To appear in Proceedings of HPCA-3 (February, 1997)

An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors

Hazim Abdel-Shafi[†], Jonathan Hall[‡], Sarita V. Adve[†], Vikram S. Adve[♭]

[†]Electrical and Computer Engineering / [♭]Computer Science
Rice University
Houston, Texas 77005

[‡]Intel Corporation
2111 NE 25th Ave, MS JF1-19
Hillsboro, Oregon 97124

Abstract

Prefetching is a widely used consumer-initiated mechanism to hide communication latency in shared-memory multiprocessors. However, prefetching is inapplicable or insufficient for some communication patterns such as irregular communication, pipelined loops, and synchronization. For these cases, a combination of two fine-grain, producer-initiated primitives (referred to as remote-writes) is better able to reduce the latency of communication. This paper demonstrates experimentally that remote writes provide significant performance benefits in cache-coherent shared-memory multiprocessors with and without prefetching. Further, the combination of remote writes and prefetching is able to eliminate most of the memory system overhead in the applications, except misses due to cache conflicts.

1 Introduction

In traditional cache-coherent, shared-memory multiprocessors, interprocessor data movement primarily occurs in response to a read operation by the consumer of the data value, and is usually at the granularity of a cache line. In such systems, software prefetching has been shown to be highly effective in hiding memory latency. Furthermore, software prefetching is effective in uniprocessors as well as multiprocessors, and is fairly straightforward to implement. Therefore, software prefetching is becoming widely available in commercial systems.

For many programs and sharing patterns (e.g., producer-consumer patterns), producer-initiated data transfers are a natural style of communication. When applicable, producer-initiated communication can be highly efficient because it moves data close to the consumer as soon as the data becomes available, minimizing the latency at the read.

Producer-initiated communication may be either coarse-grain or fine-grain. Coarse-grain producer-initiated (or bulk transfer) primitives are primarily useful for regular, coarse-grain data sharing patterns. In such cases, however, software prefetching is also highly effective and bulk transfer primitives have been shown to provide little additional performance benefit over prefetching for scientific codes [25].

In contrast, fine-grain producer-initiated primitives appear to be useful in certain cases where prefetching is inapplicable or insufficient. In particular, prefetching may not be effective for data references where either (1) the value to be read is not produced sufficiently early (before the read), or (2) the location to be accessed is not known sufficiently early. The first case can occur with synchronization variables, or with tightly synchronized pipelined computations that can arise from loop nests with recurrences. The second case occurs with dynamic sharing patterns. In both cases, a producer-initiated data transfer can be used to bring the data closer to the next reader, thereby reducing the latency seen by the read or prefetch.

In the first case, if the writer can compute which processor will read the value next, it can "send" the value to that processor, depositing it in the future reader's cache. Such an operation is well-suited to a regular producer-consumer style of data sharing, or to certain synchronization algorithms based on predictable communication patterns. Three primitives have been proposed to support such an operation: the forwarding write [16, 17], the DASH deliver [11], and the PSET_WRITE [19].

*This work is supported in part by the National Science Foundation under Grant No. CCR-9410457 and CCR-9502500, the Texas Advanced Technology Program under Grant No. 003604016, and the Defense Advanced Research Projects Agency under Contract No. DABT63-92-C-0038.
Jonathan Hall performed this work while at Rice University.
The authors can be reached at {shafi|sarita|adve}@rice.edu and jchall@ichips.intel.com

In the second case described above, the writer cannot usually predict who will be the next reader. Here, a "write-through" operation could transfer the data to memory so that the latency of a subsequent prefetch or read is reduced. Such an operation is well suited to migratory sharing where data or synchronization variables are repeatedly read and modified by different processors in unknown order. Primitives like Check-In (in the CICO model) and others [6, 23] have been proposed to support such an operation.

While the above two types of producer-initiated primitives have been previously proposed and studied, few studies have evaluated the additional benefits of these primitives when prefetching is already available. The few such studies [16, 24] have used a single producer-initiated primitive (data forwarding), and have been unable to combine the benefits of forwarding and prefetching, for reasons discussed below.

