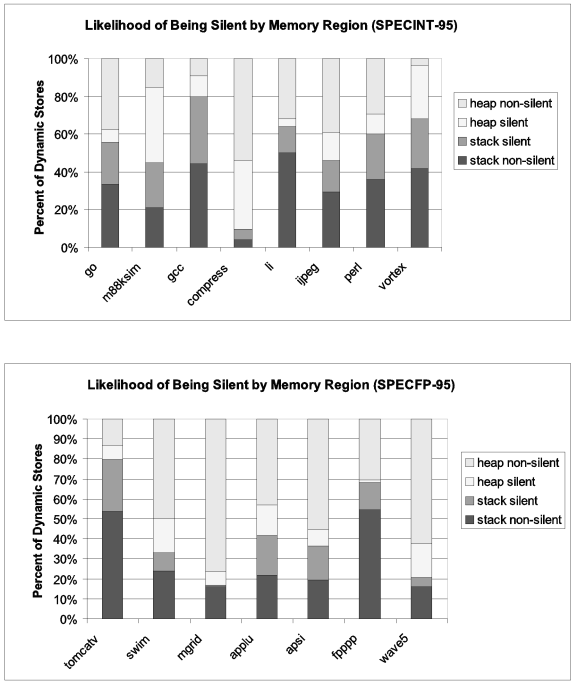


Fig. 11. Silent stores categorized by memory region.

silent stores in each benchmark. Note the log scale on both axes. In all of the integer cases, less than 25 percent of all static silent stores contribute over 90 percent of the total silent stores dynamically executed. The floating point benchmarks exhibit a similar, though not as dramatic, distribution. In certain benchmarks (such as *mgrid* and *compress*), 1 percent of static silent stores contribute over 70 percent of dynamic silent stores. This uneven distribution of silent store contribution across a variety of programs reveals that elimination (or at least modification) of even only a few static stores can have a significant impact on silent store reduction. Static techniques for removing these few, frequently silent stores seem appropriate and we leave such methods for future work.

3.3 Stack/Heap SVL

Most architectures have a distinct way of dividing memory areas into a stack and heap region. We examined whether a store was more likely to be silent depending on which portion of memory it was to, hoping to gain some insight as to whether function parameters or some other "local" variables were more likely to be silent than heap-allocated data. In SimpleScalar, the stack begins at the largest address of memory and the heap begins at the smallest. We counted stack and heap references by choosing an address threshold (one billion) and counting separate silent store statistics for instructions that generated effective addresses above and below this threshold. The results are shown in Fig. 11.

In general, little variance exists between benchmarks in how likely a store to the stack is of being silent. This suggests that there are specific attributes of how the stack is used by architectures and compilers that dictate silent stores (for example, callee-saving register values, saving return addresses, and parameter passing between functions). Because the percent of stack stores that are silent is

nontrivial in most of the benchmarks (typically 25-50 percent with the exception of *mgrid* and strongly correlated to the overall utilization of the stack in a given benchmark), it seems likely that modifications in the way that the compiler manipulates stack frames could eliminate a substantial number of silent stores. Conversely, the benchmarks exhibit high variability in the percent of heap stores that are silent. Sometimes there are far more nonsilent than silent heap stores, while other times the reverse is true. This can be explained by the fact that virtually all heap accesses are programmer directed—the compiler does not transparently manipulate it as it does to the stack. Thus, the notion of whether a heap store is silent is largely algorithmic and, so, varies greatly between programs. We also see from Fig. 11 that the breakdown of total stack and heap stores differs across the benchmarks. Control-intensive programs with many function calls (such as *gcc*) have many stack stores due to their large number of stack frame allocations, while programs that operate on large heap-allocated data sets (such as *compress* and many floating-point applications) do not. Additionally, many data structures can be stack or heap-allocated and the decision of which differs among programmers and applications.

4 Improving Store Handling Microarchitecture

Now that we have introduced the concept of silent stores and have examined their occurrence in programs, can we find a way to exploit them at the microarchitectural level? In this section, we outline some microarchitectural structures that may be enhanced by exploiting the fact that many stores are silent and explore initial experimental results under SimpleScalar [28]. For the entirety of the microarchitecture discussion, the term "silent store" refers only to update-silent stores—those which make no change to the system state because the value they write already exists at that memory location. These seem intuitively easiest to exploit with little microarchitectural complication. We will explore possible benefits of stochastically silent stores in Section 7. For the microarchitectural discussion, we consider only a weakly consistent uniprocessor memory model that enables aggressive removal of silent stores.

4.1 Core Microarchitecture and Memory Hierarchy Enhancement

In many modern microprocessors, memory performance is a bottleneck, even with the latency tolerance that 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, nonblocking caches, and deep load/store buffers have been used to improve system performance [14]. 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* (removing silent stores from program execution) may allow a designer to obtain greater performance from existing memory ordering structures or a reduction in size or complexity of these structures (which we herein refer to as the load/store queue, or LSQ) because of the relative benefit of squashing.

