

Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters

Thomas J. Ashby, Pedro Díaz, and Marcelo Cintra, *Member, IEEE*

Abstract—Implementing shared memory consistency models on top of hardware caches gives rise to the well-known *cache coherence* problem. The standard solution involves implementing coherence protocols in hardware, an approach with some design complexity, hardware costs, and restrictions on interconnect behavior. However, for some memory consistency models, an alternative is to enforce coherence in the software implementation of synchronization primitives, using software controlled invalidations and forced writebacks. This requires minimal hardware support but gives less selective enforcement, which affects performance. This paper proposes a novel hybrid software-hardware coherence mechanism. In this scheme, software is responsible for triggering the coherence actions—self-invalidations and writebacks—at appropriate times while hardware uses Bloom filters to perform more selective self-invalidations. We evaluate the proposed scheme on applications from two different domains: the SPLASH-2 scientific and ALP multimedia benchmarks. Experimental results show that while the software-only coherence scheme shows less performance degradation than expected, it still unacceptably degrades performance for some of the benchmarks. Filtering out unnecessary invalidations improves the worst-case performance by as much as 93 percent, and brings the performance of the hybrid scheme within five percent of full hardware coherence for 10 out of 13 benchmarks, on a 32-core CMP with a shared L2 cache.

Index Terms—Multiprocessors, cache coherence.



1 INTRODUCTION

SMALL- and medium-scale multiprocessors are now commonplace, from single-chip embedded systems, desktops and workstations, to multichip servers and even game consoles. These multiprocessors provide a single address space and are usually programmed using the *shared-memory* programming model. The use of hardware data caches with this programming model leads to the problem of cache coherence.

Traditionally, cache coherence has been enforced in hardware with coherence protocols. Snooping coherence was for sometime a straightforward mechanism to implement on top of shared buses. However, implementing snooping coherence in more recent technology requires much more complex interconnects both in the backplane [10] and on chip [23]. Alternatively, centralized directory coherence [19], [21] requires large centralized structures that are difficult to scale, and distributed directory coherence [25] leads to complex coherence controllers and protocols that are notoriously hard to verify [1], [28]. In the early days of shared-memory multiprocessors, a few attempts were made at enforcing cache coherence completely in software and also with minimal hardware support [11], [12], [13], [14], [16], [31],

[35], [38], [40]. The key idea in such schemes is that software (usually a compiler) identifies both the points at which data cached locally may become stale and the data. Thus, *self-invalidations* are inserted by the software to purge such possibly stale data before possible consumptions, as well as *forced writebacks* to push modified data to memory after possible productions. On the other hand, hardware assistance may be provided to refine the identification of data for self-invalidations. These attempts, however, had significant limitations in that they could only handle very simple data access patterns, and were only successfully applied to regular scientific and engineering applications.

Independent of its implementation, using caches and hardware cache coherence, programming with the shared-memory paradigm requires a clear definition of how memory operations to different memory locations can appear to interleave. For this purpose, memory consistency models are used and several have been defined. Arguably, the most intuitive models for programmers are *sequential consistency* (SC) and *release consistency* (RC). Current standard programming languages and language extensions provide models that are very similar to these (OpenMP, for instance, uses a model very similar to RC).

Since SC requires that all stores become globally visible before any of the following memory operations by the same processor, it greatly benefits from hardware cache coherence. RC, on the other hand, only requires that stores be performed before the completion of a release operation and, then, only with respect to the next processor to acquire the lock. Thus, continuous cache coherence between synchronization points under RC is simply not necessary and may be overkill. In fact, the synchronization points under RC are fully exposed, but such knowledge is largely ignored by current hardware coherence protocol implementations,

- T.J. Ashby is with the Digital Components Division, IMEC, DC/SSET, Kappeldreef 75, 3001 Leuven, Belgium. E-mail: ashby@imec.be.
- P. Díaz and M. Cintra are with the School of Informatics, University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom. E-mail: pedro.diaz@ed.ac.uk, mc@staffmail.ed.ac.uk.

Manuscript received 28 Mar. 2008; revised 12 May 2009; accepted 6 Apr. 2010; published online 18 June 2010.

Recommended for acceptance by K. Ghose.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-03-0133. Digital Object Identifier no. 10.1109/TC.2010.155.

which still assume that communication of data across threads can happen at any memory operation.

The main problem that plagues software-based cache coherence is that, for all but the most regular of applications, it is very difficult for the compiler and/or programmer to safely know what cached lines have to be invalidated at any particular communication point. As a result, most software-based cache coherence schemes either perform full cache invalidations [40] or selective but conservative cache invalidations [35]. Such indiscriminate or conservative selective invalidations may lead to the unnecessary invalidation of some cached lines and subsequent cache misses that degrade performance. The key limitations of software approaches are the limitations of program analysis and unpredictable runtime behavior, including not only variable control flow but also speculative execution and hardware prefetching. Hardware support has been shown to help reduce the amount of unnecessary invalidations [11], [12], [13], [16], [31].

In this paper, we revisit software-based cache coherence with limited hardware support and propose a scheme that attacks these limitations in two ways. First, the scheme performs invalidations only when necessary by leveraging the apriori knowledge of true communication points in the applications under the RC memory model. Previous software-based approaches have also exploited such knowledge, but mostly in the more restricted scope of DOALL and DOACROSS loops (e.g., [11], [12], [13], [16], [31], [40]). Second, the scheme filters out some unnecessary invalidations of cache lines, using a simple hardware mechanism based on Bloom filters that, similar to the Bulk scheme [8], [9], [39], encodes the write sets of processors in a small signature. The contribution of this paper is to propose a software protocol, which is enabled by the use of hardware support, solely for the manipulation of such signatures.

The proposed Bloom-filter-based hybrid software-hardware scheme is evaluated and compared against both a simple software-based scheme with full invalidations and a hardware cache coherence protocol. The study includes benchmarks from two different domains—the SPLASH-2 scientific and ALP multimedia benchmarks—and considers two chip-multiprocessor (CMP) configurations. Experimental results show that while the software-only coherence scheme shows less performance degradation than expected, it still unacceptably degrades performance for some of the benchmarks. Filtering out unnecessary invalidations improves the worst-case performance by as much as 93 percent, and brings the performance of the hybrid scheme within five percent of full hardware coherence for 10 out of 13 benchmarks, on a 32-core CMP with a shared L2 cache.

The rest of this paper is organized as follows: Section 2 discusses cache coherence and memory consistency in more detail; Section 3 presents a baseline software-based cache coherence scheme; Section 4 presents our Bloom-filter-based selective invalidation scheme; Section 5 describes our evaluation methodology; Section 6 presents the experimental results; Section 7 discusses related work; and Section 8 concludes the paper.

2 THE SHARED-MEMORY PROGRAMMING MODEL

2.1 The Cache Coherence Problem

In the shared-memory programming model, all threads share a single virtual address space. This means that when

<pre>int A, B; struct lock flag; A = B = 0; acquire(flag); barrier(); if (pid==0) { A = 1; release(flag); }</pre>	<pre>int A, B; struct lock flag; barrier(); if (pid==1) { acquire(flag); B = A; }</pre>
--	---

(a)
(b)

Fig. 1. Example of synchronization under RC. Variables A, B, and flag are shared.

two or more threads access some program variable, they all access the exact same memory location. When private caches are introduced into the memory hierarchy, multiple copies of the same variable may exist in one or more caches. If some of these copies are modified then threads will observe different values for the same variable. This is known as the cache coherence problem, and making sure that this does not happen is the responsibility of the cache coherence mechanism. However, a coherence mechanism only serves to enforce a consistency model, which may require synchronization to fully specify an ordering on a set of memory accesses by different threads.