The goal of this paper is to explore whether fine-grain producer-initiated communication primitives can provide complementary benefits to software prefetching in a shared-memory multiprocessor.

To balance performance benefits and ease of use, we choose fine-grain producer-initiated primitives that are software-controlled and cache-coherent. A software-controlled primitive enables precise control over the communication in a program (essentially like a fine-grain message), which can be exploited by application and library developers and parallelizing compilers to achieve high performance. A primitive that preserves cache coherence can be used simply as a performance hint to the system, without changing the semantics of the program.

Corresponding to the two cases identified above where producer-initiated communication appears to complement prefetching, we consider a combination of two software-controlled fine-grain primitives. The first is a slightly simplified version of the forwarding write primitive, which we call `WriteSend`. This primitive is a write that directly updates the cache of a specified processor, and is used when the next consumer of the data is known. The second primitive is called `WriteThrough`. This primitive is a write which updates memory, and is used when the next consumer is not known. Together, we refer to these two primitives as *remote writes*.

We evaluate the performance of remote writes, with and without prefetching, on two synchronization kernels and five applications. In the synchronization kernels, the use of `WriteSend` showed gains of 42% to 60% over the base system; prefetching is not applicable to these kernels. In the applications, the addition of remote writes improved performance by 10%-48% relative to the base system, and by 3%-28% relative to the base system with prefetching. In all applications, the combination of remote writes and prefetching performed better than either of them alone. Perhaps most important, the combination of remote writes and prefetching eliminated most of the memory system overheads in the base system, except cache conflicts in the direct mapped L1 cache.

The largest overall gains relative to prefetching came with the use of `WriteThrough` in applications with irregular sharing patterns, where the `WriteThrough` significantly reduced the average latency that had to be hidden by prefetches, and also greatly reduced false sharing in one case. The overall impact of `WriteSend` over and above prefetching on the applications in our study was small, because the specific patterns it benefited most did not prove to be dominant in these applications. Nevertheless, `WriteSend` was effective in reducing most of the latency it targeted; these latencies are difficult to address with prefetching.

There are three principal reasons why the results of our study differ from those of previous studies that evaluated data forwarding in combination with prefetching [16, 24]: (i) the use of a combination of `WriteSend` and `WriteThrough` primitives; (ii) more realistic architectural assumptions compared to some previous studies, and (iii) a broader application domain, including applications with pipelined loops and irregular sharing patterns, and important synchronization kernels. These reasons are discussed in more detail in Section 5.

In the following section, we define the remote write primitives, and discuss how they can be implemented in a cache-coherent, shared memory multiprocessor. Section 3 describes our experimental methodology, including how remote writes and prefetching were used in the applications we study. Section 4 presents results of our experiments. Section 5 discusses related work. Section 6 summarizes our main conclusions.

2 Remote Writes: Definition and Implementation

2.1 Definition and Programming Model

Section 1 described two situations in which producer-initiated communication could help reduce communication latency in a hardware cache-coherent shared-memory system. The two situations are typical of regular producer-consumer sharing patterns and migratory sharing patterns respectively. Based on those observations, we define two types of remote write in-

structions, for a hardware cache-coherent system. For each, we propose an option to leave the writer's copy of the line (if any) in invalid state. It is important to note that the semantics of a remote write are the same as an ordinary write; a remote write additionally only provides a performance hint. This must be ensured in part by preserving cache-coherence during these operations.

Assume that `Write <addr> <value>` is an ordinary write operation of `<value>` into location `<addr>`. The remote write instructions are:

- `WriteSend <addr> <value> <proc>`
  `WriteSendInv <addr> <value> <proc>`

  Perform an ordinary `Write <addr> <value>`. In addition, deposit the new value of `<addr>` in the cache of processor `<proc>`. If the writer has a copy of the accessed line in its cache, then `WriteSendInv` invalidates that copy, while `WriteSend` does not.