Squashing can provide performance benefit because squashed silent stores can be physically removed early from the LSQ, effectively making the LSQ larger. This removal can occur because the stores are not updating the memory image, therefore, any subsequent loads will obtain the correct value from a previous store or from the cache, discounting this silent store.[2] 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 and write buffer (locations used to hold committed but uncompleted stores) pressure. The mechanism we use to squash silent stores (covered in detail in Section 4.2) has the added benefit of prefetching store misses into the L1-D cache before the store commits, reducing the load-to-use latency for subsequent reads of the line and also reducing the occupancy in write buffers, even for nonsilent stores. The fact that many stores are silent has also been exploited to more aggressively allow loads to bypass earlier, aliasing, stores without needing to forward data explicitly from the aliasing store because it is silent [27].

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 these experiments is covered in Section 4.2).

Silent store squashing can also 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. Writeback traffic has been shown to be important in emerging classes of streaming applications in uniprocessor systems [26].

4.2 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 nonsilent). 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 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, a store port and write buffer are obtained and the write occurs to the memory system as it would normally. If the store is silent, the store retires with

2. We assume a weakly consistent system. This statement may need to be qualified in less relaxed consistency models to maintain valid memory ordering. This qualification is beyond the scope of this work.

TABLE 2
Update-Silent Store Squashing Effect on L1 Wrtiebacks

	Number of writebacks (% reduced relative to baseline case)							
Benchmark	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%)

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.).

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. Nonsilent 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.

4.3 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 [28]. 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 eight issue; 64 entry RUU; GShare branch predictor with 64K entries, 16 bit global history; six integer ALUs; two integer multipliers. The cache configurations are 64KB each split I/D L1 and 512KB 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 two clocks to match the L1 cache latency. There are four write buffers (which hold committed but uncompleted stores) and four writeback buffers. A fully pipelined, scheduled, 16 byte wide interface between the L1 and L2 caches is modeled, including an extra cycle for bus turnaround.

The core has three general purpose load/store ports with no address restrictions on parallel loads/stores. All stores start their associated store verify (described in Section 4.2) at issue, but, if the store reaches commit before it can be verified, it is assumed to be nonsilent and either enters a write buffer (if one is available) or commit is stalled. Not waiting for store verifies foregoes some opportunities for writeback reduction (explained in Section 4.4.1), but can help overall instruction throughput because the store is allowed to leave the instruction window immediately and it need not wait until the store verify completes. This was determined to be the best policy experimentally for the benchmarks presented here. 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.

4.4 Results

4.4.1 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. Note that, in contrast to the machine model presented in Section 4.3, in order to show maximal writeback reduction in these results, we allow *all* store verifies to complete regardless of their location in the memory hierarchy. All other machine parameters are unchanged.

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 percent (in *vortex*) to 0 percent (or small negative values in *li*, which we attribute to second-order LRU policy effects). The average for all benchmarks is a 33 percent reduction.

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

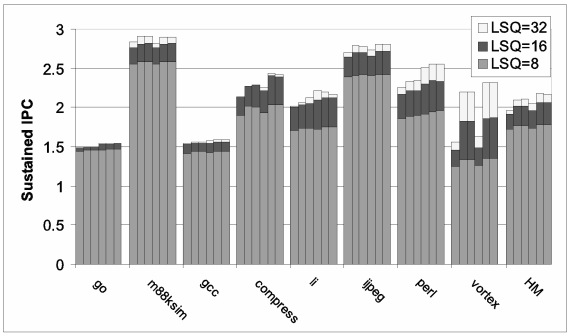


Fig. 12. SPECINT Performance with Store Squashing. For each benchmark, the left three bars show performance without store forwarding, while the right three bars show performance with store forwarding. The three bars in each group show baseline IPC, IPC with realistic store squashing, and IPC with perfect store squashing.

is 100 percent, but, if we verify the miss, the probability is less than 100 percent.

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 nonsilent 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 et al. [13].

4.4.2 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.

In Fig. 12, the three lefthand bars in each group represent performance with no store forwarding (SF) and the right-hand group represents 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 Fig. 12 that in only one case does realistic silent store squashing decrease performance in our processor model (*li* with SF, LSQ = 32), even with the added store verify operations. We attribute this slight performance degradation in *li* to added contention for memory ports caused by store verify operations with relatively little benefit due to the low percentage of silent stores in this benchmark, 20 percent.

More interestingly, we see in *m88ksim*, *compress*, *ijpeg*, and *vortex* better performance for an LSQ size of 16 and realistic squashing vs. LSQ size of 32 and no squashing; better performance for half the LSQ size. In *vortex*, the gains are 16 percent and 13 percent for no SF and SF, respectively. 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, *perl*, which obtains 12 percent speedup from SF alone with an LSQ size of 32, and *li*, as explained previously).

Comparing realistic squashing with perfect squashing, we see an interesting occurrence in *compress*, namely, realistic squashing outperforms perfect squashing by nearly 1 percent (SF, LSQ = 32). While this difference is small in absolute terms, it shows that squashing silent stores can have other second-order benefits besides affecting only structures meant for handling stores. We attribute this difference in *compress* to the prefetching effect realistic squashing has for nonsilent stores. However, there is one case where perfect squashing performs measurably better than realistic squashing: In *li* the difference is approximately 3 percent (no SF, LSQ = 32) due to reduced port contention in perfect squashing.