2.2 Memory Consistency Models

Memory consistency models specify in what order the memory operations from different threads to different memory locations may appear to the threads. In the most strict model, sequential consistency (SC), the memory operations from different threads must appear to all threads in a single total order, and also the memory operations from a given thread must appear in program order.

Many memory consistency models have been proposed that relax the ordering restrictions of SC. The most relaxed model is release consistency (RC). This model differentiates special synchronization operations—acquire and release—from normal memory operations. Fig. 1 shows a producer-consumer communication pattern under RC. The ordering constraints of RC and its use of explicit synchronization map well to the programmer's expectations when high-level synchronization structures are used.

3 SOFTWARE CACHE COHERENCE

The idea behind the software coherence mechanism we propose for release consistent applications is to allow reads and writes to shared data in private caches and enforce coherence at synchronization points. This is done by invalidating cache lines when the processor reaches an acquire or a barrier (after writing them back, if they are dirty), and by writing back dirty cache lines when the processor reaches a release. The lines to invalidate or write back are determined by a conservative approximation of preceding stores that will communicate data to another processor (writebacks) and future loads that will receive data communicated by other processors (invalidations). The simplest and most conservative approximation is to include all dirty lines in the writeback set and the whole cache in the

invalidation set. Barriers are implemented using locks and result in a combined acquire and release for all processors.

Reads and writes between synchronization points will generate incoherent cache lines. The invalidation of cached data at acquires and barriers guarantees that in the future the thread will obtain, through cache misses, the up-to-date version of data after it is allowed past a barrier or into a critical section. For instance, the invalidation of cached data at the acquire in Fig. 1b will purge the stale value of A and the load after the acquire is guaranteed to be a cache miss. The writeback of cached data at releases and barriers guarantees that the thread will update memory (or some shared lower-level cache) with its latest version of data before it exits a critical section or passes a barrier. For instance, the writeback of cached data at the release in Fig. 1a will send the new value of A to memory before the lock can be released. Finally, the sequential consistency of acquires and releases combined with RC's contract that true sharing can only occur across synchronization points guarantees that no thread will ever consume a stale version of data. Thus, the load of A after the acquire in Fig. 1b is guaranteed to return the new value of A=1.

While the idea described above seems straightforward, there are three difficulties associated with it. First, when using a conservative approximation, there may be performance degradation due to harmful coherence cache invalidations at barriers and acquires. In Section 4, we describe our Bloom-filter-scheme for selective invalidations. A second difficulty associated with the scheme is that, as described, it does not work when multiple threads write to the same cache line between synchronization points (false sharing). This is because there is no way of identifying which words or bytes have been modified by a thread and, so, it is not safe to write back the whole line, as unmodified bytes may overwrite modified bytes previously written back by a different thread. Note that this problem applies to all writebacks, not only to the software managed forced writebacks at synchronization points. A solution to this problem is described below.

A third difficulty relates to synchronization. Most lock algorithms nowadays use spin-waiting on a cacheable variable, relying on the cache coherence mechanism to flag the existence of a modified value. Relying on forced writebacks and self-invalidations to support spin-waiting is possible, but clearly suboptimal as a whole cache invalidation would be performed when clearly only the line containing the lock variable needs to be invalidated. Instead, with a software coherence mechanism, it makes more sense to modify the synchronization algorithms to use noncacheable variables and backoff mechanisms to avoid too frequent retries. A similar problem occurs with barriers, which commonly signal the release via spin-waiting on a cacheable variable and a regular store to this variable. In this case, it also makes sense to replace these regular loads and stores with noncacheable ones.

Addressing the second difficulty described above—that of correctly handling false sharing—completely in software, is difficult. The simplest hardware approach is to use write-through caches, but this comes with an associated performance cost from extra write transactions. An alternative for write-back caches is to add a dirty bit per byte, and only

write back modified bytes, when a line is evicted. This scheme comes with 12.5 percent area overhead for the dirty bits. We consider both approaches in the evaluation.

4 BLOOM-FILTER-BASED SELECTIVE SELF-INVALIDATION

4.1 Overview

Invalidating the whole cache at acquires is clearly too conservative. Ideally, we would like to invalidate in the processor acquiring the lock only the cached lines that are involved in true communication. These correspond to cached lines that have been modified by any processor that has held this lock previously *and* that are subsequently read by the processor acquiring the lock. The second condition requires knowledge of the future, but exploiting it is irrelevant as lines that are not read after the lock acquire may as well be invalidated. Exploiting the first condition, however, may save future unnecessary false coherence misses. Obviously, individually identifying all lines that have been modified by the processors, that have held each lock, is impractical. An intermediate solution is to identify a superset of such lines. This superset should be tight enough to be a good approximation of the ideal set and its encoding should be efficient enough to be manipulated in hardware. It must also be a true superset, meaning that it may include lines that have not been modified, but it must include *all* lines that have been modified.

Bloom filters [7] provide such an efficient superset encoding. Basically, these filters are a space-efficient probabilistic data structure that can be used to test whether an element is a member of a set. False positives are possible, but false negatives are not. They are space-efficient because the resulting signature is much smaller than a complete and exact representation of the contents of the set.

What we propose then is to use Bloom filters to perform selective invalidations to assist a software cache coherence mechanism. The basic idea of the scheme is as follows: A Bloom-filter-based write-set signature is maintained in hardware in each processor and is updated at each store to the cache (or cache miss) using a portion of the line address. All signatures are empty at the beginning of the program. We call these signatures the *processor-side signatures*. At a lock release the processor's signature is written to a memory location associated with the particular lock. This is the *lock-side signature*. At a lock acquire the lock-side signature of the lock being acquired is read from memory and is used by the hardware to selectively invalidate the local cache. This signature is then merged (union) with the processor's current write-set signature in order to provide the transitive communication defined by locks.

Barriers are implemented in software using locks and are mostly handled by the mechanism just described for locks. In fact, implementations of barriers use a lock that must be acquired by each processor to enter. Hence, after all processors have entered, that lock carries the signature of the complete global write-set for the program so far. This signature can then be used to selectively invalidate all caches just before resuming work past the barrier. At this point, the barrier can be released.

The size of a write-set may grow very large over the course of execution of the program. Hence, signatures may

become saturated (i.e., all bits are set and, thus, membership tests always return true). In theory, any write that has been seen by all processors can be removed from all write-sets as it no longer has to be protected. This would be difficult to do, since Bloom signatures do not support removal of individual elements. However, all signatures (processor-side and lock-side) can be reset to empty any time when all writes so far are visible to all processors. This condition naturally occurs at barriers. Thus, we clear all signatures just before a barrier is released.

4.2 Implementation Details

4.2.1 Hardware Write-Set Signature Registers

Maintaining and manipulating the write-set signatures completely in software would incur too high an overhead as each application store would have to be augmented with possibly multiple loads (depending on the size of the signature) to obtain the current signature from memory, some computation to encode the new signature, and multiple stores to save the new signature back to memory. Thus, we propose to use hardware assistance for this operation.

In our proposal, the write-set signature for each processor is stored in an architecturally visible register, similar to what is done in [39]. This register can be located in the processor core itself or in the L1 cache controller, depending mostly on the choice of address bits used to generate the signature. Being architecturally visible, this register can be addressed by software, and its contents can be loaded from and stored to memory as necessary. Naturally, such loads and stores may involve several words as the signature is likely to be in the order of a few hundred bytes in order to achieve reasonable selectivity. We name these assembly load and store instructions *LDWSIG* and *STWSIG*, respectively, and the processor's write-set signature register *RSIG1*. In addition to this register, our scheme requires a second signature register where the signature from other processors can be stored temporarily. This register is also architecturally visible, and we name it *RSIG2*.