- `WriteThrough <addr> <value>`
  `WriteThroughInv <addr> <value>`

  Same as `WriteSend` and `WriteSendInv` respectively, except that the new value of `<addr>` is updated in main memory, and there is no destination processor.

The above definitions only assume some form of hardware cache-coherence. They can be applied to invalidate or update protocols, and to directory-based or COMA protocols, although certain aspects of the definitions may be inapplicable. For example, on hierarchical COMA systems where a home location does not exist, `WriteThrough` would default to a local write operation. Furthermore, remote writes can be beneficial under either a relaxed memory consistency model or sequential consistency.

Since `WriteSend` specifies only one destination processor, in cases where multiple consumers exist, separate `WriteSend`s to different copies of the data at the different consumers would be needed. Previously proposed primitives similar to `WriteSend` use a bit vector to specify multiple destination processors [11, 16, 19]. We chose a single destination primarily because multiple sends from a single instruction are more difficult to implement within a coherence protocol, and a bit vector is not scalable. All our applications, except for a single broadcast operation in one application, needed only one destination processor.

A remote write can be specified in the source code by bracketing an assignment with annotations. It can be specified at the object code level (without modifying the instruction set) by using available store opcode bits such as ASI bits in the SPARC, or unused high-order address bits, or by preceding the remote write with a store to a special memory address. The destination processor for `WriteSend` can be specified by writing to a special-purpose off-chip register.

2.2 Implementation of Remote Writes

We describe an implementation of remote writes on a base hardware cache-coherent, shared-memory multiprocessor with a directory, invalidation, and ownership-based coherence protocol, and release consistency. We assume each node in the system has a write-through no-write-allocate L1 cache, a write buffer, and a write-back L2 cache.

The two remote write primitives can be implemented using a common set of mechanisms in the underlying hardware. Perhaps most importantly, all but one of the basic mechanisms (described next) would already be present in a typical implementation of the underlying cache-coherence protocol.

Processor-Cache Sub-system. The one additional mechanism we use to improve the performance of remote writes is a coalescing (or merging) line write buffer [3, 4], located between the L1 and L2 caches. Such a buffer can merge multiple remote writes to different words in the same cache line, making them appear as a single remote write message to the rest of the memory system.[1] Dirty bits per word in the buffer indicate which words in the line have been modified.

Remote writes are placed in the coalescing buffer once they are issued from the regular write buffer. They are issued from the coalescing buffer to the L2 cache either on a replacement (using LRU), or when the buffer is flushed at a release. Since a release must wait for acknowledgements from outstanding normal and remote writes, the release could experience the full delay of the buffered remote writes. To reduce this cost, we also flush the coalescing buffer on acquires.

The producer's L2 cache is responsible for forwarding the remote write data to the directory. If the L2 cache has ownership of the line, it merges the new data into the line and forwards it; otherwise, it forwards the modified words with the dirty bits from the coalescing buffer. Additionally, if the producer's L2 cache has a copy of the accessed line, a `WriteThroughInv` or `WriteSendInv` invalidates the copy, while a `WriteThrough` or `WriteSend` updates it

[1] We compared performance with and without the coalescing buffer and found that the buffer does reduce overall network traffic, sometimes by a large amount, and improves execution time for some of our applications [1].

and sets it to shared state.

With `WriteSend` and `WriteSendInv`, we chose to move the cache line into the destination's L2 cache, rather than the L1 cache, to minimize the possibility of cache conflicts. This is important because a `WriteSend` may put the data in the consumer's cache well before it is accessed.

Coherence Protocol. Both `WriteThrough` and `WriteSend` do not require ownership of the cache line. They are handled with simple extensions to the base coherence protocol. To avoid introducing a new class of races in the protocol, we restrict the implementation to (i) serialize all writes to a single cache line, including remote writes, at the directory, just as in the base protocol, and (ii) merge modified words into the cache line at the producer or at the directory, for both types of remote writes. Specifically, remote writes result in the following operations at the directory.