Over the range of benchmarks for LSQ size of 32, we see speedups of 6.5 percent for realistic and perfect squashing with SF over the baseline with SF (7.0 percent and 7.4 percent, respectively, for each without SF over the baseline without SF.) In *vortex*, we see a speedup of 41 percent—due mostly to a 60 percent reduction in the number of cycles the LSQ was full and a 65 percent reduction in the number of cycles commit was stalled due to full write buffers. Of course, as evidenced in Fig. 12, SF always provides some additional benefit along with squashing because not *all* stores are silent. In general, the performance difference between realistic and perfect squashing is small (less than 0.5 percent), as discussed previously, leading us to believe 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 4.5 percent (contrasted with the 6.5 percent of both realistic and perfect squashing with SF, 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) and pipeline configurations similar to the one modeled by SimpleScalar, given equal hardware costs, silent store squashing provides greater benefit than store forwarding.

5 Critical Silent Stores in Uniprocessors

Although a silent store does not update the contents of stored data, it does set a line's dirty bit in a write back cache configuration. If the dirty bit would not have been set otherwise, this writeback could be avoided by squashing the silent store.

Definition. *A* critical silent store *is a specific dynamic silent store that, if not squashed, will cause a cache line to be marked as dirty and, hence, require a writeback.*

This definition applies for each lifetime of the given cache line in the cache (the time between each allocation and replacement). Each cache line lifetime may have zero to n critical silent stores. Trivially, if there are no stores to the line, there are no critical silent stores either. Similarly, if there is even one nonsilent store, there are no critical silent

TABLE 3
Writeback Reduction Due to Critical Silent Stores in Various Cache Configurations

	32KB/32B			8KB/32B			8KB/8B		
SPEC95 Benchmark	Baseline WB/inst (x10^{-3})	% WB reduced by squashing	% of silent stores that are critical	Baseline WB/inst (x10^{-3})	% WB reduced by squashing	% of silent stores that are critical	Baseline WB/inst (x10^{-3})	% WB reduced by squashing	% of silent stores that are critical
go	1.04	9.2%	4.5%	5.57	6.4%	6.6%	6.51	14.0%	10.3%
m88ksim	0.30	56.0%	14.0%	0.36	59.0%	15.0%	1.05	64.2%	15.4%
gcc	1.14	18.8%	13.9%	2.67	23.3%	8.9%	5.58	30.1%	17.6%
compress	11.5	58.7%	71.0%	12.8	58.4%	71.4%	33.8	75.9%	81.9%
li	1.64	0.7%	1.1%	4.13	1.3%	2.8%	13.0	1.4%	6.6%
ijpeg	1.35	19.7%	16.5%	2.99	14.5%	23.4%	9.75	20.4%	31.9%
perl	0.30	9.0%	0.3%	4.95	16.6%	8.9%	9.90	25.7%	17.6%
vortex	6.17	78.7%	38.7%	7.93	70.6%	47.1%	26.3	82.4%	49.1%
tomcatv	1.62	3.4%	0.6%	2.20	4.6%	1.5%	3.83	6.1%	2.7%
swim	6.35	10.6%	24.4%	21.0	24.8%	46.1%	22.0	20.2%	51.9%
mgrid	3.96	6.6%	77.5%	4.11	6.6%	78.9%	14.0	7.7%	89.6%
applu	4.93	6.3%	7.1%	4.99	6.6%	7.4%	19.5	30.6%	52.4%
apsi	4.50	20.9%	19.7%	5.28	23.8%	24.9%	12.5	20.6%	32.8%
wave5	13.6	14.1%	55.0%	21.7	16.9%	63.4%	48.2	16.6%	68.9%

stores. However, if there are one or more silent stores to the line and no nonsilent stores, the former set of silent stores is defined as critical since failing to squash any of them will result in a writeback. Put more simply, it is sufficient to only squash the critical silent stores to obtain maximal writeback reduction. Squashing noncritical silent stores has no benefit (in terms of writeback reduction) because the line will be written back when it leaves the cache anyway because a nonsilent store wrote the line during its lifetime. Furthermore, squashing noncritical silent store misses can actually degrade performance because we incur the load and compare overhead of a store verify without any compensating reduction in writebacks. This problem is worse in multiprocessors. A noncritical silent store is replaced with a store verify (read), but a subsequent nonsilent store to that cache line will require that the line be upgraded from a shared to modified state (requiring an upgrade/invalidate bus transaction). We can be sure this upgrade will in fact take place because of the definition of a critical silent store. If the noncritical silent store suffered a cache miss and was not squashed, the line would have been brought into the cache with a read-with-intent-to-modify transaction (hence obtaining the line directly in modified state) and the upgrade message would not be required. Thus, squashing a noncritical silent store miss leads to an additional address bus transaction—namely the line upgrade when a nonsilent store to the line occurs.