Updating the write-set signature involves simple binary logic and shift operations, but on a possibly wide datapath. We propose to implement this operation in hardware, using dedicated logic. This logic is placed next to the *RSIG1* and *RSIG2* registers so that it can operate on them. Updates occur on every application store or only on every cache write miss, again depending on the choice of address bits used to generate the signature. As we chose to ignore address bits in the byte-offset portion of the address, our scheme only updates *RSIG1* on cache write misses. Also, for the hash functions that we study in this paper, the encoding operation amounts to a traditional N to 2^N DECODER, which, in current technology, will only take a small number of a processor cycles. Besides being inexpensive, this operation can be performed in parallel with the memory lookup. Thus, we do not expect this operation to affect either the hit or miss latencies, for both loads and stores.

To perform the selective invalidations of cache lines, we rely on dedicated hardware, which is described in Section 4.2.2. These operations, however, are invoked by the software through a new assembly instruction, *SIGINVAL*. This instruction takes no argument and always uses the

value in *RSIG2* as the mask for the invalidations. We also propose to use dedicated hardware under software control to perform the union of signatures, which is required after the cache invalidation upon a lock acquire, as described in Section 4.2.3. This union operation of Bloom-filter-signatures is a straightforward bitwise, OR operation, and will only take a fraction of a processor cycle. The instruction to invoke this operation is named *MRGSIG* and takes no argument, always assuming that the value in *RSIG2* is to be merged with that in *RSIG1* and stored in *RSIG1*.

A final architectural extension, required to support the scheme, is an additional assembly instruction used by the software routines to reset the two signature registers. We name this instruction *RSETSIG*.

In summary, transferring write-set signatures between processor/cache and memory is done in hardware under software control, as is using signatures to perform the selective invalidations and resetting signatures. Updating the local processor-side write-set signature, on the other hand, is transparently handled by the hardware.

4.2.2 Hardware-Selective Cache Invalidations

Fig. 2 depicts the hardware required to perform all the relevant signature operations in our scheme. Fig. 2a shows the high-level organization of the hardware for performing the selective invalidation at lock acquires and barriers (Sections 4.2.3 and 4.2.4), the union operation at lock acquires (Section 4.2.3), and the signature update at each cache write miss (Section 4.2.1). As we can see from this figure, the major hardware cost comes from the hardware for performing the selective per-line invalidations. Each circuit, for selective invalidations, amounts to a DECODER and a number of 2-input AND gates, as can be seen in Fig. 2b. Each DECODER is similar to those found in memory address decoding circuits and many optimization techniques from that domain can be applied, such as predecoded pairs of bits and dynamic logic. In the worst case, each DECODER can be implemented as shown in Fig. 2c, using large fan-in AND gates or, alternatively, a tree of low fan-in AND gates. In either case, the cost of the hardware for selective invalidations is proportional to the number of such circuits that are used in parallel. The diagram in Fig. 2a shows an aggressive implementation with one circuit per cache tag.

One alternative to reduce this hardware cost is to group cache lines and to time-share a single selective invalidation hardware per group and, thus, sequentially process each line in a group. This optimization trades off hardware costs for a larger performance cost for the selective invalidations. We estimate that the time for the circuit in Fig. 2b to process one cache tag is small enough that it is reasonable to assume a single invalidation logic per cache set in a two or four-way set-associative cache. It is worth noting that while the hardware described above may not be much cheaper in terms of silicon cost as that of a coherence controller, the benefits we aim for are in terms of design and verification complexity, not silicon area. The logic described above is simple and local to each processor and does not involve a complex protocol.

As the hit latency of upper level caches is commonly on the processor's critical path, it is important that the hardware additions proposed do not increase this latency.

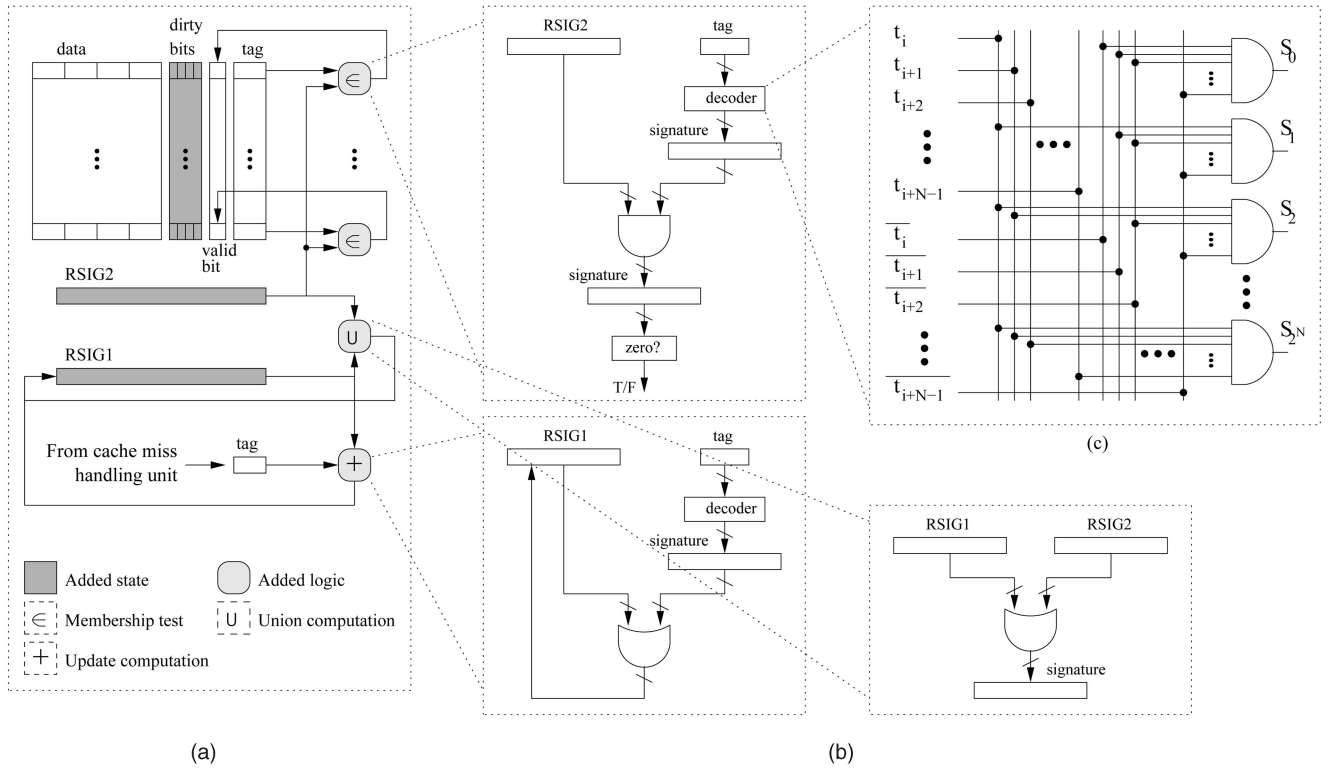


Fig. 2. Proposed architectural extensions: (a) overview of the architecture for the selective invalidation (used at lock acquires and barriers), union operation (used at lock acquires), and signature updates (used at cache write misses); (b) mechanisms for performing the signature union, membership test, and signature update; and (c) simple circuit for decoding N bits from the tag into a 2^N bit signature.

The potential problem is that the circuit used in Fig. 2b presents an extra load on the cache tags. However, we note that the output of this circuit is not needed for the cache accesses. Thus, in the case that this extra load may impact the tag checking latency, one can simply place this circuit in series with the normal tag checking logic, and out of the critical cache hit path. Also, the selective invalidation circuit shown in Fig. 2a is only activated at synchronization points and, thus, also has no impact on normal cache operation.