If a `WriteSend` finds the line at the memory in shared state, the new data is simply merged with the line in memory. Invalidations are issued to all processors holding copies of the line except the writer and the destination processor. In parallel, the merged line is sent to the destination processor in shared state. If the `WriteSend` finds the line in exclusive (or dirty) state the directory first issues an invalidation (and write-back) message to the processor with the dirty line.[2] Once the dirty line is received at the directory, its state is set to shared, modified words are merged with the line, and the line is sent to the destination processor in shared state.

The operations for a `WriteThrough` are similar to those for a `WriteSend`, except that actions pertaining to the destination processor are not relevant.

Finally, since `WriteSend` fills arrive unsolicited at the destination processor's cache and can generate a new message (a write-back), they raise the possibility of deadlock. In our base system, write-backs occur on a cache fill and requests do not reserve any buffer space for write-backs. Deadlock is avoided by sending the write-back on the reply network which is guaranteed to be deadlock free. Thus, the deadlock avoidance mechanisms of our base system prevent deadlocks due to `WriteSend`s as well [1].

3 Experimental Methodology

We evaluate the performance impact of remote writes by comparing four systems: a base cache-coherent shared-memory system, and the base system extended with remote writes, software prefetching, and both prefetching and remote writes. We refer to these systems as Base, RW, PF, and PF+RW.

[2] A race occurs if the directory's write-back message reaches the processor after the processor has itself issued a write-back for the line. However, an analogous race needs to be handled even for the base protocol when a new write request is forwarded to a processor with a dirty line.

Parameter	Value
Processor and cache parameters	
Processor speed	300 MHz
Cache line size	64 bytes
Write buffer	8 words
Coalescing remote write buffer	4 cache lines
L1 cache (size varies by application)	Direct mapped
L2 cache (size varies by application)	4-way associative
Network parameters	
Topology	2D-mesh
Network speed	150 MHz
Flit size	4 bytes
Maximum packet size (data+header)	64 + 16 bytes
Flit delay per network hop	1 network cycle
Memory System Latencies (in CPU cycles)	
L1 cache access	1 cycle
L2 cache access	8 cycles
Directory and memory access	18 cycles
Memory transfer bandwidth	8 bytes/cycle
Sample Contentionless Miss Latencies (in CPU cycles)	
Read miss (local clean)	55 cycles
Read miss (dirty 2-hop)	140 cycles
Read miss (dirty 3-hop)	175 cycles

Figure 1: Simulation parameters.

For our evaluation, we use an execution-driven simulator based on direct execution, derived from the Rice Parallel Processing Testbed (RPPT) [2, 18].

3.1 Architectures Modeled

The Base system is similar to the one described in Section 2.2 (but without support for remote writes). For the RW system, we augment Base with support for remote writes as described in Section 2.2. For PF, we augment Base with both shared and exclusive prefetching. We prefetch into the L1 cache because data is usually prefetched just before it is accessed, and is therefore less likely to replace a useful value or be replaced itself. This gives prefetching an advantage over `WriteSend`. PF+RW combines the additional support of both PF and RW.

Figure 1 specifies key system parameters. To illustrate their impact, Figure 1 also gives sample latencies for various shared-memory read operations, assuming no contention. Cache sizes and number of processors vary by application as discussed in Section 3.2. We assume a single-issue, in-order processor model with blocking reads. The functional unit cycle counts are similar to the UltraSparc processor. We assume that accesses to instructions and private data hit in the L1 cache. We simulate contention in the network and memory system.

Because of the combination of a simple processor model and an aggressive network and memory system

Application	#Procs	Data Set	L1	L2
MP3D	16	50,000 particles	16KB	128KB
Water	32	512 molecules	8KB	64KB
Radix	32	1M keys, r=1024	32KB	256KB
Erlebacher	32	$64 \times 64 \times 64$	4KB	32KB
MICCG3D	32	$64 \times 64 \times 64$	4KB	32KB

Figure 2: Application parameters

(relative to recently announced microprocessor-based systems), our results are likely to be conservative for the improvements achievable using prefetching and remote writes in future systems.