We determine the critical silent stores in our SimpleScalar simulator in the following way: When a line enters the cache, we allocate a list (called the candidate list) to hold all stores to that line during its lifetime. When a store accesses the line, we add it to the candidate list for that line. When the line leaves the cache, we check if the line is dirty. If it is dirty, we know all stores to the line are noncritical (because a writeback of the line occurs anyway). If it is not dirty, if any stores exist in the candidate list, we know that all of these stores must have been silent and also, since a writeback is not occurring, they are all critical (by definition). We then account for the stores properly in the global critical silent store tracking structures as the current lifetime for the line has ended. Finally, we clear the candidate list.

Table 3 presents the number of writebacks in each SPEC benchmark without silent store squashing (baseline case), the percent of those writebacks that are eliminated when silent stores in all levels of the memory hierarchy are squashed and the percentage of silent stores that are critical for three different cache configurations (32KB cache with 32B lines, 8KB cache with 32B lines, 8KB cache with 8B lines). We see that, in some cases, selectively squashing only a fraction of all silent stores can dramatically reduce the total number of writebacks incurred by a program (for example, in vortex, 79 percent of the total writebacks can be eliminated by squashing only 39 percent of all silent stores). We also see that decreasing the cache size increases the number of silent stores that are critical because lines spend less time in the cache before being replaced. This highlights the fact that our definition of critical stores depends on cache line lifetime and can be influenced by such factors as cache size, associativity, and replacement policy. Since lines are replaced more frequently, there is not as much opportunity for nonsilent stores to write to them and the lines have a better chance of leaving the cache unmodified. Decreasing the line size also increases the number of critical silent stores for the same reason that decreasing line size helps to eliminate false sharing. If there are few words per cache line, there is less chance that one of them will be written to by a noncritical silent store. However, it is important to remember that, even though decreasing line size and total cache size may yield a greater percentage of writebacks that can be eliminated by squashing, the total number of writebacks increases as well.

Because no nonsilent stores can occur to a line if it is to avoid being written back, the greatest potential for writeback reduction exists when multiple silent stores occur to the same cache line. In other words, if a segment of code silently stores data that consumes an entire line, there is no room in that line for other data that, when stored nonsilently, can cause a writeback. Such a case is most prevalent inside a loop body when a store continually increments its address by a fixed offset (different location and different/same value). When the cache line is written back, every block has been silently stored to and the writeback can be avoided. A similar example of this occurs

```
do {
    *(htab_p−16) = −1;
    *(htab_p−15) = −1;
    *(htab_p−14) = −1;
    *(htab_p−13) = −1;
    *(htab_p−12) = −1;
    *(htab_p−11) = −1;
    *(htab_p−10) = −1;
    *(htab_p−9)  = −1;
    *(htab_p−8)  = −1;
    *(htab_p−7)  = −1;
    *(htab_p−6)  = −1;
    *(htab_p−5)  = −1;
    *(htab_p−4)  = −1;
    *(htab_p−3)  = −1;
    *(htab_p−2)  = −1;
    *(htab_p−1)  = −1;
    *htab_p      = −1;
} while ((i −= 16) >= 0);
```

Fig. 13. *compress*: Because these silent stores span across multiple cache lines, they are often critical.

in *compress* (Fig. 13) when the *cl_hash* function clears every entry in a large hash table resident in contiguous memory. Because many of the hash entries remain unused (and already contain the initialization value -1), most of the stores are silent. Each time this occurs for every entry in a line, a writeback can be removed, leading to over 76,000 removable writebacks from squashing only these silent stores (a 19 percent reduction in total writebacks for this benchmark).

Another situation in which writebacks can be removed occurs when multiple fields of a structure are silently stored to at nearly the same time. Because structure fields exhibit a great degree of spatial locality, it is likely that the silent stores occur within the same cache line and, thus, the writeback can be eliminated. The *makesim* function found in *m88ksim* illustrates this (Fig. 14). In many cases, the address pointed to by *opcode* and the results of the table lookup (*tblptr*) are identical to earlier invocations of *makesim* and the stores to *opcode*'s fields are silent. Additionally, because the size of the *IR_FIELDS* struct is the same as the simulated line size (32B), these stores are likely to fall within the same line, leading to over 1,000 removable writebacks if squashed.

To summarize, the issue of whether or not a silent store is critical depends on several factors, including the temporal locality of the address, spatial and temporal locality within a cacheline, cache size and configuration, and general program behavior.

6 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

```
void makesim(unsigned int instr, struct IR_FIELDS *opode){
   register INSTAB *tblptr;
   if(!(tblptr=lookupdisasm(instr & classify(instr))))
   tblptr = &simdata;
   opcode−>op = tblptr−>flags.op;
   opcode−>dest=uext(instr,tblptr−>op1.offset,tblptr−>op1.width);
   opcode−>src1=uext(instr,tblptr−>op2.offset,tblptr−>op2.width);
   opcode−>p = tblptr;
   .....
}
```

Fig. 14. *m88ksim*: Critical silent stores often occur when stores of multiple structure fields are silent.

that many stores are silent and even more are stochastically silent requires new definitions of true and false sharing.