4.2.3 Lock Acquire and Release

One of the key ideas in our proposal is to rely on the software to decide when communication has to occur and what the set of processors that are involved is. As explained in Section 3, this is achieved by leveraging the high-level information provided by the RC memory consistency model. In order to potentially reduce the impact of cache flushes, what we propose is to augment the lock and barrier software data structures with additional write-set signature information, and their software routines with code to direct the hardware to manipulate and use these signatures.

Each lock is potentially associated with a communication chain, and the data modified under its protection can be associated with a write-set signature. We extend the lock data structure with a field where its associated write-set signature can be stored. This field consists of multiple words, and hardware support is required to load and store it into hardware registers (Section 4.2.1). The lock acquire routine is extended to instruct the hardware to load this write-set signature, to use it to perform the selective cache invalidations, to merge it with the local processor's write-set signature, and to store it back to memory. Fig. 3a shows (in

a mix of C and assembly pseudo-code) this extended acquire routine for the simple algorithm based on spinning with exponential backoff [30]. Similarly, the lock release routine is extended to instruct the hardware to load the lock's write-set signature, to merge it with the local processor's write-set signature, and to store it back to memory. This is shown in Fig. 3b. Note that reloading the lock's write-set signature on a release is necessary in order to support nested locks.

Other, more complex, lock algorithms can also be used with the proposed hybrid software/hardware cache coherence mechanism. The key to porting such algorithms to work with the proposed mechanism is to make all the data structures used to communicate state in the algorithm (e.g., the array of slots and the "next slot" index for the array-based queue lock algorithm [30]) noncacheable. Another important consideration is that, as already mentioned, all spin-waiting should be done with some backoff mechanism to avoid too much traffic.

We envision that these changes only need to be applied to the synchronization library, and no change is required to the application. Finally, note that writebacks are still performed, as described previously, for the software coherence mechanism in Section 3.

4.2.4 Barriers

Barriers are usually implemented in software, using a small number of locks and counters, and many different algorithms have been proposed (e.g., [30]). Basically, all algorithms involve acquiring a "check-in" lock, updating some shared state, releasing the lock(s), and then waiting on

<pre> struct lock flag; void acquire(lock thisflag) { /* original lock acquire code */ /* thisflag is non-cacheable delay is priv. */ while (test_and_set(thisflag) == LOCKED) { pause(delay); delay = delay*2; } /* added code for select. inv. */ __asm__ ("LDWSIG RSIG2, thisflag.wsig" "SIGINVAL" "MRGSIG" "STWSIG thisflag.wsig, RSIG1"); } </pre>	<pre> struct lock flag; void release(lock thisflag) { /* added code for writeback */ __asm__ ("FLUSHD"); /* added code for updating sig. */ __asm__ ("LDWSIG RSIG2, thisflag.wsig" "MRGSIG" "STWSIG thisflag.wsig, RSIG1"); /* original lock release code */ /* thisflag is non-cacheable */ thisflag = UNLOCKED; } </pre>
(a)	(b)

Fig. 3. Pseudocode for lock acquire (a) and release (b) in the proposed hybrid software/hardware cache coherence mechanism. The base algorithm is the simple spin with exponential backoff [30]. The `FLUSHD` instruction is equivalent to the Sparc `FLUSH` instruction but operates on the data cache. The example assumes a `test_and_set` primitive, but other hardware primitives could be used instead.

some global release variable. These algorithms work correctly in any system that correctly implements locks and cache coherence, including the software scheme with full cache invalidations described in Section 3. Using the signature-based selective invalidation, however, requires a small change to the barrier routine.

The problem is that as processors arrive at the barrier and obtain the check-in lock, they observe a signature that encodes the cumulative write-sets of all processors that have previously arrived at the barrier. Thus, only the last processor to arrive observes the complete signature. To correct this problem, we modify the barrier routine in two ways. First, we add a new *barrier-side signature*, stored alongside the barrier data structure, that is produced by the last arriving processor using the lock-side signature of the check-in lock merged with its own processor-side signature. As with the lock-side signatures, this barrier-side signature is computed by hardware logic but under software control, which requires augmenting the barrier code executed by the last arriving processor. Second, we augment the barrier code so that a processor's first action, once the barrier is released, is to load the (now updated) barrier-side signature and to use it to invalidate its own cache. Again, this is done with hardware logic but under software control. Fig. 4 shows (in a mix of C and assembly pseudo-code) the extended routine for a simple sense-reversing centralized barrier [30]. Again, more complex barrier algorithms can be extended to handle the signatures and the key consideration for the extension is to make the data structures used by the algorithm noncacheable.

One important property of barriers is that they represent application points where all threads must observe exactly the same state of the shared memory. Our scheme guarantees that this is the case by ensuring that all shared data are not cached anywhere. As no shared data are cached at this point, it is safe to reset all the write-set signatures, both the processor- and lock-side ones. Resetting all the processor-side write-set signatures is triggered by the software barrier code by issuing a `RSETSIG` instruction as the first operation, once a processor, resumes execution after the barrier (Fig. 4).

Resetting the lock-side write-set signatures of all the locks is somewhat more involved. First, the synchronization library code for barriers must be able to identify all the active lock variables in the application. To support this, we

augment the lock declaration or initialization code with extra instructions to add the lock to a dynamic linked list of locks. For instance, pthreads require that all locks be initialized (with `pthread_mutex_init()`). We, then, augment the barrier routine to traverse this linked list and reset all the lock-side signatures, by storing a null value in them (with a `RSETSIG` and a series of `STWSIG`). This

```

struct barrier_t barrier;

barrier.count = P;
barrier.sense = TRUE;
private int local_count;
private int local_sense = TRUE;

void central_barrier() {
    /* toggle own sense */
    local_sense = !local_sense;

    /* check in to barrier */
    acquire(barrier.check_in);
    local_count = barrier.count--;
    release(barrier.check_in);

    /* check if last processor */
    if (local_count == 1) {
        /* added code for lock-side sigs. */
        reset_locksidesigs();

        /* reset barrier for next instance */
        barrier.count = P;
        barrier.sense = local_sense;

        /* added code for barrier-side sig. */
        __asm__ ("MRGSIG"
                "STWSIG barrier.wsig, RSIG1");
    }
    else {
        /* wait for barrier release */
        /* sense is non-cacheable delay is priv. */
        while (sense != local_sense) {
            pause(delay);
            delay = delay*2;
        }

        /* added code for sel. inv. */
        __asm__ ("FLUSHD"
                "LDWSIG RSIG2, barrier.wsig"
                "SIGINVAL"
                "RSETSIG");
    }
}

```

Fig. 4. Pseudocode for sense-reversing centralized barrier [30] with write-set signatures. The procedure `reset_locksidesigs()` resets all the lock side signatures and is omitted in the interest of space.

operation only needs to be performed once, so we make it the responsibility of the last arriving processor, and it is done after producing the barrier-side signature and just before releasing the other processors (Fig. 4). Note in Fig. 4 that it is not necessary to reset the barrier-side signature at any point, as it is overwritten at the barrier's next instance. This operation may appear expensive, but in practice most applications only use a relatively small number of (static) locks. Moreover, the frequency of global barriers is low enough and their base cost already high enough that the performance impact of resetting the lock-side write-set signatures is likely to be small in practice.

4.2.5 Hash Function

Bloom filters use a number of hash functions to map set elements to positions in a bitmap. The number and behavior of these functions depend on the intended use of the Bloom filter. An important design constraint in our case is that because these functions must be implemented in hardware, they must be relatively simple and fast. We have found that using just one function that extracts certain bit slices from the memory address, performs well enough on our test workloads. For a write-set signature of 2^K bits ($K = 11$ in our experiments), we extract a slice of K contiguous bits from the write address. Our experiments showed that different slices can lead to significantly different selectivity rates. In particular, we noted that bits in the middle of the address give far better results. These middle bits of the address contain the page number the address belongs to. Using these bits provides good selectivity between private and shared memory pages in threads.