3.2 Kernels and Applications Used

We use two synchronization kernels and five applications for our experiments. The two kernels are MCS locks [13] and tree barriers. We chose these kernels because they are of fundamental importance to shared-memory applications, and are well-suited to producer-initiated communication. The five applications are Radix from the SPLASH-2 suite [26], Water and MP3D (without locking for cells) from the SPLASH suite [22], MICCG3D from the MIT Alewife group, and Erlebacher from the Rice Parallel Compiler group. The last two applications have static access patterns, and contain tightly synchronized pipelined loop nests. MP3D and Water have dynamic access patterns, and Radix exhibits both types of patterns.

Input sizes, cache sizes, and the number of processors used for each application are listed in Figure 2. The input and cache sizes are based on the methodology described by Woo et al. [26]. We use 32 processors for all applications except MP3D, for which we used 16 processors because of its poor scalability. We evaluate our synchronization kernels on systems with 1 to 32 processors, using caches that fit their working sets.

3.3 Methodology for Inserting Prefetches and Remote Writes

We use simple heuristics based on application behavior to insert prefetches and remote writes by hand. Therefore, for our architectural parameters, our results provide an upper bound on the performance benefits from prefetching and remote writes. It is important to first determine this bound before attempting compiler support for prefetching and remote writes. Nevertheless, when developing our heuristics for remote writes, we took care to use information that would be possible to infer in an advanced compiler. Our heuristics essentially depend on two items of information: (1) whether a write to a shared location is likely to be followed by a read from a different processor, and (2) whether a cache line is likely to be replaced in the reader's L2 cache in the interval between a `WriteSend` that pushes the line into the cache and a subsequent read operation on that line. Although this information is more than is required for prefetching, advanced parallelizing compilers should be able to determine this information.

We generally followed Mowry's algorithm [14] to insert prefetches in the applications, except that in some cases, we used extensive experimentation to schedule prefetches so as to maximize performance. We do not prefetch synchronization variables, although there may be benefit in prefetching low-contention locks. We insert prefetches only for cache lines that are expected to miss in the L2 cache. The overhead of prefetching L2 cache hits usually outweighs the savings in latency.

For the RW and PF+RW versions, we insert `WriteSend` on a write to a shared variable if the writer can compute the identity of another processor that will access the data next (i.e., a regular sharing pattern), and if the sharing is fine-grain (e.g., in pipelined loops and synchronization). `WriteSend` can also be used for (regular) coarse-grain sharing if the consumer's working set between the write and the subsequent read is smaller than the cache size, so that the remote-written data would remain in the cache until it is read. In such a case prefetching will usually also be effective but with higher network traffic and instruction overhead. We, therefore, opt for `WriteSend` in this case, and then do not use prefetching in PF+RW.

We insert `WriteThrough` on a write to a shared variable if the identity of the next reader cannot be computed (as in an irregular sharing pattern), but where the next reader is likely to be a different processor. In PF+RW, we also insert a prefetch before a read that is part of such an irregular sharing pattern, to bring the data from the home node into the L1 cache. With irregular sharing, it can be difficult to issue prefetches sufficiently early, and the preceding `WriteThrough` can reduce the latency seen by the prefetch.

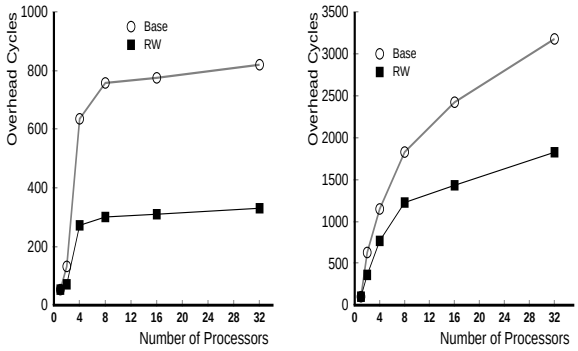
Test&Test&Set lock variables also exhibit irregular sharing. For the unlock of such variables, we use `WriteThroughInv` instead of `WriteThrough` since the next reader will obtain ownership to get the lock.

