Temporally Silent Stores

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

Recent work has shown that silent stores--stores which write a value matching the one already stored at the memory location--occur quite frequently and can be exploited to reduce memory traffic and improve performance. This paper extends the definition of silent stores to encompass sets of stores that change the value stored at a memory location, but only temporarily, and subsequently return a previous value of interest to the memory location. The stores that cause the value to revert are called temporally silent stores. We redefine multiprocessor sharing to account for temporal silence and show that in the limit, up to 45% of communication misses in scientific and commercial applications can be eliminated by exploiting values that change only temporarily. We describe a practical mechanism that detects temporally silent stores and removes the coherence traffic they cause in conventional multiprocessors. We find that up to 42% of communication misses can be eliminated with a simple extension to the MESI protocol. Further, we examine application and operating system code to provide insight into the temporal silence phenomenon and characterize temporal silence by examining value frequencies and dynamic instruction distances between temporally silent pairs. These studies indicate that the operating system is involved heavily in temporal silence, in both commercial and scientific workloads, and that while detectable synchronization primitives provide substantial contributions, significant opportunity exists outside these references.

1 INTRODUCTION

There is widespread agreement that communication misses are one of the most pressing performance limiters in shared-memory multiprocessor systems running commercial workloads. For example, both Barroso et al. [3] and Martin et al. [18] report that about one-half of all off-chip memory references are communication misses; i.e. the references are satisfied from dirty lines in remote processor caches. Communication misses are caused by remote writes to shared cache lines; in single-writer or invalidate protocols a write requires all remote copies of a shared line to be invalidated [9]. Subsequent references to those remote copies lead to misses that must be satisfied from the writer's cache. Two current trends are likely to exacerbate this problem: systems that incorporate an increasing number of processors will likely lead to an increased probability that a remote write to a shared line will occur; and systems with larger and more aggressive local cache hierarchies that eliminate most capacity and conflict misses, but cannot reduce communication misses.

Prior work has shown that many communication misses can be avoided by detecting false sharing [10] or silent stores [16]. These approaches attempt to extend the lifetime of a shared copy of a cache line by monitoring remote writes and determining whenever the remote write either changes a different word in the line or does not change it at all. In this paper, we introduce a new method for extending cache line lifetime that relies on the temporal behavior of writes to the same location. Specifically, we exploit a program characteristic which we call *temporal silence*. Informally, temporal silence describes the net effect of two or more writes that change a register or memory location to an intermediate value, but subsequently revert the location to a previous value of interest. In this paper, we focus on exploiting the temporal silence of store operations in multiprocessor systems for the purpose of eliminating communication misses.

The phenomenon of temporal silence is not unique to multithreaded programs running on multiprocessor systems, and has been exploited (although not explored in depth) in earlier work on speculative multithreading. For example, compiler-based Thread-Level Speculation (TLS) systems optimistically parallelize programs and rely on hardware support in the coherence protocol to detect memory dependence violations. In recent work, Steffan et. al [21] and Cintra et. al [7] examine different methods of improving memory value communication in such a system. The load value prediction approach they describe, when used in concert with a last-value predictor, effectively exploits temporal silence of intervening stores to the loaded memory location: otherwise predicting the last value observed for that memory location will surely fail. In addition, architects have exploited temporal silence of register values. As one example, Dynamic-Multithreading [1], exploits temporal silence of register writes on procedure calls and returns for callee saved registers by predicting that register values used by a speculative thread are those values which exist at the point the speculative thread is spawned. In other related work, Speculative Lock Elision detects and exploits temporal silence of "silent store- pairs"[20].

In multiprocessor systems, we find that a surprising number of writes that induce communication misses are temporally silent. We find that an idealized scheme for exploiting temporal silence can remove up to 45% of sharing misses in the commercial workloads we study. Further, we propose a realistic, non-speculative method of exploiting temporal silence that captures nearly all of this opportunity, eliminating up to 42% of such misses.

The remainder of this paper describes the phenomenon of temporally silent write pairs in multiprocessor systems, defines *temporal silent sharing* (Section 2), quantifies its contribution to communication misses in a shared-memory multiprocessor (Sec-

ASPLOS X 10/02 San Jose, CA, USA

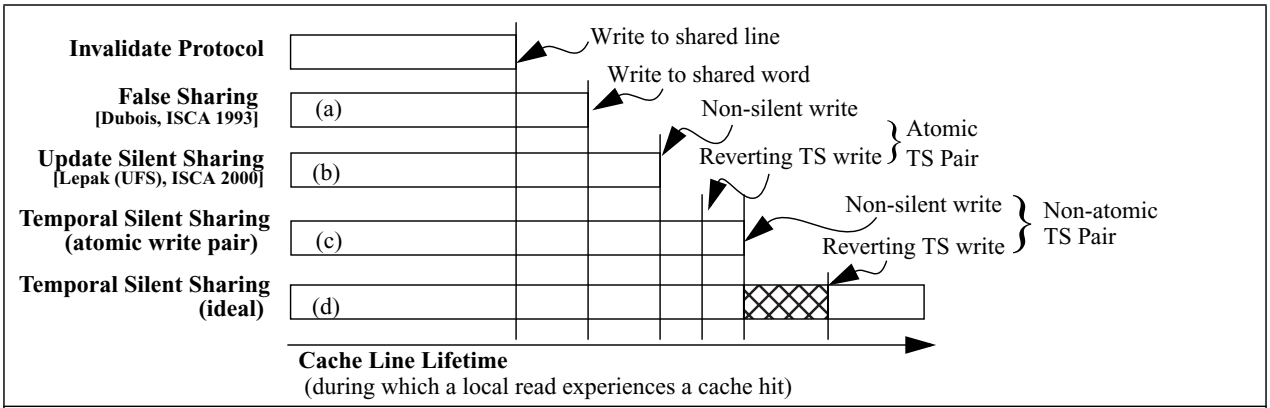

FIGURE 1. Extending the Lifetime of a Shared Cache Line. Lifetime is extended by avoiding writes to other parts of the word (false sharing), ignoring silent writes (update silent sharing), ignoring temporally silent atomic write pairs, and reinstating a line upon reversion due to a temporally silent write pair that is not atomic.

tion 3), and characterizes the phenomenon of temporal silence by examining value distributions, contribution of atomic primitives, and kernel, library, and user code (Section 4). We also discuss and evaluate several implementation issues related to detecting (Section 5) and exploiting this new program characteristic non-speculatively with changes to an existing MESI coherence protocol (Section 6). We do not report direct performance benefit due to the fact that the commercial workloads we study are nondeterministic and hence difficult to measure precisely [2]. Rather, we show dramatic reductions in communication misses and argue that substantial performance benefit will result. This argument is consistent with prior results that show the substantial performance impact of communication misses [3,18]. Detailed evaluation of performance benefit will be reported in future work.

2 DEFINITIONS

2.1 Temporally Silent Stores

In order to provide context for understanding temporally silent stores, we need to review the definition of silent stores presented in [16] briefly. In that work, we contributed the insight that many memory writes do not change the value stored at a memory location (they do not *update* system state) because the write value matches the value stored at the memory location. Such writes are called *(update) silent stores*.

In Section 1 we informally introduced the concept of temporal silence; the idea that a value written matches a previous value seen at that location. Since our focus in this work is eliminating communication misses, we only consider temporal silence of memory locations. Therefore, strictly speaking, we consider a store temporally silent if it writes a value to a memory location which has ever existed there previously. With this definition, consider a pathological case in which the same byte in memory is incremented repeatedly. After 256 writes of the location, all subsequent stores to the location become temporally silent, just with respect to a different previous value. Obviously such a definition does not have much practical value. Therefore, a more useful definition of temporal silence depends on the desired application.