We also tried signatures that encode the exact set of cache set indices as suggested in [8], but found that they provide no selectivity at all in our case, as every set in the caches is touched by a store operation to a line that maps to it.

4.3 Discussion

4.3.1 Hardware Costs

Table 1 summarizes the hardware extensions required by the whole scheme.

In summary, the hardware extensions required by the whole scheme we propose amounts to:

1. Five new instructions;
2. 8 bytes per cache line for dirty bits;
3. 512 bytes for the two signature registers;
4. 2,048 2-input OR gates for the signature union logic;
5. 11 NOT gates, 1,100 2-input AND gates, and 2,048 2-input OR gates for the signature update; and
6. 11 NOT gates, 1,100 2-input AND gates, and 2,048 2-input OR gates for the membership test per group of lines (Section 4.2.2).

These numbers assume a 2 Kbit signature.

4.3.2 Compiler and OS Support

Doing away with hardware cache coherence has obvious implications for legacy shared-memory parallel programs. However, properly synchronized programs, especially those that use calls to a synchronization library should be easily ported to the hybrid software-hardware cache coherence scheme proposed here. One potentially difficult

TABLE 1
Summary of Hardware Extensions Required by the Proposed Hybrid Software/Hardware Cache Coherence Scheme

Extension	Cost
<i>Instruction Set</i>	
5 new instructions	Negligible
<i>SRAM</i>	
Dirty bits per byte (64byte lines)	8 bytes per cache line
2 signature registers (2Kbit signatures)	512 bytes total
<i>Logic</i>	
Signature union (2Kbit signatures)	2048 2-input OR gates total
Signature update (2Kbit signatures)	11 NOT gates + 1100 2-input AND gates + 2048 2-input OR gates total
Membership test (2Kbit signatures)	11 NOT gates + 1100 2-input AND gates + 2048 2-input AND gates

The costs for signature update and membership test assume a simple DECODER logic and could be optimized. The costs for membership test are for each individual circuit (Section 4.2.2).

case is that of multithreaded OS's, which make intensive use of synchronization to access their key data structures in a parallel environment. While porting them to the proposed scheme is likely not a trivial endeavor, it is facilitated by the fact that they are usually maintained by a relatively small number of programmers who have a deep understanding of the code and the required synchronization points.

Also, the proposed scheme does not require any special compiler or OS support, but it can potentially benefit from some assistance. The OS could help, for instance, by placing the private data of different threads in pages that will not alias under the hash function used by the hardware. The compiler could help, for instance, when applications do not use barriers often, which can lead to signature saturation. In this case, the compiler can find appropriate points in the program to insert safe barriers. We leave a study of OS and compiler support for the proposed scheme as the future work.

4.3.3 Potential Difficulty: Adoption of Relaxed Models

For our scheme to be efficient, communication must be infrequent and must be clearly marked. Applications designed for RC and some other relaxed models meet these requirements, but applications designed for other memory consistency models may not. Thus, the scheme depends heavily on the adoption of relaxed memory consistency models.

Relaxed memory consistency models were originally introduced mainly because of better performance with respect to SC, and because they are arguably easier to implement in systems with complex memory hierarchies [3]. Later, it was noted that aggressive dynamic instruction scheduling in processors could close a large fraction of this performance gap while still providing a tight memory model to the programmer, which led to the suggestion that this should be the case [17]. This approach, however, has only been implemented in one processor, the MIPS R10K. The suggestion was based on the assumption that SC is a better memory consistency model from a software engineering perspective. However, many believe that most

programmers have neither the skills nor the tools to safely use handcrafted communication and synchronization that relies on the strict ordering of memory operations imposed by SC, thus negating its main purported benefit.

While the debate is still open on what memory consistency models the architecture should provide and what models programmers should see, in the end what matters is the models used in de facto standards for parallel programming environments. For scientific HPC the leader is OpenMP, and for embedded, desktop and server applications the dominant models are Java, and POSIX threads. The consistency model of OpenMP is only slightly less relaxed than RC in that it does not allow loads to cross below a release or stores to cross below an acquire. As such, our scheme can be easily extended to work with it by the addition of memory fences to acquires and releases. The memory model for POSIX thread libraries is deliberately informal, but essentially conforms to RC in that all communication must be protected by calls to synchronization routines. Thus, our scheme should work “out of the box” with POSIX threads. The Java memory model is much more involved than those for C, but, in essence, it also requires that programs use explicit synchronization. Finally, a trend in recent programming languages is to use “atomic” blocks (e.g., [4], [15]), which also fit straightforwardly with our proposed scheme.

5 EVALUATION METHODOLOGY

5.1 Applications

For our quantitative performance analysis, we use all nine SPLASH-2 application benchmarks [41] and four ALPBench benchmarks [27]. The SPLASH-2 benchmarks are representative of scientific and engineering workloads while the ALPBench benchmarks are representative of multimedia workloads. We built both benchmark suites using POSIX threads, with explicit locks and barriers implemented, using custom assembly code (including all the instructions necessary to manipulate the lock- and barrier-side signatures). The ALP benchmarks were optimized so that they created one set of threads during the initialization phase rather than repeatedly setting up and tearing down threads during the course of the program.

We used the reference inputs for the SPLASH-2 benchmarks. Only 20 frames from the test file are used for MPGDEC/ENC to reduce simulation time. Because the input sets for the ALPBench benchmarks were not intended to be used with more than 16 processors, we do not simulate larger systems for these benchmarks. For all benchmarks, we only consider the execution time of the parallel region; i.e., after initialization and before cleanup and printing of results. Table 2 lists the benchmarks we used, their inputs, the number of dynamic barrier occurrences, the number of dynamic lock acquire occurrences, and the average number of cycles between two successive synchronization points.

5.2 Simulation Environment

We built our simulators on top of the SPARC target simulator of Virtutech’s Simics [29]. The baseline Simics simulator models an in-order single-issue processor using the UltraSPARC III+ instruction set. This simulator does not model the microarchitecture in detail, but we extended the

TABLE 2
(a) Applications Used and (b) Parameters of the Memory Subsystem Model

Benchmark	Dyn. Barriers	Dyn. Acq.	Cycles/Sync.
water-nsq	20	1102	330K
water-spt	20	19	8.1M
ocean-np	870	207	423K
ocean-cp	900	207	507K
raytrace	1	3,037	1.6M
radiosity	2	322,779	1.9K
barnes	21	2,403	363K
volrend	20	3,660	241K
fmm	33	6,549	736K
face_rec	3	0	2.8M
MPEG_dec	40	0	15M
MPEG_enc	40	0	27M
sphinx3	4,593	54,151	115K

(a)

Parameter	Value
No. of cores	1 to 32
Core freq.	2GHz
Issue width	1, in-order
L1,L2 size (bytes)	64 KB, 2MB
L1,L2 associativity	4-way, 4-way
L1,L2 line repl. policy	LRU, LRU
L1,L2 line size (bytes)	32 byte, 64 byte
L1,L2 latency (cycles)	3, 15
Memory latency (cycles)	200

(b)

The Total Number of Dynamic Lock Acquires Is for 16 Processors and Does not Include Retries.

simulator with a detailed model of the cache and memory subsystem. While not modeling the microarchitecture in detail does introduce some errors, we believe that differences due to, for instance, pipelining, operand bypass, and branch prediction, can be considered as secondary effects when compared to the timing impact of the memory accesses.