Finally, when data is written soon after it is read (e.g., migratory data), PF uses exclusive prefetches. In PF+RW, some of these may be replaced by shared prefetches if the corresponding write is a remote write, since remote writes do not need to obtain ownership.

4 Results

4.1 Overview of the Results

We first provide an overview of all of our results, and then discuss the results for the individual kernels



(a) MCS Locks (b) Tree Barriers

Figure 3: Overhead for synchronization kernels

and applications.

Figure 3 compares the average overhead of the synchronization kernels on Base and RW; prefetching is not applicable to these kernels. Using remote writes reduce overhead by 60% and 42% for MCS locks and tree barriers respectively on a system with 32 processors.

Figure 4 summarizes our results for the five applications and for one key phase of Erlebacher which we study in isolation. For each application, the four graphs (from left to right) show the execution time normalized to the base system, the number of L1 and L2 misses per processor, and the total volume of network traffic.[3] The execution time for each system shows the components due to data write stall time, data read stall time, stall times due to different synchronization operations (flags, locks, and barriers), and busy time. The bars for L2 read misses also show the component of misses due to late prefetches (i.e., L2 misses that occur while there is an outstanding prefetch for the requested line). We did not collect statistics on late prefetches at the L1 cache. Finally, the bars for network traffic show the components of traffic on the request and reply networks.

Our results show the following overall trends: (1) PF+RW always performs best, achieving improvements in execution time between 10% and 48% relative to Base, and between 3% and 28% relative to PF for our applications. (2) PF+RW consistently has the lowest L2 cache read misses, eliminating 70–91% of these misses compared to Base and 10–70% compared to PF. (3) PF+RW always shows lower network traffic than Base or PF (but the traffic on RW is sometimes even lower). (4) Compared to Base, PF+RW eliminates most of the memory system overheads in our applications. The remaining memory system overhead is primarily due to cache conflicts. There is also remaining overhead due to load imbalance, but such overhead is usually not directly targeted by memory system optimizations like remote writes or prefetches.

[3]The graphs for the L1 and L2 misses for MICCG3D show the misses scaled down by a factor of 1000.

4.2 Synchronization Kernels

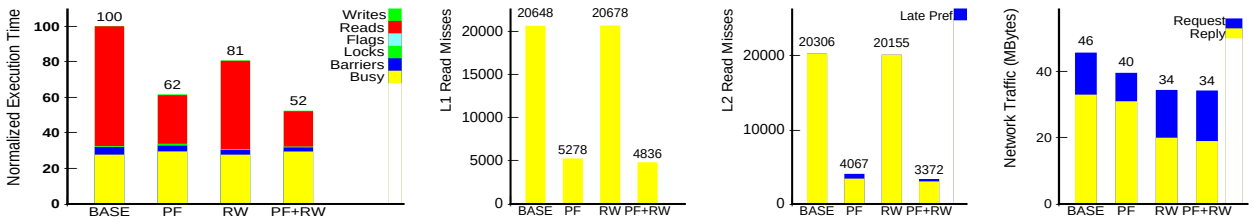
MCS locks: An MCS lock is efficient under moderate to high lock contention [13]. Processors needing a lock form a queue and spin locally on different variables. The processor holding the lock releases it by writing directly to the variable of the next processor in the queue, using `WriteSend` in the RW version. If no processors are waiting, then a global lock variable is reset, using `WriteThrough` in the RW version.