In a multiprocessor system, arguably the most interesting temporally silent stores are the ones that cause a memory location to revert to a value that has been previously observed by a remote processor. In this context, we consider a store temporally silent if it writes a value matching a stale value in a remote processor cache that is currently in invalid state. These temporally silent stores change the location back to a value recently observed by the other processor. We can imagine this value can be communicated at low cost. It will become clear in the following sections when discussing multiprocessor sharing why we choose such a definition. Note that there is nothing in our subsequent definitions which precludes multiple (distinct) intermediate values or temporally silent values for a given memory location; bounding the values is simply a matter of practicality. We use this definition of temporally silent store throughout this work.

As we have discussed, temporal silence has an important difference over update silence; namely an intermediate value which is non-silent. Therefore, to succinctly describe specific dynamic stores of interest, we define a *temporally silent store pair* to consist of two parts: A non-silent store to the memory location, called the *intermediate value* store, which visibly changes the system state from a previously observed value; and a non-silent store, called the *temporally silent* store, which then reverts the state back to a previously observed value. Note that there may be additional intervening stores to the same address; we do not consider these part of a temporally silent store pair.

2.2 Reducing Multiprocessor Sharing

Communication in multiprocessor systems can be reduced by monitoring writes to shared cache lines more closely and detecting false sharing [10] or silent stores [16], thereby extending the lifetime of a cache line in remote caches. These two methods of extending cache line lifetime are depicted graphically in Figure 1(a) and Figure 1(b).

Along the same lines, we can exploit temporal silence to further increase cache line lifetime. Two scenarios of interest are indicated in Figure 1(c) and Figure 1(d). In scenario (c), the cache line is written with an intermediate value store, but a temporally silent store follows and reverts the line to a previous value. Memory consistency rules allow the stores to be collapsed into a single atomic event that is now effectively silent, since the memory location contains the same value before and after the store pair has executed. Therefore, we refer to this temporally silent store pair as an *atomic temporally silent store pair*. We find that capturing such temporal silence by delaying writes (hence exploiting PowerPC weak ordering) is relatively ineffective, leading to a maximal 3% reduction in sharing misses. Speculative techniques for ensuring atomicity, similar to [20], could improve that result, but since this paper focuses on nonspeculative tech-

Program	Instr.	Loads	Stores	Silent Stores	Temporally Silent Stores	Description
barnes	1.76B	508M	304M	117M (39%)	3.4M (1.1%)	SPLASH-2 N-body simulation (8K particles)
ocean	561M	177M	36M	8.0M (22%)	1.8M (5.4%)	SPLASH-2 Ocean simulation (258x258 ocean)
radiosity	2.43B	763M	326M	138M (42%)	1.1M (0.34%)	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)
raytrace	414M	254M	45M	17M (38%)	1.2M (2.6%)	SPLASH-2 Raytracing application (teapot)
specweb	4.60B	1.13B	624M	233M (37%)	23M (3.7%)	Commercial Web-Serving application
specjbb	3.58B	964M	431M	152M (35%)	22M (5.1%)	Commercial Server-Side Java application
tpc-w	1.41B	1.40B	340M	231M (68%)	9.2M (2.7%)	Commercial 3-Tier Web-Based OLTP application

Table 1: Benchmark Characteristics and Description. Instructions are measured excluding the operating system idle loop.

niques, we only consider the final scenario indicated in Figure 1(d).

In scenario (d), the cache line is written with an intermediate value and consistency rules force this store to be ordered; hence there is a time window when the previous copy of the cache line is invalid. Later, a temporally silent store occurs causing the current copy to match a previous copy, extending the cache line lifetime further. We can exploit this occurrence in many cases by augmenting an existing MESI coherence protocol, as described in detail in Section 6.

2.3 Temporal Silent Sharing (TSS)

Until recently (e.g. [16]), prior work in defining sharing has focused on the address of shared data and whether or not it had ever been written by another processor. In [16], we modified the definition of sharing to consider the value locality of stores and indicate that silent stores can be exploited to extend cache line lifetimes of remote cache lines (Figure 1b) by avoiding invalidation of these lines. We call this definition of sharing Update Silent Sharing (USS).[1]

Having defined temporally silent stores in multiprocessors, we can now rigorously define sharing considering these stores. We can effectively accomplish this by simply redefining which misses are Essential Misses in Dubois' classification scheme [16]; i.e. those misses which communicate meaningful, new, information about system state. Informally, we extend the definition of Essential Miss presented in [16] to capture the case where cache line values revert to the value previously observed by another processor in the system (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) *a different value than the last value observed by that processor for that block* since the last essential miss to that block, it is an essential miss.

In our new definition of Essential Miss, we establish that the **net effect** of all writes to the location of interest since a processor's last observation of the location must constitute new system state for the miss to be essential. Note that this definition makes no statement about how many distinct processors have written to a specific word (with intermediate value or temporally silent stores) or other places within the cache line--it only requires that the value of interest be the same as the last value observed for a given processor. The distinction will become clear as we describe different methods of capturing temporal silence in the following sections. Also note that our new definition of temporal silent sharing (TSS) is a superset of USS.

1. Previously (i.e. [16, 17, 4, 15]), we called this Update False Sharing (UFS). However, since the term "False Sharing" implies unnecessary communication due to cache block granularity [11], we choose instead to use "Silent Sharing"; communication eliminated through store value locality is due to properties of the data itself, not its spatial layout.

3 REDUCING DATA TRAFFIC WITH TSS

We have argued in Section 2.2 that exploiting temporal silence can extend cache line lifetime. In this section, we quantify the potential benefit of exploiting temporal silent sharing for reduction in data traffic in snoop-based multiprocessor systems.

3.1 Experimental Setup

We study a 4 processor, snoop-based system using the SimOS-PPC [13] full system simulator with 64B cache lines, running AIX 4.3.1. We collected data for 128B and 256B cache lines as well, but we do not present it for the sake of brevity and because it does not differ materially from the 64B case. When presenting characterizing data in subsequent sections, we assume infinite caches unless stated otherwise so that observations about intrinsic program characteristics are not skewed by finite cache effects. The processor core used to drive timing for the cache studies is similar to a single-threaded IBM RS64-III (Pulsar) [5] used in IBM S85 server systems. It is in-order with 128KB each split L1 instruction and data caches, and 16MB inclusive L2, with latencies of one and seven cycles, respectively. We model a perfect L1 cache CPI of 1.0, which is very close to the measured CPI on real hardware for a similar system for these workloads [6]. We simulate four benchmarks from the SPLASH-2 [22] suite for entire runs of reduced input sets and three, three-second snapshots of commercial workloads described in [6]. Table 1 summarizes workload characteristics.

3.2 TSS Limit Results

In Figure 2 we present infinite and finite cache miss rates for our new definition of sharing as compared to USS,[2] and also Dubois' definition [10], labeled "Baseline". The stacked bars (normalized to baseline, infinite cache) indicate the contribution of cold, true sharing, false sharing, and capacity/conflict misses for finite caches of decreasing sizes (16MB, 8MB, and 4MB). Focusing first on the infinite cache results, we see that TSS can reduce the overall miss rate for infinite caches by up to 33% and

2. Note that in this work, USS corresponds to UFS-P in [16], i.e. we perform store squashing for cache misses with perfect knowledge of store silence. This leads to maximal data traffic reduction for USS, providing the best possible results for comparison.

27% over the baseline and USS, respectively (in *specweb*). The harmonic mean reduction across the scientific benchmarks (SPLASH-2) is 15% and 12%, respectively, and for the commercial workloads, 25% and 21%.

In the case of finite caches, the relative reduction in overall miss rate is smaller because of two effects: 1) Additional misses created by pure capacity/conflict misses; 2) Creation of additional capacity/conflict misses due to fewer "invalidated" lines, and therefore a larger working set (as explained for USS in [16]). However, for all finite caches, we see the reduction in misses tracking the infinite cache reductions in absolute number of misses prevented. The only notable exception is in *tpc-w* for 4MB caches, where the reduction in overall miss rate drops from over 20% to less than 6% (normalized to misses in the baseline, infinite cache case). This data indicates that studies conducted throughout the rest of this paper with infinite caches are reasonable, since finite caches do not grievously affect the absolute number of misses prevented under TSS. Therefore, we focus on infinite cache studies in the rest of the paper.