5.3 Designs Evaluated

In our evaluation, we concentrate on CMP systems with 32 processors. Consistent with the current trend toward CMP’s with many, but simple, cores, we assume cores with a relatively low clock rate of 2 GHz and with single issue in-order pipelines. We perform experiments with systems from two different design domains: a future embedded CMP with private write-back L1 data caches and no globally shared L2 (such as Tilera’s Tile64 [5] and Intel’s SCC [18]), and a future desktop/workstation CMP with private write-through L1 data caches and a shared write-back L2 cache. The reason for including both is to assess the impact of the cost of reloading self-invalidated data, which is an order of magnitude different between an on-chip shared cache and main memory. Arbitrators for memory transactions are implemented using separate request and response transactions. The on-chip interconnect from L1 to L2 cache is 32 byte wide and runs at the same frequency as the processors. The off-chip memory bus is one quad-word wide (16 bytes) and also runs at the same frequency. Contention is modeled in detail on all interconnects, properly accounting for the overheads of the forced write-backs. We used exponential backoff for the lock acquire

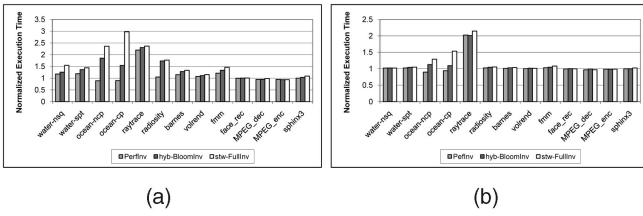


Fig. 5. Execution time for 32/16 processors normalized to hardware coherence: (a) no L2 configuration, (b) configuration with shared L2.

spin-wait. Table 2b shows the processor and system parameters used in the simulations.

For comparison, we model CMP systems with full hardware coherence using an MSI snooping protocol (*hdw-MSI*), with a baseline software coherence scheme with full self-invalidations (*stw-FullInv*), our proposed hybrid software-hardware scheme with selective self-invalidations (*hyb-BloomInv*), and an ideal software-based scheme with perfect selective self-invalidations at lock acquires and barriers (*PerfInv*).

With *hyb-BloomInv*, we model in detail the latency to load and write the signatures from memory, as these operations were added to the assembly code of locks and barriers. As for the hardware operations shown in Fig. 2, we assume that all operations take one cycle for simplicity (as discussed in Section 4.2.1 only the encoding operation would take longer, but it can still be easily overlapped with the memory operation for the update signature operation and its impact in the, already expensive, synchronization operations is expected to be minimal). We assume one invalidation logic per cache set, and invalidations proceed serially with all overhead properly accounted for.

6 EXPERIMENTAL RESULTS

6.1 Overall Performance

Fig. 5 shows, for all applications, the performance of *stw-FullInv*, *hyb-BloomInv*, and *PerfInv*, for 32 (SPLASH-2) and 16 (ALPBench) processor systems, relative to that of *hdw-MSI*. As can be seen from this figure, the simple *stw-FullInv* scheme performs surprisingly well for many applications, specially for configurations with a shared L2 cache. However, despite its overall reasonable performance, *stw-FullInv* performs unacceptably poorly for some applications: up to 298 percent performance degradation for systems without L2 and up to 215 percent performance degradation for systems with shared L2. Our proposed Bloom filter based hybrid scheme (*hyb-BloomInv*) is able to improve performance over *stw-FullInv* by as much as 93 percent, and 15 percent on average, for systems without L2 and by as much as 40 percent, and

five percent on average, for systems with shared L2, significantly closing the gap. The performance gap between *hyb-BloomInv* and *hdw-MSI* for systems without L2 is between -6 percent (i.e., speedup) (*MPEG_enc*) and 130 percent (*raytrace*), and 36 percent on average. The gap is between -1 percent (i.e., speedup) (*MPEG_enc*) and 100 percent (*raytrace*), and 11 percent on average, for systems with shared L2. More importantly, the gap is less than five percent for 10 out of 13 applications, for systems with shared L2. Moreover, *hyb-BloomInv* performs very close to the ideal *PerfInv*, with a performance gap between 0 percent (*MPEG_enc*) and 107 percent (*ocean-ncp*), and 20 percent on average, for systems without L2, and between 0 percent and 25 percent, for systems with shared L2.

To assess the relative impact of full and selective self-invalidations for different system sizes, we vary the number of processors in the system from 2 to 32/16. Fig. 6 shows the performance results, again normalized to that of *hdw-MSI*. As can be seen from this figure, this relative performance is mostly independent of the number of processors, at least up to 32/16 processors.

6.2 Invalidation Behavior

The success of the proposed scheme depends mainly on two factors: the fraction of the cache that is invalidated unnecessarily at each synchronization point and the frequency of synchronization points. Fig. 7 shows, for all applications, the fraction of the caches that is invalidated for signatures using different ranges of bits ($b[X-Y]$), and for the ideal case (*ideal*). The numbers are for a single execution interval between two barriers. We noticed that for all applications, the invalidation behavior is very similar across all such intervals. As can be seen from this figure, the best signature across all applications is $b[24-14]$, which simply decodes 11 bits from the physical address starting from bit 24 to bit 14, inclusive. Further, to assess the impact of signature size, we tried signatures with 1,024 and 4,096 bits, encoding bits $b[23-14]$ and $b[25-14]$, respectively. The results showed that $b[25-14]$ did not perform significantly better than the baseline $b[24-14]$ and $b[23-14]$ performed slightly worse. Overall, considering that $b[25-14]$ signature takes twice as long to be read and written to memory, we believe that the baseline $b[24-14]$ offers a good design point.

To put this invalidation fraction within a perspective of execution progress, Fig. 8 shows, for a representative selection of applications (due to space limitations—*fmm*, *face_rec*, *MPEG_dec*, and *MPEG_enc* behave similarly to *water-spt*; *water-nsq*, *raytrace*, *volrend*, and *sphinx3* behave similarly to *barnes*; and *ocean-ncp* and *radiosity* behave similarly to *ocean-cp*), the fraction of the caches that is invalidated for signature $b[24-14]$ (*hyb-BloomInv*) and for the

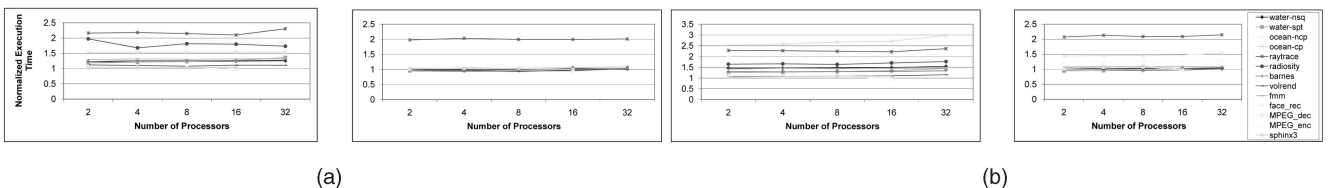


Fig. 6. Execution time for varying number of processors normalized to hardware coherence: (a) (leftmost two charts) Bloom-filter-based selective invalidations without L2 (left) and with shared L2 (right), (b) (rightmost two charts) full invalidations without L2 (left) and with shared L2 (right).