6.1 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 invalidate 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 et al. [4] as it provides the most accurate definition of address-based sharing. For a review of other definitions developed prior to Dubois et al. 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 nonessential 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 noncritical 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.

6.2 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 roman):

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.

6.3 Stochastic False Sharing (SFS)

In light of the work of Lipasti et al. [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 interprocess 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.*

An example that illustrates our new definitions is provided in [21].

6.4 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 et al. [4] and exercise it with our multiprocessor benchmarks for PowerPC under six different scenarios:

- The baseline scenario corresponds to the Dubois definition of false sharing and treats stores just as Dubois et al.'s mechanism [4], 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 4; 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. In this scenario, only cache hits are squashed because of the potential effect squashing misses has on coherence transactions (outlined under UFS-P).
- 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 ($S \rightarrow M$) for nonsilent 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 Fig. 2.
- 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.

- 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, aand 512B are plotted in Fig. 15. For the commercial workloads on the left side, we observe measurable but not dramatic 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* and *tpc-w*, 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 can be squashed.

7 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 6.4: 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. Fig. 16 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 6.4.

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 and Goodman [29], 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 Fig. 16 since the frequency of writebacks of dirty lines is also reduced. With UFS-P store squashing, we observed 5-82 percent reductions (depending on line size) in the writeback rates for *oltp*, 16-17 percent reductions for *ocean*, and 5-16 percent for *barnes*. For the most aggressive SFS case (scenario 5), we observed writeback rate reductions of 8-85 percent, 25-26 percent, and 16-29 percent for *oltp*, *ocean*, and *barnes* respectively.

For UFS-P store squashing, the total data bus traffic reductions observed were (depending on line size) 3-23 percent for *oltp*, 8-12 percent for *specweb*, 4-6 percent for *specjbb*, 6-30 percent for *tpc-w*, 10-11 percent for *ocean*, 10-12 percent for *raytrace*, and 13-19 percent for *barnes*. For the most aggressive SFS case (scenario 5), we observed much higher data bus traffic reductions (e.g., 15-48 percent for *oltp*, 24-55 percent for *ocean*, and 45-63 percent 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. Fig. 17 shows the incoming invalidate rate (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 (recall that E state identifies a line as being exclusive in the local cache, hence, a store clean hit only requires a silent $E \rightarrow M$ 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