We can see in Figure 2 that the contribution of cold misses is substantial for many of the workloads, largely due to the limited amount of real time that we can practically simulate. Due to this, we expect the relative fraction of cold misses to be smaller in a real system. Because exploiting TSS in SMPs will not affect cold misses, we focus solely on communication misses throughout the rest of this work.

Examining the communication miss component only in Figure 2, we see up to 45% and 35% reductions over the baseline and USS cases (in *specweb*), respectively. We see harmonic mean reductions in the scientific benchmarks of 24% and 19%, respectively; 42% and 32% for the commercial workloads. Most of the improvement is in true sharing, although some false sharing is also eliminated. Interestingly, TSS provides substantial benefit over USS, particularly in the commercial workloads. We will explore possible explanations for this throughout subsequent sections. Given that communication misses are a limiting factor in the performance scalability of large-scale SMPs (due to both latency and data bandwidth [3],[18]) attacking these misses is very important.

4 UNDERSTANDING TEMPORAL SILENCE IN SMPs

We showed in the previous section that the potential reduction in communication misses by exploiting TSS is large. In this section, we focus on characterizing and understanding the temporal silence phenomenon and its interaction with application code and the operating system. This will allow us to develop a better understanding of temporal silence before detailing a method of exploiting it in the sections that follow.

4.1 Temporally Silent Program Behavior

To determine program structures and behaviors exhibiting temporal silence, we augmented our simulator to detect instructions which contributed to TSS. This allows us to better understand the occurrence of temporal silence at the program level.

In Table 2, we show the contribution of different functions within user level code, shared library code, and kernel code. The "% TSS Misses" column indicates the percentage of TSS avoidable misses the given function participates in by contributing an intermediate value store or temporally silent store. Percentages

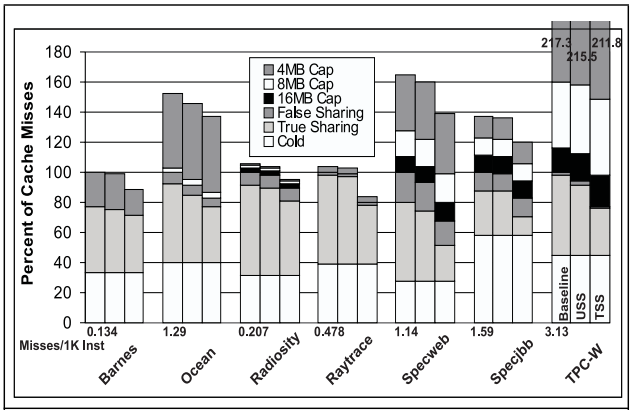

FIGURE 2. Percentage of Cache Misses for Different Definitions of Sharing. The data is normalized to the Baseline case for 64B cache lines and infinite caches. The top three bars show cumulative additional capacity/conflict misses for 8-way set-associative caches of 16MB, 8MB, and 4MB**.**

within this column may add to greater than 100% because multiple functions may participate within a single TSS avoidable miss (e.g. if a separate function is used for lock acquire and release, a single miss due to this temporally silent pair will be counted in this column for both the acquire function and release function).

The "% Dynamic TSS Stores" column indicates the percentage of all dynamic stores (either intermediate value store or temporally silent store) contributing to TSS avoidable misses that occur within the given function. Because this column is normalized to the total number of dynamic stores contributing to TSS avoidable misses, this column will add up to 100% (or less). However, the total is always less than 100% when functions are considered individually because it is only practical to present data for an interesting subset of functions.

Because we do not have source code for the shared libraries of interest or the AIX v4.3.1 kernel, it is difficult to discern exactly which program semantics are causing temporal silence to occur. However, when possible, we provide function names to allow reasonable conjectures to be made.

Focusing first on the scientific workloads, it is interesting that most locking-related TSS avoidable misses in *ocean* are not within the application itself, but rather, in kernel support routines. The contribution of application spin-lock acquires/releases is less than 8%. In *barnes* (representative of *radiosity* and *raytrace*), the contribution of user-level spin locks is large (over 80%). However, substantial contributes within the AIX kernel (14.5%) are still noted. Furthermore, user-level code not directly related to atomic primitives contributes less than 5% and 6% in each benchmark. This data indicates that studying temporal silence without considering operating system and library code, even for scientific workloads, may ignore substantial opportunity.

In the commercial workloads, most TSS avoidable misses in *specjbb* occur within the Java runtime environment or JRE (*libjava.a* and *libjitc.a*). As shown in Figure 4, it appears that non-trivial sharing misses related to atomic as well as data operations occur within the JRE. In *specweb*, many temporally silent stores occur within process management, TLB, and page table-related code. This seems reasonable given the structure of this implementation of *specweb* which spawns a new perl process for each incoming web request, causing frequent process creation and destruction [6]. Results for *tpc-w* were qualitatively similar to *specjbb* and *specweb* and are omitted for brevity..

Benchmark	% TSS Misses	% Dynamic TSS Stores	Function	Comments
specjbb	40.0%	13.9%	check_lock (K)	Compare and swap with import fence (lock acquire) primitive
specjbb	36.5%	38.5%	libjava.a (L)	Java Runtime Environment
specjbb	22.4%	8.9%	libpthreads.a (L)	Thread management
specjbb	17.7%	8.5%	clear_lock (K)	Atomic write with export fence (lock release) primitive
specjbb	16.9%	18.5%	libjitc.a (L)	Java Runtime Environment--JIT
specjbb	0.0%	0.0%	all user code (U)	All user-level application code
specweb	40.2%	20.9%	rsimple_lock (K)	Recursive simple lock acquire
specweb	20.1%	3.8%	simple_lock (K)	Non-recursive simple lock acquire
specweb	20.0%	4.1%	simple_unlock (K)	Simple lock and recursive simple lock release
specweb	19.4%	1.9%	p_inspte_p64 (K)	Process creation/deletion, page table entry insert
specweb	14.7%	1.2%	v_inspft (K)	AIX kernel
specweb	13.7%	1.5%	v_lookup (K)	AIX kernel
specweb	13.2%	1.3%	delall_pte_p64 (K)	Process creation/deletion, page table entry delete
specweb	11.9%	2.6%	invtlb_ppc (K)	Process creation/deletion TLB manipulation/invalidation
specweb	11.8%	1.0%	v_delpft (K)	AIX kernel
specweb	6.2%	0.9%	px_rename_p64 (K)	AIX kerne)
specweb	5.9%	1.0%	rsimple_unlock (K)	Recursive simple lock release
specweb	4.8%	0.6%	v_scoreboard (K)	AIX kernel, includes v_descoreboard
specweb	3.0%	0.4%	insque/remque (K)	Shared queue management
specweb	0.3%	0.1%	all user code (U)	All user-level application code
ocean	70.6%	(*)53.7%	all kernel locks (K)	All kernel level locks/releases, not within application code
ocean	46.8%	14.9%	simple_lock (K)	Kernel lock acquires, not within application code
ocean	42.6%	13.2%	simple_unlock (K)	Kernel lock releases, not within application code
ocean	15.2%	14.0%	rsimple_lock (K)	Kernel lock acquires, not within application code
ocean	11.5%	1.9%	unlock_enable (K)	Kernel lock releases, not within application code
ocean	9.0%	1.4%	test_and_set (K)	Other atomic primitives, not called directly by application code
ocean	7.7%	1.9%	cs (K)	Atomic compare and swap (application code lock acquire)
ocean	7.6%	1.8%	user mode store (U)	Application code lock release
ocean	7.0%	4.9%	state_save_point (K)	AIX kernel
ocean	5.6%	3.4%	v_lookup (K)	AIX kernel
ocean	5.5%	1.6%	v_inspft (K)	AIX kernel
ocean	5.3%	3.1%	p_inspte_p64 (K)	AIX kernel
ocean	4.5%	3.5%	call_dispatch (K)	Kernel thread dispatching/scheduling
ocean	4.4%	1.7%	set_curthread (K)	Kernel thread dispatching/scheduling
ocean	4.2%	1.0%	other user code (U)	All user-level application code (not lock releases)
barnes	80.9%	6.8%	cs (K)	Atomic compare and swap (application code lock acquire)
barnes	80.0%	6.9%	user mode store (U)	Application code lock release
barnes	9.8%	4.3%	Kernel Locks (K)	simple_lock(), simple_unlock(), disable_lock(), unlock_enable(); not within application code
barnes	4.7%	3.0%	Other kernel (K)	Process creation/deletion and thread management
barnes	5.7%	78.6%	other user code (U)	All user-level application code (not lock releases)

Table 2: Functions Actively Participating in Temporal Silence. The table indicates the percentage of dynamic intermediate value and reversion stores contributed within the specified functions in the benchmarks indicated for cases of useful TSS. (K) Denotes kernel functions, (L) denotes library functions, and (U) denotes user code. (*) Includes contributions from multiple functions listed individually within the table as well, so for *ocean* this column adds to more than 100%.