We compare the overhead of an MCS lock on Base and RW, using a method similar to that used by Lim and Agarwal [12]. We measure the total time for processors to execute 1000 acquire-release pairs each, with a 200-cycle holding time for the lock to model a small critical section. There is a uniformly distributed delay between 0 and 1000 cycles after a release, before the same processor tries to reacquire the lock. We also measure the same execution time when using a zero-latency spin lock [12]. The difference between the two times, divided by the number of locks acquired per processor, is the average overhead for the MCS lock.

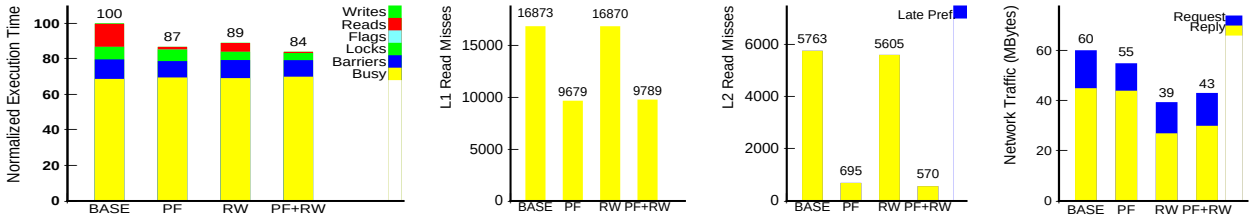
Figure 3(a) shows that `WriteSend` reduces the overhead by about 60% except at very low contention (less than 4 processors accessing the lock). This is because the lock transfer in RW requires a single `WriteSend` operation instead of the three messages required in Base (invalidation, read miss, and data response). In fact, the combination of `WriteSend` and MCS locks achieves essentially the same efficient communication pattern as hardware queue-based locks such as QOLB [5][4], but with much simpler and more general-purpose hardware support.

Tree barriers: Our tree barrier implements a full barrier using a binary tree. Since the identity of the processor waiting at each node is known, `WriteSend` operations can be used to efficiently wake up each processor once all processors have arrived at the barrier. We estimate the overhead of the barrier algorithm as the average time taken for all processors to exit the barrier after the last processor has entered it. We measure this overhead averaged over a series of twenty barriers separated by random delays to randomize the order in which processors enter each barrier. Figure 3(b) shows that `WriteSend` reduces the overall barrier overhead by 42% on a system with 32 processors. Again, the savings in RW are

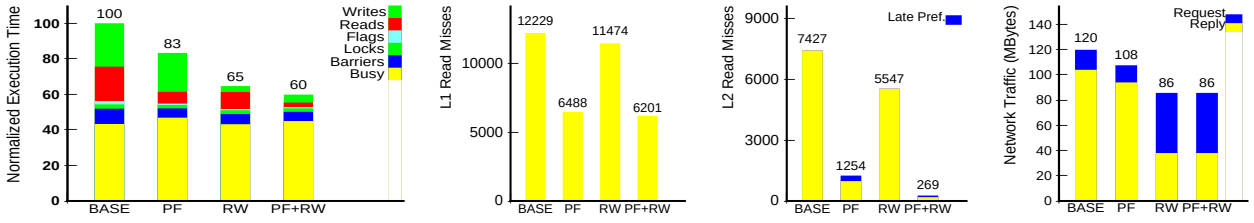
[4]This was observed by John Mellor-Crummey.