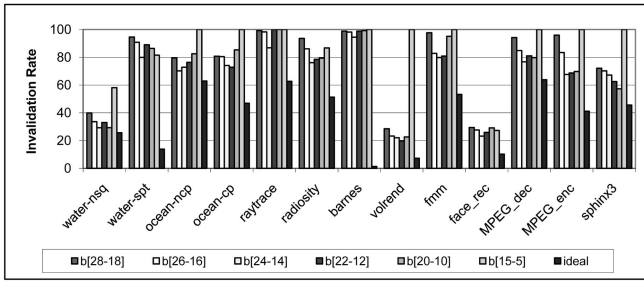


Fig. 7. Invalidation rates for different signatures.

ideal case. Unlike Fig. 7, the numbers in this figure are for all synchronization events (lock acquires and barriers) for a representative execution window of 50M cycles (again, we did not observe much variability across different windows for any given benchmark). Looking at this figure at the relative performances in Fig. 5, and the rightmost column in Table 2, we can see three types of behavior. In the first type of behavior—*water-spt*—synchronization is fairly infrequent and *hyb-BloomInv* achieves performance very close to the ideal invalidation coherence and somewhat close to the hardware coherence, regardless of whether it tracks the ideal invalidation rates closely or not. In the second type of behavior—*barnes*—synchronization is frequent, but *hyb-BloomInv* tracks the ideal invalidation rate closely. In this case, *hyb-BloomInv* tends to still perform very close to the ideal invalidation coherence and somewhat close to hardware coherence (the exception is *raytrace*, which has a very high synchronization frequency and all, but the hardware coherence scheme performs poorly). Finally, in the third type of behavior—*ocean-cp*—synchronization is also frequent and *hyb-BloomInv* does not track the ideal invalidation rate closely. In this case, *hyb-BloomInv* does not perform as closely to the ideal invalidation coherence or hardware coherence, but it still leads to significant performance gains over *stw-FullInv*. Comparing results with the first and third behaviors, we can see that the frequency of synchronization events has a significant impact in the performance of *hyb-BloomInv*, and comparing results with the second and third behaviors, we can see the performance impact of how well *hyb-BloomInv* tracks the ideal invalidation rates.

We did experiment with signatures that encode the exact set of cache set indices as suggested in [8]. We found, however, that this provided no selectivity at all, as every set in the caches is touched by a store operation to a line that maps to it. The difference between our negative results with this technique and the positive results in [8] is that the

threads in our work are much larger than the transactions and speculative threads in that work.

7 RELATED WORK

7.1 Software and Hybrid Cache Coherence

Software-directed cache coherence on shared-memory multiprocessors has been previously proposed as an alternative to hardware-based cache coherence [11], [14], [35], [37], [38], [40]. Like the scheme we propose, such schemes were also based on self-invalidations and forced writebacks of the private caches at synchronization points (some required a write-through cache instead of forced writebacks). As in our case, they relied on explicitly marked synchronization points (often simply the boundaries of parallel loops). Additionally, to filter out unnecessary invalidations, such schemes relied on the programmer [35], [37] or the compiler [11], [12], [14], [31], [38], [40] to identify what data are involved in the communication and, thus, must be written back and invalidated. Such schemes, however, were only applicable to very regular array and loop-based codes, where static analysis can identify a tight superset of the communicated data or when the programmer can clearly specify critical sections and their data.

Some previous works, similarly to ours, have proposed additional hardware support to assist with the actual invalidations. Closest to our work, both [12] and [31] rely on generational behavior of shared data along synchronization points. They use per-word version/timestamp tags updated by software to identify cache lines that do not have to be reloaded after a synchronization point. The earlier work in [11] used a single bit to differentiate data versions between two generations, and it was later extended in [13], [16]. However, unlike our work, those require complex program analysis to statically manage the update of the version tags. As a result, they were only successful in programs with simple control flow (often DOALL and DOACROSS loops) and regular array accesses. Also, the tag overhead in the caches is much higher than the overhead of our signatures.

7.2 Hardware and Software Self-Invalidation

Self-invalidation of locally cached data has been proposed as a mechanism to reduce the amount of coherence transactions. Both hardware [24], [26] and compiler-directed [33] schemes have been proposed. Those schemes differ from ours in that they are non-exact and they still rely on hardware coherence mechanism to ensure correctness.

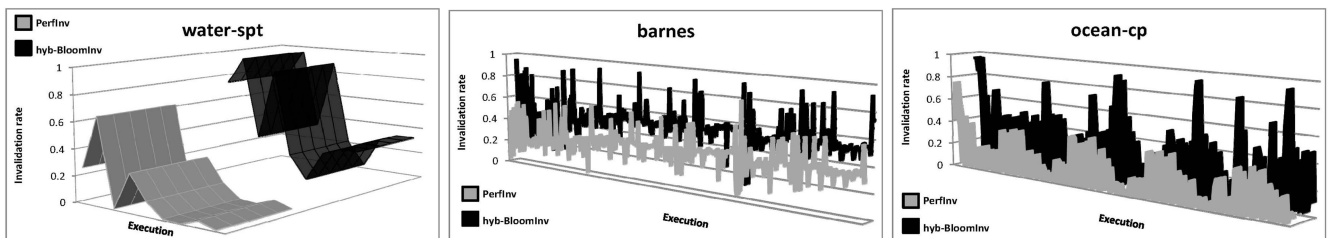


Fig. 8. Invalidation rates per synchronization event for Bloom-filter-based selective invalidation and perfect invalidation. Results are for processor 0 in a 32/16 processor configuration.

7.3 Bloom-Filter-Based Bulk Invalidation

The idea of using Bloom-filter-based signatures to encode a thread's read- and write-sets was first proposed in [8]. That work used the signatures in the context of Thread-Level Speculation (TLS) and Transactional Memory (TM) for detecting conflicts across speculative memory accesses and to selectively invalidate incorrect speculative data from the caches in the case of roll-backs. Recently, other hardware TM systems have been augmented with this signature approach [32], [42]. Both TLS and TM are novel programming models that differ significantly from the familiar explicitly parallel programming model that we tackle in our work. Closer to our work, [9] used Bloom filter read and write-set signatures in the context of explicitly parallel shared-memory applications for achieving the behavior of the SC memory model while allowing hardware to reorder memory operations. Again, the signatures are used for detecting conflicts across speculative memory accesses and to selectively invalidate incorrect speculative data from the caches in the case of roll-backs. The key difference between such previous work and ours is that in those the protocol is almost entirely implemented in hardware, while our approach moves most of the required functionality to software. Another important difference is that all such previous works are based on protocols with speculative execution, which require even further hardware support for roll-back, while our approach is not speculative.

Concurrently with our work, the work in [39] has also advanced the idea of exposing signatures to the software while supporting the costly signature operations in hardware. Our work differs from that in that our goal is to provide a software signature-based mechanism that specifically aims at supporting a software cache coherence approach. The goal of that work is to provide a very flexible software signature mechanism, so that it is overkill for addressing the specific problem that we address. Also, in our work, we demonstrate in detail how to modify the software in order to use such a software signature mechanism to support software cache coherence.

7.4 Software-Distributed Shared Memory

There has been much work on software and hybrid distributed shared memory systems (e.g., [6], [20], [22], [36]). One common difference between all of those schemes and ours is that those targeted clusters of workstations and not only dealt with cache coherence but also provided the shared memory image. In our case, we assume multiprocessors with shared memory and only consider the problem of providing cache coherence in the software layer. Most of those schemes also differ from ours in the mechanisms employed to enforce coherence: we rely on hardware mechanisms to identify the data that must be invalidated and/or written back and to handle false sharing, while those rely on the virtual memory system and a page level diff approach.

8 CONCLUSIONS

In this paper, we revisit the idea of software-based cache coherence with explicit writebacks and self-invalidations. Our study of software cache coherence for relaxed consistency models has shown that, for a large number of

benchmarks from our test suite, even the most conservative approach produces surprisingly little performance impact compared to a hardware coherence scheme. Certain benchmarks do suffer from a large negative performance impact due to the full cache invalidations. We then proposed and evaluated a novel hardware-assisted scheme that uses Bloom-filter-based signatures to encode the threads' write-sets efficiently and to more selectively invalidate the caches. The proposed scheme requires only minor changes to the IS and the cache controller. Experimental results show that the scheme improves the worst-case performance and brings the performance of the hybrid scheme very close to full hardware coherence for the vast majority of the applications studied. This suggests that the proposed scheme may be a viable alternative to full-blown hardware cache coherence.