We also note that the two metrics presented (% TSS Misses and % Dynamic TSS Stores) do not always correlate strongly with one another. In some cases, many dynamic stores are contributing to relatively few TSS avoided misses, while in other cases the converse is true (most notably *barnes* in “other user code” and “cs”, respectively). Therefore, if our goal is eliminat-

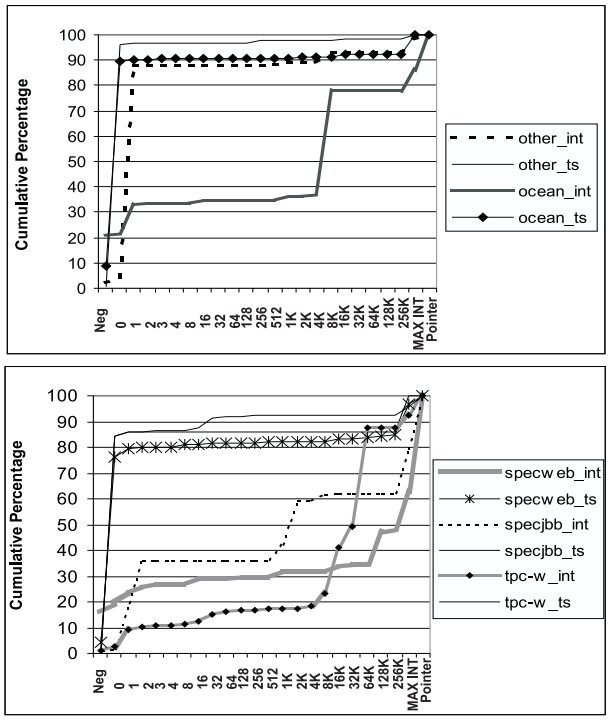

FIGURE 3. Value Cumulative Distributions for Useful TSS. The figure indicates the cumulative distribution of observed intermediate values and temporally silent values for TSS avoidable misses. Only values for integer stores are shown--contributions from floating point stores were measurable, but negligible. Results for *barnes, radiosity,* and *raytrace* do not differ materially and are represented by the "other_int" and "other_ts" categories in the scientific workload graph**.**

ing communication misses, it may not be prudent to develop mechanisms which target each dynamic temporally silent store with equal effort and assume that a corresponding reduction in TSS avoidable misses will occur.

4.2 Value Distribution

The values observed for both intermediate values and temporally silent values of memory words may reveal insight into temporal silence. In Figure 3, we show the cumulative distribution of temporally silent values and intermediate values for TSS avoidable misses. Across both the scientific and commercial applications, the graph indicates over 75% of temporally silent values are zero, which is not surprising given similar results were observed for update silent stores [4]. However, in the commercial workloads, an observable fraction (over 5% in *tpc-w* and *specweb*) of temporally silent values are non-null pointers[1], a point which we will return to shortly.

Examining the intermediate value distributions for the scientific applications, we see (with the exception of *ocean*) the predominant intermediate value is integer one, which is caused by user-level spin-locks and other flag values. In *ocean*, the largest contribution comes from values in the range 4K-8K, which is unexpected. We examined this benchmark further, and found

1. We approximated pointer values by storing all virtual addresses touched by each process and assuming any observed value which matched a previously observed virtual address was in fact a pointer.

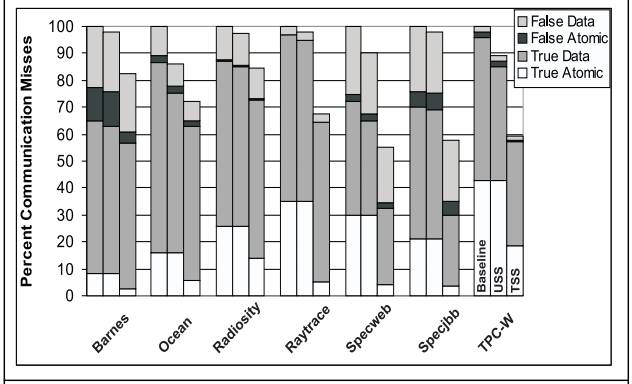

FIGURE 4. Contribution of Atomic Operations Under Different Definitions of Sharing. The figure indicates the contributions of cache lines written with at least one stwcx/ stdcx (store-conditional) operation to sharing misses for Baseline, USS, and TSS. The data is normalized to sharing misses in the Baseline case**.**

these values are actually the thread IDs used by AIX for the concurrent threads of *ocean*. These intermediate values still appear to be lock-related, with AIX using the thread ID to indicate which thread is holding the lock to thread-safe memory allocation routines and other operating system structures protected with the *simple_lock(), simple_unlock(), disable_lock(),* and *unlock_enable()* routines. These are higher level locks (not simple spin-locks) provided by the AIX kernel [8]. In the commercial applications, the intermediate value distributions show strong contributions throughout the range, many of which match thread IDs of running processes (in the range 512-64K). This enlarged contribution from high-level locks in the commercial applications is reasonable; under a highly concurrent commercial application load we expect fewer simple spin-locks because they can reduce system throughput by wasting processor cycles spinning. More noteworthy is a particularly strong contribution for intermediate pointer values. The contribution of these values is over 40% in *specweb* and 20% in *specjbb*. Many of these revert to null (and therefore are counted under temporally silent value zero), but some also revert to non-null values, indicating that temporally silent pairs also occur in what are likely shared data structures.

4.3 Explicit Atomic Operations

Lock variables are great candidates for exhibiting temporal silence since they revert to their unheld value when released. Indeed, Speculative Lock Elision exploits "silent store-pairs" (which we refer to as atomic temporally silent store pairs) to elide transfers and execute critical sections concurrently [20]. However, we found that memory locations written with explicit ISA atomic operations (i.e. store-conditionals) are not the only contributors to TSS by measuring how much of the benefit of TSS comes from cache lines touched with such operations, and how much benefit is contributed by data operations.

In Figure 4 we show the contribution to sharing misses by atomic operations for Baseline, USS, and TSS. We determine explicit atomic operations in a very liberal manner: if any location within a cache line has been written with a PowerPC stwcx/ stdcx (store-conditional) instruction, we denote a subsequent miss to that cache line as related to an explicit atomic operation (in reality, there may also be data operations within the same line). Load-linked/store-conditional instructions can implement various atomic primitives, including locks, atomic updates, com-

pare-and-swap, atomic list insertion/deletion, and others [9]. Determining which atomic primitive is implemented is non-trivial without instrumenting the binary at compile time. Since most temporally silent pairs occur within the AIX kernel and within the various libraries which make up our commercial workloads, we are unable to further classify atomic primitives.