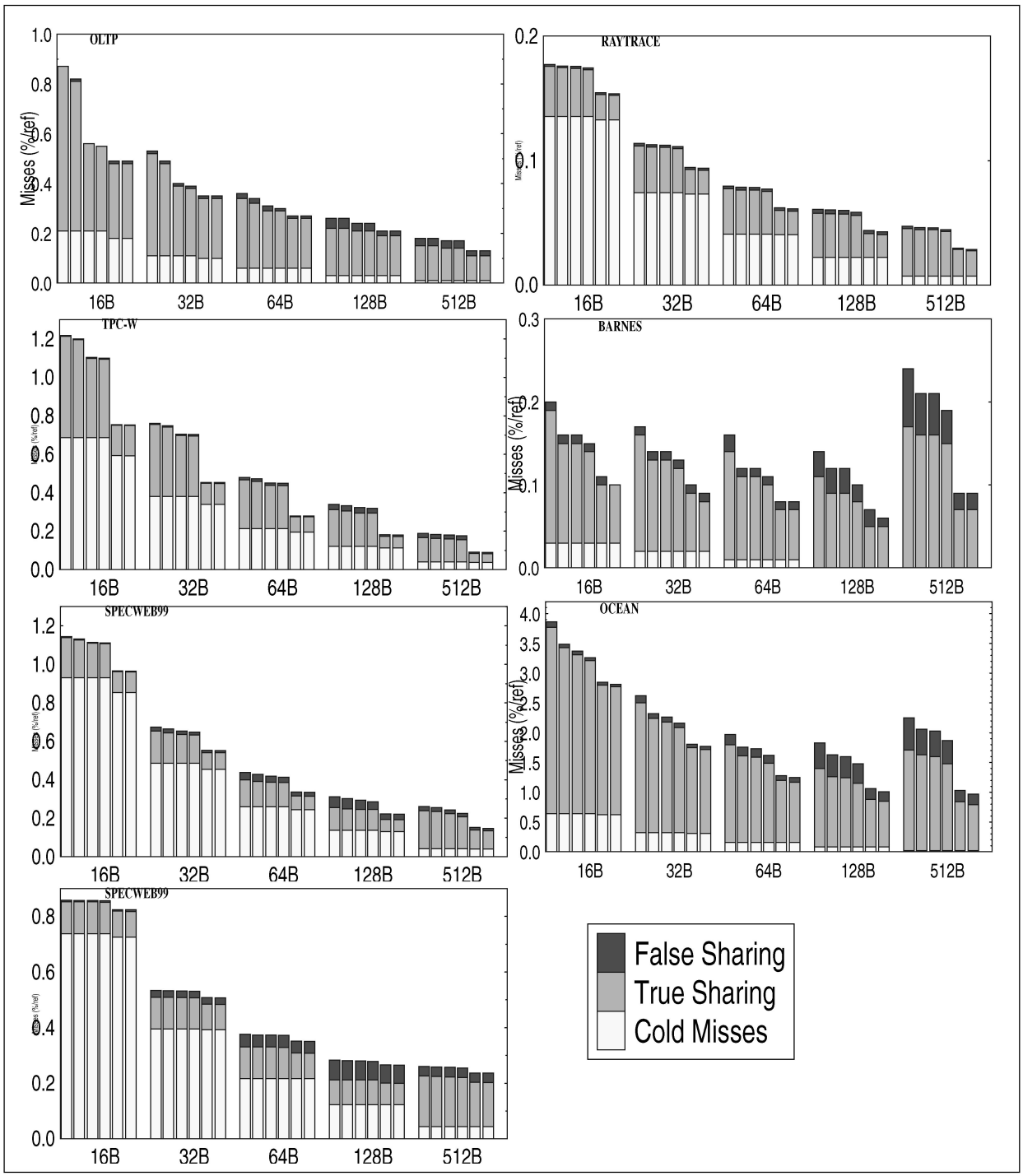


Fig. 15. Multiprocessor sharing. Left to right, the stacked bars show cold, true sharing, and false sharing misses of the baseline, UFS, UFS-P, MSFS, PSFS, and MSFS+PSFS scenarios.

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, 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 an MSI coherence protocol that omits the extra complexity of the E state always generates less address bus traffic than an 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

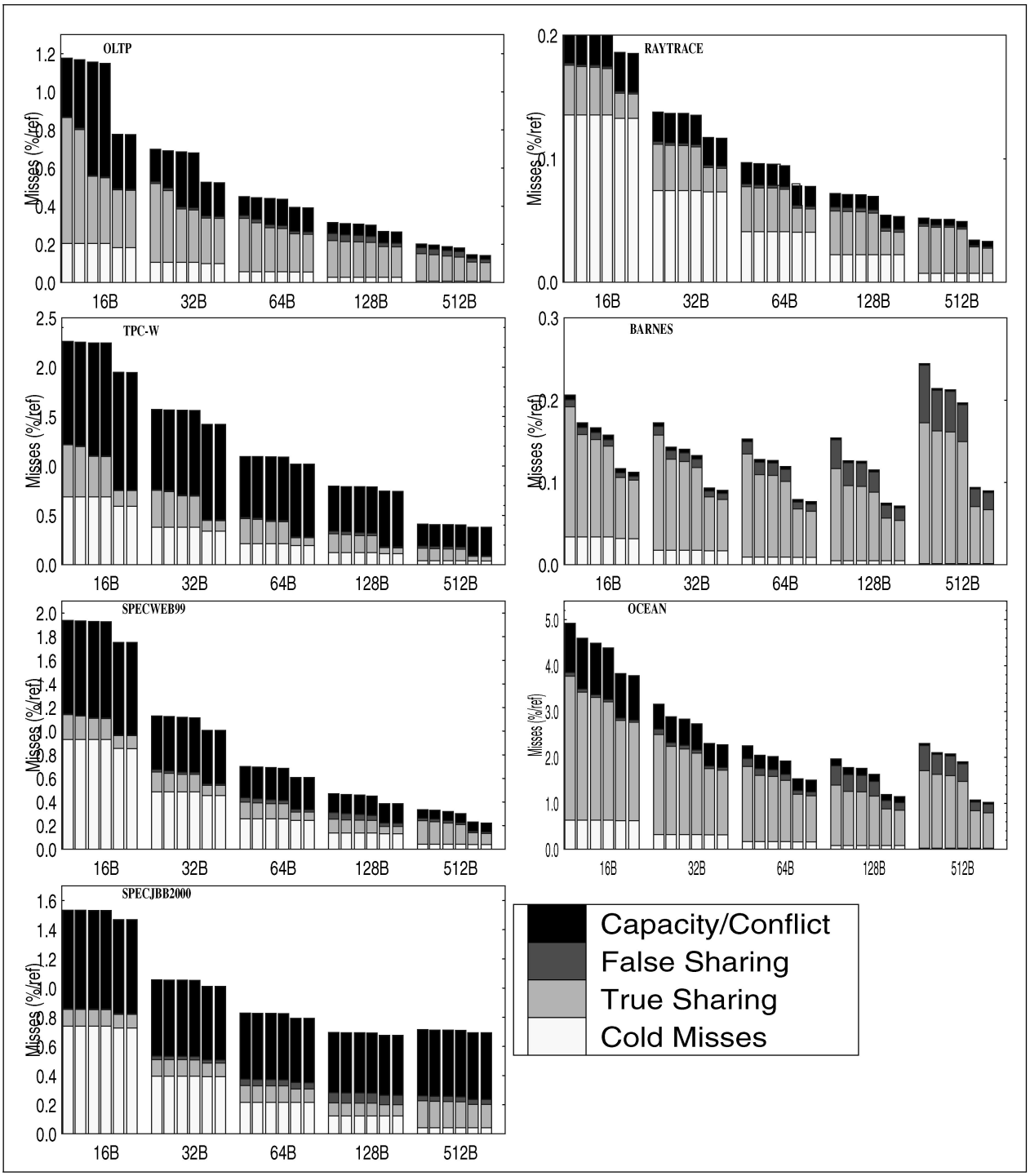


Fig. 16. Multiprocessor miss rates for 1MB 4-way set-associative data cache. Left to right, the stacked bars show cold, true sharing, false sharing, and capacity/conflict misses for the baseline, UFS, UFS-P, MSFS, PSFS, and M/PSFS scenarios.