ACKNOWLEDGMENTS

The authors thank Per Stenström and his group for their many invaluable suggestions during the visit of the lead author, Dr. Ashby, to Chalmers University of Technology, supported by the HiPEAC network. Sincere thanks are also due to Aris Efthymiou and the anonymous referees for their suggestions. This work was supported in part by EPSRC under grant GR/S79572/01 and by the EC under grants IP 27648 (FP6) and HiPEAC IST-004408. This research work was performed mostly while author Dr. Ashby was with the University of Edinburgh.

REFERENCES

- [1] D. Abts, S. Scott, and D.J. Lilja, "So Many States, so Little Time: Verifying Memory Coherence in the Cray X1," *Proc. 17th Int'l Symp. Parallel and Distributed Processing*, Apr. 2003.
- [2] S.V. Adve, V.S. Adve, M.D. Hill, and M.K. Vernon, "Comparison of Hardware and Software Cache Coherence Schemes," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 298-308, May 1991.
- [3] S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, no. 12, pp. 66-76, Dec. 1996.
- [4] E. Allen, D. Chase, V. Luchango, J.-W. Maessen, S. Ryu, G.L. Steele, Jr., and S. Tobin-Hochstadt, "The Fortress Language Specification. Version 0.618," <http://research.sun.com/projects/plrg/fortress0618.pdf>, 2010.
- [5] S. Bell et al., "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," *Proc. Int'l. Conf. Solid-State Circuits (ISSCC '08)*, pp. 88-89, Feb. 2008.
- [6] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Mar. 1990.
- [7] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, pp. 422-426, July 1970.
- [8] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," *Proc. Int'l. Symp. Computer Architecture*, pp. 227-238, June 2006.
- [9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulk Enforcement of Sequential Consistency," *Proc. Int'l. Symp. Computer Architecture*, pp. 278-289, June 2007.
- [10] A. Charlesworth, "The Sun Fireplane Interconnect," *IEEE Micro*, vol. 22, no. 1, pp. 36-45, Jan./Feb. 2002.
- [11] H. Cheong and A.V. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. Int'l. Symp. Computer Architecture*, pp. 299-307, June 1988.
- [12] H. Cheong and A. Veidenbaum, "A Version Control Approach to Cache Coherence," *Proc. Int'l. Conf. Supercomputing*, pp. 322-330, June 1989.

- [13] H. Cheong, "Life Span Strategy—A Compiler-Based Approach to Cache Coherence," *Proc. Int'l. Conf. Supercomputing*, pp. 139-148, June 1992.
- [14] R. Cytron, S. Karlovsky, and K.P. McAuliffe, "Automatic Management of Programmable Caches," *Proc. Int'l. Conf. Parallel Processing*, pp. 229-238, Aug. 1988.
- [15] Cray Inc. "Chapel Language Specification 0.750.," <http://chapel.cs.washington.edu/spec-0.750.pdf>, 2010.
- [16] E. Darnell and K. Kennedy, "Cache Coherence Using Local Knowledge," *Proc. Int'l. Conf. Supercomputing*, pp. 720-729, June 1993.
- [17] M.D. Hill, "Multiprocessors Should Support Simple Memory Consistency Models," *Computer*, vol. 31, no. 8, pp. 28-34, Aug. 1998.
- [18] J. Howard et al., "A 48-Core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS," *Proc. Int'l. Solid-State Circuits Conf.*, Feb. 2010.
- [19] R. Kalla, B. Sinharoy, and J.M. Tandler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro*, vol. 24, no. 2, pp. 40-47, Mar./Apr. 2004.
- [20] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. Int'l. Symp. Computer Architecture*, pp. 13-21, May 1992.
- [21] P. Kongetira, K. Angaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21-29, Mar./Apr. 2005.
- [22] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr, S. Dwarkadas, and M.L. Scott, "VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks," *Proc. Int'l. Symp. Computer Architecture*, pp. 157-169, June 1997.
- [23] R. Kumar, V. Zyuba, and D.M. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling," *Proc. Int'l. Symp. Computer Architecture*, pp. 408-419, June 2005.
- [24] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," *Proc. Int'l. Symp. Computer Architecture*, pp. 139-148, June 2000.
- [25] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. Int'l. Symp. Computer Architecture*, pp. 241-251, June 1997.
- [26] A.R. Lebeck and D.A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," *Proc. Int'l. Symp. Computer Architecture*, pp. 48-59, June 1995.
- [27] M.-L. Li, R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench Benchmark Suite for Complex Multimedia Applications," *Proc. Int'l. Symp. Workload Characterization*, pp. 34-45, Oct. 2005.
- [28] D. Lie, A. Chou, D. Engler, and D.L. Dill, "A Simple Method for Extracting Models from Protocol Code," *Proc. Int'l. Symp. Computer Architecture*, pp. 192-203, June 2001.
- [29] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, Feb. 2002.
- [30] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization of Shared-Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 9, no. 1, pp. 21-65, Feb. 1991.
- [31] S.L. Min and J.-L. Baer, "A Timestamp-Based Cache Coherence Scheme," *Proc. Int'l. Conf. Parallel Processing*, pp. 23-32, Aug. 1989.
- [32] C.C. Minh, M. Trautmann, J.-W. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," *Proc. Int'l. Symp. Computer Architecture*, pp. 69-80, June 2007.
- [33] M.F.P. O'Boyle, R.W. Ford, and E.A. Stöhr, "Towards General and Exact Distributed Invalidation," *J. Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1123-1137, Nov. 2003.
- [34] S. Owicki and A. Agarwal, "Evaluating the Performance of Software Cache Coherence," *Proc. Int'l. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 230-242, Apr. 1989.
- [35] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.S. Melton, V.A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. Int'l. Conf. Parallel Processing*, pp. 764-771, Aug. 1985.
- [36] S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proc. Int'l. Symp. Computer Architecture*, pp. 325-336, June 1994.
- [37] H. Sandhu, B. Gamsa, and S. Zhou, "The Shared Regions Approach to Software Cache Coherence on Multiprocessors," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 229-238, May 1993.
- [38] A.J. Smith, "CPU Cache Consistency with Software Support and Using One Time Identifiers," Technical Report CSD-86-290, Apr. 1986.
- [39] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization," *Proc. Int'l. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 145-156, Mar. 2008.
- [40] A. Veidenbaum, "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. Int'l. Conf. Parallel Processing*, pp. 1029-1036, Aug. 1986.
- [41] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Int'l. Symp. Computer Architecture*, pp. 24-36, June 1995.
- [42] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," *Proc. Int'l. Symp. High-Performance Computer Architecture*, pp. 261-272, Feb. 2007.



Thomas J. Ashby received the PhD degree in 2005 from the University of Edinburgh for work on using advanced languages and compiler optimizations for scientific computing. He is a senior research engineer in the Digital Components Division of IMEC, the Belgian International Microelectronics Research Center. He has worked on design automation for embedded systems and image processing for hyperspectral applications.



Pedro Díaz received the MS degree in computer engineering from the Universidad Politécnica de Madrid in 2005. He is currently working toward the PhD degree in computer architecture at the University of Edinburgh. His doctoral research involves the study of timeliness of prefetching algorithms for uniprocessor and multiprocessor systems.



Marcelo Cintra received the BS and MS degrees from the University of Sao Paulo in 1992 and 1996, respectively, and the PhD degree from the University of Illinois at Urbana-Champaign in 2001. After completing the PhD degree, he joined the faculty of the University of Edinburgh, where he is currently an associate professor. His research interests include parallel architectures, optimizing compilers, and parallel programming. He has published extensively in these areas. He is a member of the ACM, the IEEE, and the IEEE Computer Society. More information about his current research activities can be found at <http://www.homepages.inf.ed.ac.uk/mc>.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.