Examining the figure, we see that a large fraction of true sharing misses are due to atomic primitives across all benchmarks in the Baseline case, ranging from 9% in *barnes* to 43% in *tpc-w*. We also observe that over 75% of TSS avoidable misses are from atomic primitives except in *specjbb*, where the fraction contributed by atomic primitives is 45%. This data indicates we may be able to leverage explicit atomic operations to more efficiently exploit temporal silence. However, in *specjbb*, a majority of TSS avoidable misses are not due to atomic primitives, indicating a general mechanism—one not focused on just these constructs—is desirable. We focus on such a general mechanism throughout the subsequent sections.

Finally, note that even for TSS avoidable misses which can be determined to be locks with high likelihood (see Table 2, Figure 3, Figure 4), the value distributions in Figure 3 indicate the transfers avoided predominantly indicate the lock is free. This implies that eliminating these misses improves synchronization performance.

5 DETECTING TEMPORAL SILENCE

In this section, we outline methods of detecting when temporal silence occurs within various processor structures relating to the memory hierarchy. All of the methods for detecting temporal silence outlined in this section can be useful, depending on the distance between the intermediate value store and temporally silent store which comprise a silent pair. We will return to this point throughout Section 6.

5.1 Inside the Processor Core

If we assume a processor that implements update-silent store squashing (i.e. issuing a load operation, a comparison of the value to be written against the previous value, and a conditional store depending on the silence outcome) as in [16], we can augment the load/store queue (LSQ) to keep the load data for a verified store in the LSQ [17]. Then detecting temporal silence simply involves checking any new store data values against the system visible value for that location (i.e. the first loaded value for the location of interest in the LSQ). Any stores which become temporally silent in the queue can be dropped[1], while all other stores are performed as usual.

Another method involves augmenting a traditional merging write buffer or write cache [14, 19] to perform the same function. An advantage to using this structure is that we need not perform traditional update-silent store squashing to exploit temporal silence if the structure is write-allocate. Write-allocate structures are common because they simplify write handling into the level behind them. In this case, when a committed store allocates an entry in the write buffer, it can keep a copy of the system-visible (stale) version of the buffer along with the merge (dirty) data. As stores are retired, they can be incrementally checked against the stale data for temporal silence. If temporal silence is detected for the entire buffer entry (i.e. the dirty data matches the stale data), the buffer entry can be freed with no write action. Even though we are doubling the amount of data storage for each write buffer entry in a naive implementation, the size of the buffer itself will not double because tags and most of the datapath can be shared between the dirty data and stale data. Preliminary data (presented in Section 6.3) also indicates that only a few words per cache line need to be tracked; however, we leave detailed exploration of this design space to future work.

1. Provided memory ordering rules allow the stores to appear atomic as described in Section 2.2.

5.2 Outside the Processor Core

Once store operations have become non-speculative (i.e. leave the processor core), they must become visible to the rest of the system to maintain correct memory reference ordering. The approaches described in the previous section are only capable of capturing a short distance between the intermediate value store and the temporally silent store due to relatively small buffering capabilities within the core. Detecting temporal silence outside the processor core will allow us to capture a longer dynamic program distance between the temporally silent pair than is feasible with the techniques presented previously. We can then use a new coherence event (Section 6.3) to communicate non-speculatively when it is safe to use a previously seen version of a cache line, because it has reverted to a previous value. We will show in Section 6.4 that in many cases there is more distance between the temporally silent pair than may be reasonably buffered inside a processor core, motivating this decoupled approach.

One method of detecting temporal silence outside the processor core exploits the natural behavior of inclusive cache hierarchies. Consider a writeback L1-D cache with an inclusive L2. When the processor writes a cache line, the line is brought into the L1-D and is written while the L2 updates its directory to indicate the line is modified in the L1-D. Note that the L2 data is actually a stale version of the cache line, and we can use it for detecting temporal silence. This technique can arbitrarily delay the detection of temporal silence; if the delay exceeds the time to a remote read, there is no benefit. Section 6.5 shows that in most cases, a reasonable delay is acceptable. This idea--exploiting the natural versioning that occurs in a cache hierarchy--can be generalized to lower levels in the hierarchy.

A more timely way to detect temporal silence involves augmenting each level in the memory hierarchy with explicit stale value storage. Whenever a system-visible value is overwritten in the cache, the stale value is saved, and the new value is written into the cache. When a new write occurs, the new write value is both written into the cache and compared against the stale value. If the new write matches the stale value, temporal silence for that write has occurred. If all words within the cache line match the stale version (either because they have never been written or because they have all become temporally silent), the entire cache line has become temporally silent. This scheme requires extra storage for stale values and can complicate cache write timing paths. However, in many cache designs, to ease circuit and datapath design and provide ECC protection, writes effectively become read-modify-writes. Clearly, a RMW operation can be modified to save the stale value (as in [17]).

6 COHERENCE SUPPORT

Whenever temporal silence is detected, it needs to be communicated to other processors to be able to avoid sharing misses. In this section, we describe a new coherence protocol which allows such communication without employing speculation.

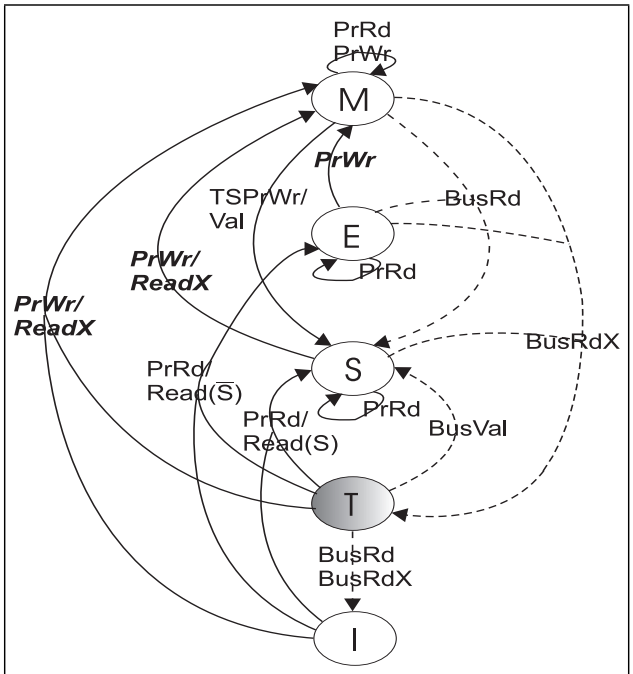

FIGURE 5. State Machine for the MESTI Protocol. We augment MESI (using the notation from [9]) with temporal silence support by adding a "temporally invalid" (T) state. "Val" denotes the added validate transaction required for MESTI to communicate the occurrence of temporal silence against the previous globally visible value for a cache line, and the italicized/bold PrWrs indicate where a stale version which can be reverted to is saved. To simplify this diagram, we assume a processor performing USS-P silent store squashing as described in Section 3.2 and [16] will issue PrRd/Read transactions for silent store misses.

6.1 Coherence Support for Temporal Silence

In Figure 5, we show the additional coherence support we propose which allows intermediate values to be ordered but then reverted (as outlined in Section 2.2). We call our proposed protocol "MESTI", which adds the temporally invalid state "T". This state is entered upon receipt of an invalidate from another processor in the system if the data was previously valid (M, E, or S states). If another bus transaction occurs to a line in T state, it transitions to I state, unless the transaction is a validate, in which case it transitions to S state. Entering T state allows remote processors to save the last globally visible copy of a cache line so it can be reverted to later. The validate transaction allows a remote processor to validate a remote cache line when temporal silence against the previous globally visible value is detected. In Figure 5, we indicate the PrWrs in bold/italic text when a globally visible version of the cache line is saved; this enables a subsequent validate transaction if the line reverts to this version.[1]

A validate effectively places a cache line back into remote caches with simply an address transaction, as opposed to sending new data, as is done in update protocols. However, address traffic may not be equal to an update protocol because a validate is triggered only when the final temporally silent store occurs to the cache line; multiple intervening stores not part of the temporally silent pair or other temporally silent stores to different locations within the cache line do not cause validates.