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.

8 Conclusion

In this work, we explore various aspects of the value locality of store instructions. In doing so, we make six 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 identify and characterize source-level causes of silent stores. Fourth,

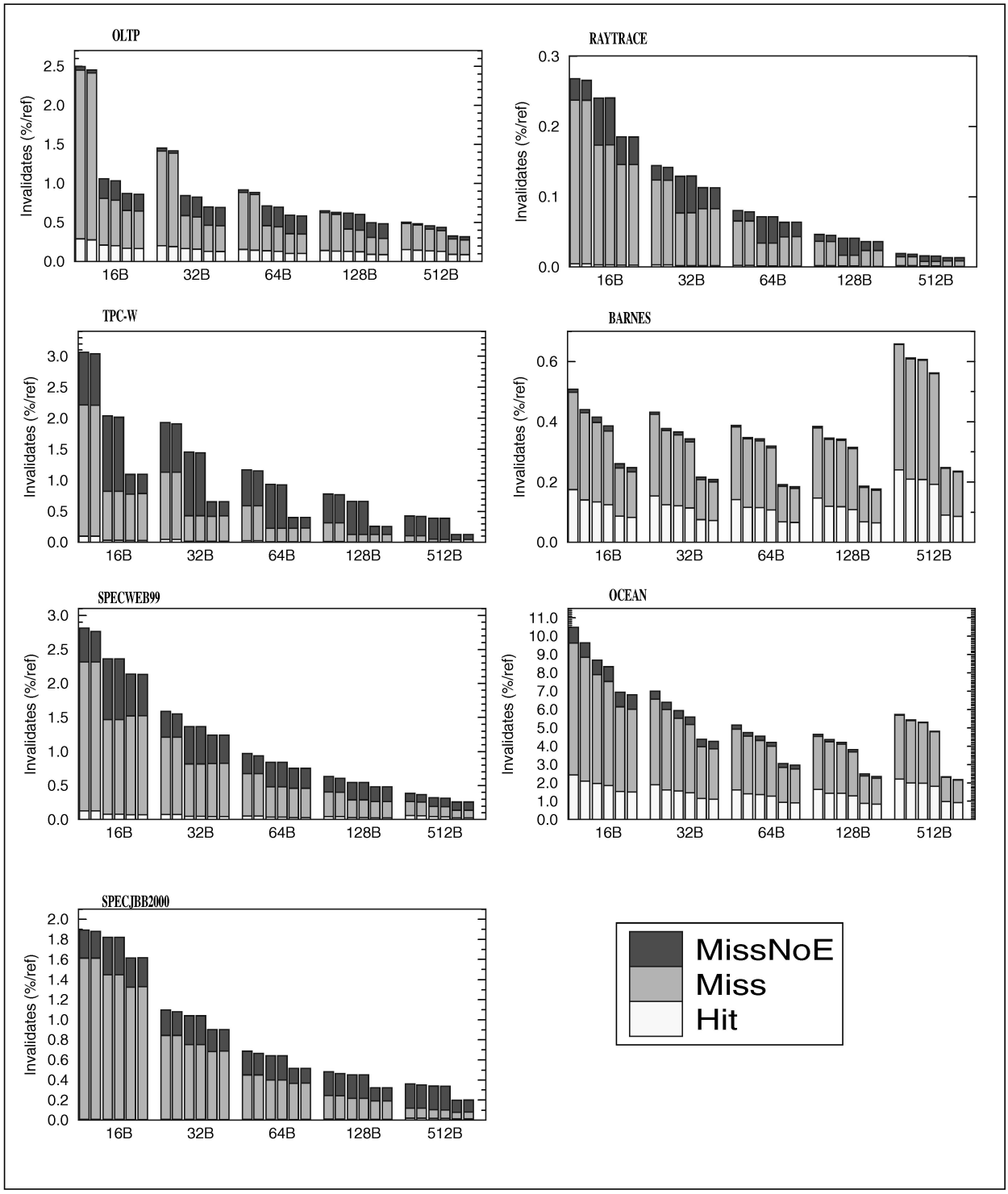


Fig. 17. Multiprocessor invalidates for 1MB 4-way set-associative data cache. Left to right, the stacked bars and data points show invalidate traffic for the baseline, UFS, UFS-P, MSFS, PSFS, and M/PSFS scenarios.

we describe how to enhance the performance of uniprocessor programs by squashing silent stores. Fifth, 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.

ACKNOWLEDGMENTS

This work was supported in part by an IBM University Partnership Award and US National Science Foundation Grants CCR-0073440 and CCR-0083126 and equipment and financial donations from IBM and Intel.

REFERENCES

[1] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* June 1995.

[2] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," *Proc. 26th Int'l Symp. Computer Architecture, Computer Architecture News,* vol. 27, no. 2, pp. 64-75, May 1999.

[3] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proc. 30th ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1997.

[4] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström, "The Detection and Elimination of Useless Misses in Multiprocessors," *Proc. 20th Int'l Symp. Computer Architecture,* May 1993.

[5] J.H. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz 64-Bit Quad-Issue CMOS RISC Microprocessor," *Digital Technical J.,* vol. 7, no. 1, 1995.

[6] S.J. Eggers and T.E. Jeremiassen, "Eliminating False Sharing," *Proc. 20th Int'l Conf. Parallel Processing,* Aug. 1991

[7] J. Gonzalez and A. Gonzalez, "Control-Flow Speculation through Value Prediction for Superscalar Processors," *Proc. Parallel Architectures and Compilation Techniques,* Oct. 1999.

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

[9] T. Keller, A.M. Maynard, R. Simpson, and P. Bohrer, "Simos-ppc Full System Simulator," http://www.cs.utexas.edu/users/cart/simOS, 2001.

[10] M.H. Lipasti, C. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems VII,* Oct. 1996.

[11] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1996.

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

[13] C. Molina, A. Gonzalez, and J. Tubella, "Reducing Memory Traffic via Redundant Store Instructions," *Proc. Int'l Conf. High Performance Computing and Networking,* pp. 1246-1249, Apr. 1999.

[14] A. Moshovos, "Memory Dependence Prediction," PhD thesis, Univ. of Wisconsin, Dec. 1998.

[15] J. Torrellas, M.S. Lam, and J.L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Misses," *Proc. Int'l Conf. Parallel Processing,* Aug. 1990.

[16] Transaction Processing Performance Council, *TPC benchmarks,* http://www.tpc.org, 2001.

[17] K. Lepak and M.H. Lipasti, "Silent Stores for Free," *Proc. 33rd Int'l Symp. Microarchitecture,* Dec. 2000.

[18] K. Lepak and M.H. Lipasti, "On the Value Locality of Store Instructions," *Proc. 27th Int'l Symp. Computer Architecture,* June 2000.

[19] G. Bell, K. Lepak, and M.H. Lipasti, "A Characterization of Silent Stores," *Proc. Parallel Architectures and Compilation Technique,* Oct. 2000.

[20] H. Cain, M. Marden, R. Rajwar, and M.H. Lipasti, "A Characterization of Java TPC-W," *Proc. Int'l Symp. High Performance Computer Architecture,* Jan. 2001.

[21] SPECWEB99 Benchmark Specification, available from http://www.specbench.org, 2001.

[22] SPECJBB2000 Benchmark Specification, available from http://www.specbench.org, 2001.

[23] S.P. Harbison, "An Architectural Alternative to Optimizing Compilers," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 57-65, Mar. 1982.

[24] S.E. Richardson, "Exploiting Trivial and Redundant Computation," *Proc. 11th Symp. Computer Arithmetic,* pp. 220-227, July 1993.

[25] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," *Proc. Int'l Symp. Computer Architecture,* June 1997.

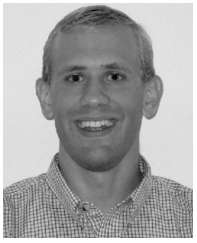
[26] H.-H.S. Lee, G.S. Tyson, and M.K. Farrens, "Eager Writeback—A Technique for Improving Bandwidth Utilization," *Proc. Int'l Symp. Microarchitecture,* Dec. 2000.

[27] A. Yoaz, R. Ronen, R.S. Chappell, and Y. Almog, "Silence Is Golden," *Proc. Work-in-Progress Workshop in conjunction with Seventh Int'l Symp. High Performance Architecture (HPCA-7),* Jan. 2001.

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

[29] S. Kaxiras and J.R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction," *Proc. Int'l Symp. High Performance Computer Architecture,* Jan. 1999.

[30] A. Appel and M. Ginsburg, *Modern Compiler Implementation in C.* Cambridge, U.K., New York: Cambridge Univ. Press, 1998.



Kevin Lepak is a graduate student in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. He received the BS (1999) and MS (2000) degrees in electrical engineering from the same university. His research interests include many aspects of high performance computer architecture and performance modeling, with particular focus on improving memory system performance. He is also interested in VLSI design and EDA. He is a student member of the IEEE.

Gordon Bell is a graduate student in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. He received the MS degree from UW-Madison in 2001 and the BS degree in computer science and engineering from the University of Notre Dame in 1999. His research interests include several areas of computer architecture and I/O performance. He is a student member of the IEEE.

Mikko Lipasti received the MS and PhD degrees from Carnegie Mellon University and the BS degree from Valparaiso University. He is an assistant professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests span multiple aspects of high-performance computer architecture. He is a member of the IEEE and the IEEE Computer Society.