1. We assume a write-allocate cache to save the previous globally visible version for the I to M and T to M transitions. Note that the version saved here is the data arriving from the system before the line is modified.

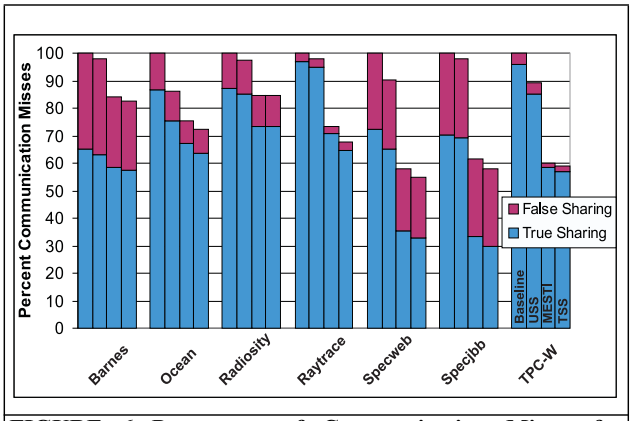

FIGURE 6. Percentage of Communication Misses for Different Definitions of Sharing. The data indicates the number of communication misses normalized to the Baseline case for 64B cache lines and infinite caches, including the reduction in sharing with our MESTI protocol.

We refrain from giving a rigorous validation of the correctness of the MESTI protocol in this work and prefer to focus on its ability to exploit TSS. However, the protocol change we propose is inherently simpler than other protocol changes because T state is only used for performance optimization and not for correctness. In the case of any subtle protocol races, T state can simply be handled equivalently to I state, and we can regress to the MESI protocol.

6.2 Temporal Silence Captured

In order to show the maximal data traffic reduction possible with our MESTI implementation, we model enough stale storage throughout the processor core and memory hierarchy to detect all cases of TSS which MESTI can exploit. Principally, this means augmenting traditional cache structures with stale storage of 64B per cache line, matching the cache line length. This implies a doubling of the data storage capacity of the cache. Also, as soon as a cache line has become temporally silent due to a temporally silent store, we broadcast a validate transaction to all other processors in the system.

In Figure 6, we show the reduction in communication data traffic possible with MESTI. We see harmonic mean reductions in the scientific benchmarks of 21% and 15% over the Baseline

	CPU 0		CPU 1	
Time	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**
T0	LD [A]	Read/Miss	LD [A]	Read/Miss
T1	ST [A], 1	Invalidate		
T2	MEMBAR			
T3	ST [A], 0	Validate		
T4	MEMBAR		MEMBAR	
T5			LD [A]	Read/Miss

Table 3: Code Example for MESTI. Coherence transactions for a core with USS store squashing and weak ordering are shown (LD [A] at T0 returns 0). The LD miss at T5 can be eliminated with MESTI.

	CPU 0		CPU 1		CPU 2	
Time	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**
T0	LD [A]	Read/Miss	LD [A]	Read/Miss	LD [A]	Read/Miss
T1			ST [A], 1	Invalidate		
T2	MEMBAR		MEMBAR			
T3	LD [A]	Read/Miss				
T4			ST [A], 0	Invalidate		
T5			MEMBAR		MEMBAR	
T6					LD [A]	Read/Miss
T7			ST [A], 1	Invalidate		
T8	MEMBAR		MEMBAR			
T9	LD [A]	Read/Miss				
T10	ST [A], 0	Invalidate				
T11	MEMBAR				MEMBAR	
T12					LD [A]	Read/Miss

Table 4: Code Example for TSS. Coherence transactions for a core with USS store squashing are shown (LD [A] at T0 returns 0). The LD misses at T6, T9, and T12 can be eliminated with ideal TSS.

and USS cases; and 40% and 32% for the commercial workloads. Note that MESTI is within 4% of the TSS limit in communication misses for all workloads except *raytrace* (6%) Also note that the relative ability of MESTI to approach the TSS limit is slightly larger in the commercial workloads versus the scientific workloads, indicating that available opportunities for exploiting temporal silence are different between workload types. We have already discussed this extensively in Section 4.

To illustrate why MESTI is incapable of exploiting all cases of TSS, we show two sample load/store sequences, one that can be exploited (in Table 3) and one that cannot (in Table 4). In Table 3, the Read/Miss at T5 can be eliminated because the data it requires is validated at T3. However, Table 4 shows a more complicated scenario in which the load misses at T6, T9, and T12 are all TSS (because the values read at those points by respective CPUs do match the value seen previously by that particular CPU), but are not exploitable with MESTI. In short, MESTI is not able to exploit all cases of TSS because it only allows reversion to the immediately previous globally visible version of a cache line. Depending on sharing patterns, multiple versions which differ between CPUs may be required. Augmenting MESTI and investigating other system enhancements to further approach the TSS limit is a subject of continuing work. However, the relatively simple, non-speculative, protocol seems promising.

6.3 Stale Storage Considerations

Stale storage of an entire cache line allows the best possible performance of MESTI. However, this implies a doubling of cache storage. In Figure 7, we show the effect of limiting stale storage to a subset of an entire cache line--either two separate 8 byte blocks (16B case) or one 8 byte block (8B case)--for both MESTI and TSS[1]. We see that for all benchmarks, limiting stale storage to 16B (1/4 of a cache line) achieves nearly the ideal reduction in communication misses versus full cache line stale storage. The only exception is in *tpc-w* for TSS, where the difference is 5%. When stale storage is limited to 8B (1/8 of a cache line), *barnes* and *specweb* also show a non-trivial lost opportunity of 8% and 5%, respectively. However, in all cases, the reduction in sharing misses is substantial even with limited stale storage. Since most of the reduction with TSS is in true sharing misses, this implies that for sharing patterns which exhibit temporal silence, relatively few locations within the cache line are participating in the sharing. This observation may aid us in developing cost-effective methods for exploiting temporal silence in the future.

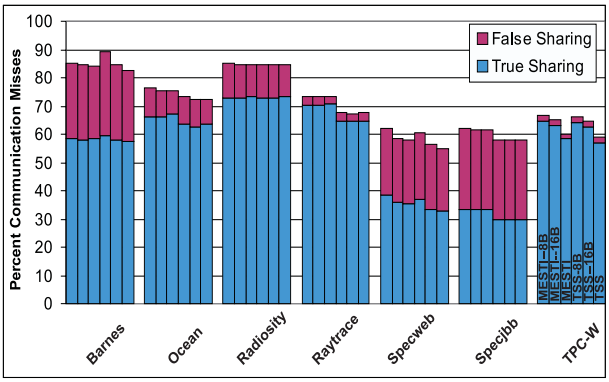

FIGURE 7. Effect of Limiting Stale Storage on Exploiting Temporal Silence. The reduction in communication misses normalized to the Baseline case (see Figure 6) for 64B cache lines and infinite cache lines for MESTI and TSS with limited stale storage (16B/cache line, 8B/cache line) is shown.

We also found that for 16MB caches, using nearly equivalent overall storage (i.e. a 16MB conventional cache vs. a 14MB, 7-way associative cache with ~1.8MB stale storage organized as 8B/cache line), the overall miss rate of the 14MB cache with either MESTI or TSS was better than a conventional 16MB cache with USS store squashing. We leave detailed exploration of efficiently implementing limited stale storage to future work, especially exploiting inclusive cache hierarchies to obtain stale storage at no extra cost.

1. Limiting stale storage for general TSS does not correspond directly to an implementation in some sense (consider the example in Table 4). However, we still present characterizing data for this case for completeness.

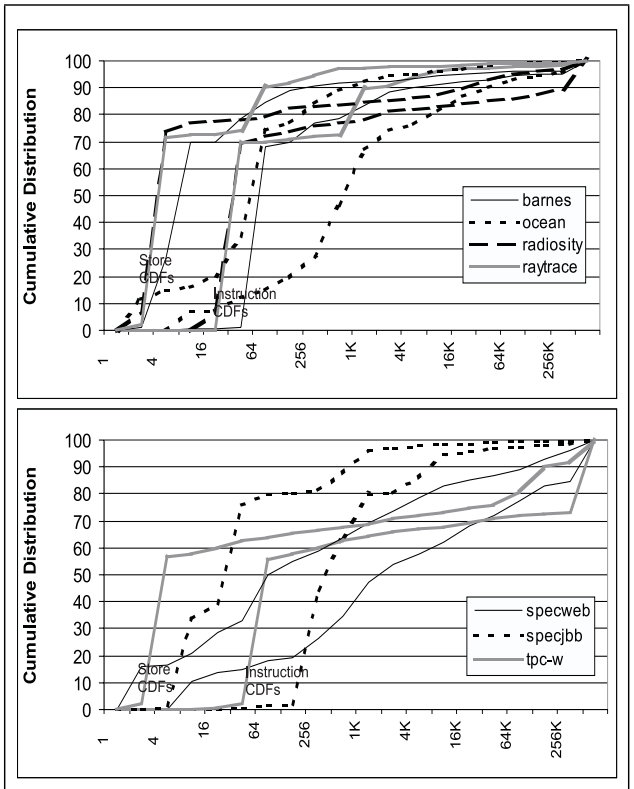

FIGURE 8. Dynamic Instruction/Store Distances Between Useful Temporally Silent Pairs. The figure shows the cumulative distribution of distance (in dynamic instructions and dynamic stores) between the intermediate value store and the temporally silent store in a silent pair for cases in which MESTI prevents a remote miss.

6.4 Temporally Silent Pair Distance

To efficiently exploit temporal silence, we need to understand the distance between the intermediate value store and the temporally silent store which make up a temporally silent pair. To first order, the distance determines the difficulty of exploiting TSS with finite buffering, since stores that are far apart require a deeper memory for tracking the original value.

In Figure 8, we show the cumulative distribution of store pair distance measured in terms of both dynamic non-silent stores and instructions executed for "useful" temporally silent pairs. We call a pair useful if we can exploit it to prevent a communication miss under MESTI. Focusing first on dynamic store distance, in the scientific workloads we see that over 80% of useful silent pairs can be captured within a distance of 64 dynamic stores. However, in the commercial workloads this same distance will only capture 50% of useful silent pairs, with the 80% level not passed until distances of 8K and 64K in *specweb* and *tpc-w*, respectively. Examining dynamic instruction distance, the trends are similar, but we observe greater separation between benchmarks. In the scientific workloads, a short distance of 32 instructions captures almost 70% of opportunity in *radiosity* and *raytrace*, but the same level is not reached for *barnes* and *ocean* until distances of 128 and 2K, respectively. In the commercial workloads, *tpc-w* reaches 55% of opportunity within distance 64, but for *specjbb* and *specweb*, the vast majority of opportunity is not reached until distances of 1K and 16K instructions or more, respectively.

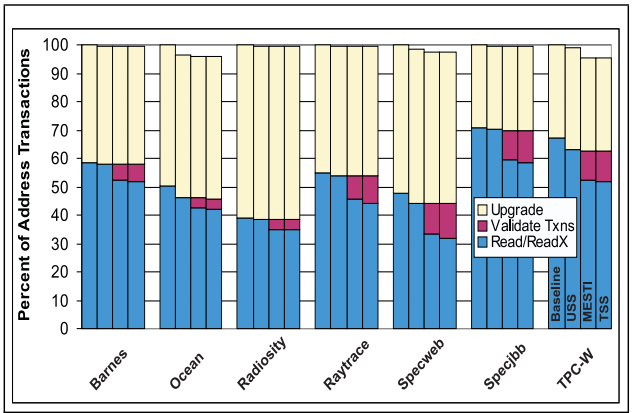

FIGURE 9. Best Case Address Traffic Exploiting Temporal Silence. Address traffic increases for MESTI and TSS are shown for an oracle predictor of useful temporal silence.

In summary, for many cases the temporally silent pair distance can be substantial, especially in commercial workloads. Again, this implies that in-core techniques with limited buffering are unlikely to be effective.

6.5 Address Traffic Effects

We have focused on data traffic exclusively thus far. However, with the addition of coherence states and transactions to exploit temporal silence, the increase in address traffic must be examined. In Figure 9 we present the best possible performance of MESTI and TSS with respect to observed address traffic in the system. The stacked bars indicate the percentage of address traffic in the system, normalized to Baseline, due to data requests (Read/ReadX), additional requests due to temporal silence exploitation (Validate), and finally ownership (Upgrade) requests. In the figure, we assume an oracle predictor for "useful" validates, i.e. a validate is only broadcast if at least one remote processor is able to avoid a data transaction due to it. For all benchmarks, overall address traffic remains essentially constant or decreases slightly. This result is reasonable, as a single validate can avoid multiple demand data transactions.

Examining a more realistic scenario, in Table 5 we show the measured increase in address traffic over Baseline when broadcasting a validate at each temporal silence detection (Naive Validate) for differing cache configurations. In most cases the address traffic increases considerably over the ideal case in Figure 9, motivating attempts at reduction. A simple way to reduce the traffic is to collect snoop responses to ReadX/Upgrade transactions indicating whether or not the cache line was present in a remote cache at the time. If the responses indicate it was not present in any remote cache, we are certain that any validate for this line is useless and we can avoid it. We call this policy *Snoop-Aware Validate*, with its performance indicated in the table[1]. The reduction in address traffic due to this simple optimization is non-trivial in *ocean*, *radiosity*, *raytrace*, and *specjbb* in the case of infinite caches.

In the case of finite caches, we see that the address traffic increases for Naive Validate are significantly less than the infinite case, primarily due to two factors: 1) Absolute increase in baseline address traffic due to added capacity/conflict misses; 2) Fewer Validates because lines which have been replaced from the cache between the intermediate value store and the temporally

1. Figure 5 does not show support for this optimization to reduce clutter.

Benchmark	Naive Validate				Snoop-Aware Validate				Last Write Accuracy
	Inf.	16MB	8MB	4MB	Inf.	16MB	8MB	4MB	
barnes	15.3%	15.3%	15.3%	15.3%	15.3%	12.6%	12.6%	12.6%	5.80%
ocean	45.3%	31.5%	30.0%	17.6%	38.7%	24.0%	22.2%	9.01%	15.4%
radiosity	52.4%	35.1%	32.3%	31.0%	43.8%	27.7%	25.5%	24.3%	14.3%
raytrace	70.2%	32.8%	32.8%	31.5%	57.9%	20.0%	20.0%	19.2%	24.9%
specweb	92.2%	52.7%	47.7%	41.2%	88.8%	45.4%	39.2%	32.0%	24.1%
specjbb	107.7%	53.9%	49.2%	44.9%	97.3%	40.6%	36.5%	32.0%	20.0%
tpc-w	80.6%	55.8%	39.9%	25.6%	77.6%	52.2%	34.7%	17.9%	26.0%

Table 5: Address Traffic Increase by Exploiting Temporal Silence and Last Write Statistics. Results are shown for MESTI with infinite, 16MB, 8MB, and 4MB (8-way associative) caches. Last Write Accuracy is measured for an infinite cache.

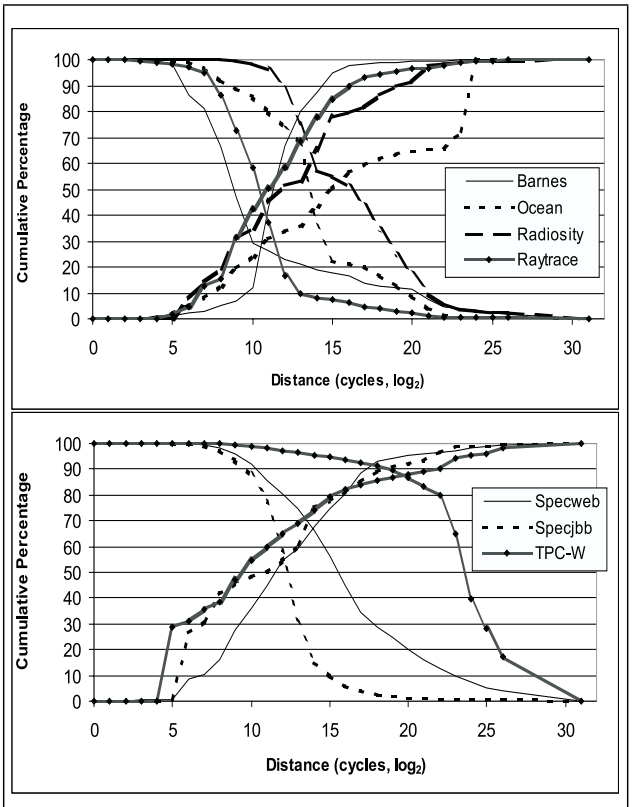

FIGURE 10. Temporally Silent Write to Fetch/Next Write Histograms. The monotonically increasing curves indicate the cumulative distance (in cycles) from a temporally silent write to a subsequent non-silent write for useless cases of temporal silence. The monotonically decreasing curves indicate the cumulative distance (in cycles) from a temporally silent store to a subsequent MESTI avoidable miss**.**

silent store do not lead to a Validate (or prevent a remote miss) because stale storage is not added in memory. More noteworthy is the relative effectiveness of the Snoop-Aware Validate policy in the case of finite caches. All benchmarks show a non-trivial reduction in additional address transactions due to due to Snoop-Aware Validate; in all workloads except *barnes* and *tpc-w* (for the 16MB case) the reduction is at least 7%.

The substantial increase in address traffic can be understood by examining how often a temporally silent write is the last write to the cache line of interest, where the last write is the final write to the cache line before it is requested by another processor [12]. It is clear that if many temporally silent writes are not last writes, many validates and invalidates not present in the baseline case will be broadcast because the current owning processor will write the cache line again before a remote processor requests it. We refer to this as *address thrashing*. Given that the last write accuracy of temporal silence is always less than 26%, address thrashing will occur often if we do not actively try to prevent it.

One possible way of reducing this occurrence is to place any outbound validate into a queue with a fixed delay. The validate remains in the queue until the delay time expires and the validate is broadcast; until a demand transaction occurs for the line and temporal silence is communicated to the requestor; or until a non-silent write occurs to the cache line and the validate is dropped. Delaying validates in this manner filters validates when we detect a non-silent store to the cache line in the queue, but timeliness of the validates will also be affected. Figure 10 characterizes the effectiveness of this delay queue approach. For each benchmark, the monotonically *increasing* curves indicate the distance, in processor cycles, from the final temporally silent store to the cache line to a subsequent (non-silent) write for cases in which the temporally silent write is not the last write. This cumulative distribution indicates how many useless validates we can avoid by delaying them by a fixed number of cycles. The monotonically *decreasing* curves indicate the cumulative distribution of the distance, in processor cycles, from the final temporally silent store to a cache line to a subsequent MESTI avoidable miss to the cache line. This cumulative distribution indicates how long we can wait before propagating a validate and still avoid a communication miss.

Ideally, the overall distribution would be bi-modal, i.e. the "not last write" distance would be short and the "MESTI avoidable miss" distance would be long. In this case, we could simply build a delay queue long enough to capture most of the "not last write" distance, and we would trade relatively few useful validates for this while still allowing useful validates plenty of time to propagate to remote processors. For the scientific workloads, the figure indicates a short queue (2^7 cycles) can eliminate approximately 15% of address thrashing, with approximately 5% of temporal silence opportunity lost in *ocean*, *radiosity*, and *raytrace*. However, the distribution is not sufficiently bimodal for this approach to eliminate the majority of address thrashing without sacrificing most of the opportunity. For the commercial workloads a short queue (2^7 cycles) removes 30%-35% of address thrashing in *specjbb* and *tpc-w* with less than 1% of opportunity lost. The distribution is sufficiently bimodal in *specweb* and *tpc-w* (at 2^{13} cycles) to allow at least 60% of thrashing to be removed with lost opportunities of only 25% and 5%,

respectively. We leave detailed exploration of the delay queue design space to future work.

Finally, we note that the "MESTI avoidable miss" distribution indicates that neglecting address bus contention in these limit study results does not affect timeliness of useful validates significantly. The vast majority of useful validates (if sent immediately) will have plenty of time to reach a remote processor.

7 CONCLUSION

Communication misses are a pressing problem in modern multiprocessor systems. Previous work [16] has shown that silent stores can be exploited to reduce communication misses. This work extends the definition of silent stores to include the set of stores that change the value at a given memory location, but only temporarily, and subsequently return a previous value of interest to the memory location. We examine the occurrence of dynamic stores which contribute to temporal silence in user code, the operating system, and shared libraries, and also provide quantitative characterizing data indicating that temporal silence exists in many forms and arises under many program constructs. These analyses also reveal that studying temporal silence using full-system simulation (including operating system and library code) is worthwhile in scientific workloads and necessary in commercial workloads. We redefine sharing to account for temporal silence and show that up to 45% of communication misses can be eliminated by exploiting this property. We also describe a non-speculative, system-level mechanism which captures up to 42% of temporal silent sharing (TSS) by detecting when temporal silence occurs and communicating it to remote processors with new coherence support. We provide measurement and characterization data concerning temporal silence to guide methods of efficiently exploiting it and include methods and insight into limiting additional address traffic necessary for temporal silence exploitation. This work examines further aspects of store value locality and illuminates how it can be exploited to improve multiprocessor performance, as well as providing fundamental insight into value communication in multiprocessor systems.

In future work, we will continue to enhance methods of detecting and exploiting temporal silence to further approach the TSS limit introduced in this work, and will explore implementation trade-offs necessary for a practical realization. Results presented in this work indicate this is a promising area of future research for eliminating communication misses in multiprocessor systems.

Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670, and CCR-0133437, and generous equipment donations and Fellowship support from IBM and Intel. We would like to thank Jim Goodman and Mark Hill for valuable feedback on both terminology and presentation, as well as Gordie Bell, Ilhyun Kim, and Trey Cain for fruitful discussions, suggestions, and contributions to earlier versions of this work. We also thank the anonymous reviewers for their many helpful comments.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, Dallas, TX, USA, 30 November–2 December 1998. ACM Press.

[2] A. Alameldeen, C. Mauer, M. Xu, P. Harper, M. Martin, D. Sorin, M. Hill, and D. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Proceedings of Computer Architecture Evaluation using Commercial Workloads (CAECW-02)*, February 2002.

[3] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[4] G. B. Bell, K. M. Lepak, and M. H. Lipasti. A characterization of silent stores. In *Proceedings of PACT-2000*, Philadelphia, PA, October 2000.

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

[6] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural characterization of java tpc-w. In *Proc. of HPCA-7*, January 2001.

[7] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA*, 2002.

[8] IBM Corporation. AIX v4.3 online documentation. http://nc-sp.upenn.edu/aix4.3html/, 2002.

[9] D. Culler and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1999.

[10] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *20th Annual International Symposium on Computer Architecture*, May 1993.

[11] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[12] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of HPCA-5*, Orlando, January 1999.

[13] T. Keller, A. M. Maynard, R. Simpson, and P. Bohrer. Simos-ppc full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[14] G. Lauterbach and T. Horel. UltraSPARC-III: designing third generation 64-bit performance. *IEEE Micro*, 19(3):56–66, 1999.

[15] K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50(11), November 2001.

[16] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA-2000*, Vancouver, B.C., Canada, June 2000.

[17] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of MICRO-2000*, Monterrey, CA, November 2000.

[18] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending SMPs. *ACM SIG-PLAN Notices*, 35(11):25–36, November 2000.

[19] C. Moore. POWER4 system microarchitecture. In *Proceedings of the Microprocessor Forum*, October 2000.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, December 2001.

[21] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, 2002.

[22] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.