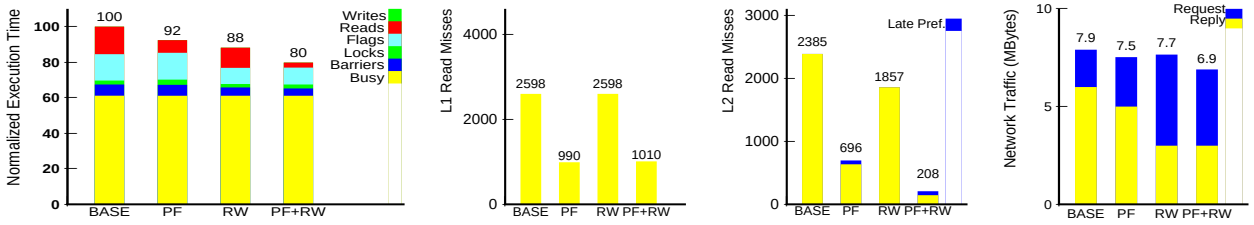
(a) MP3D



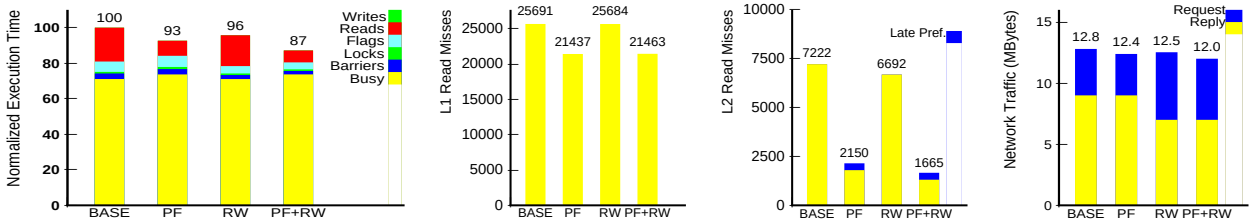
(b) Water



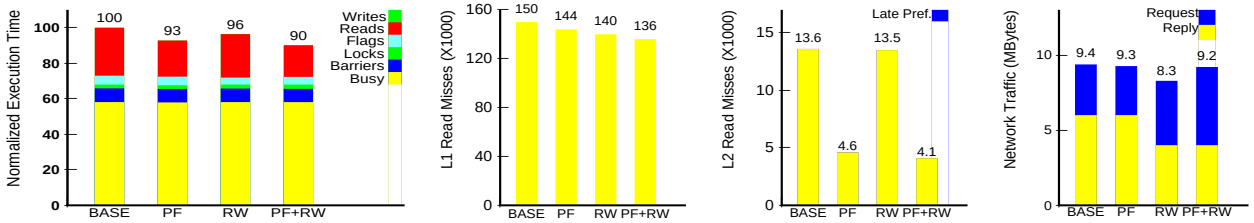
(c) Radix



(d) Erlebacher (Pipeline Phase)



(e) Erlebacher (Full Application)



(f) MICCG3D

Figure 4: Results for Applications

obtained because `WriteSend` achieves the same efficient data transfer pattern as for MCS locks at each node of the tree. Kranz et al. show similar results for a tree-barrier using fine-grain processor-to-processor messages in Alewife, although those messages are not cache-coherent and user-level handlers are required to handle them at the receiver [9]. All our applications use our tree barrier implementation, with `WriteSend`s in the RW and PF+RW versions.

4.3 MP3D

MP3D is a simulation of particles in rarefied flow. The two primary data structures are particles and an array of cells representing physical space. The main computation loop iterates over the particles, computing the new cell that each particle must be moved to. This requires updates of the particles and the current and new cells of the particles. The particles are equally partitioned among the processors.

PF uses exclusive prefetches for the particles and their current cells, but not for the new cell of a particle because the address of this cell is computed just before it is accessed. Scheduling the exclusive prefetches is difficult. Prefetching too early can invalidate active data from another processor's cache. On the other hand, prefetching too late incurs a high penalty because of the relatively high frequency of misses and large latencies. After extensive experimentation, we achieved the best results by prefetching particles 5 iterations ahead and cells 2 iterations ahead. Nevertheless, PF still results in several (640) late prefetches at the L2 cache.

RW uses `WriteThrough` to update the cells, which reduces read time by converting dirty misses to clean misses (i.e., converting 2-hop or 3-hop misses into local or 2-hop misses respectively). The amount of network traffic is smaller than in Base because only the modified words, and not the entire cache line, must be written out to the directory. Overall, RW decreases execution time by 19% compared to Base.

PF+RW combines the optimizations used in PF and RW. `WriteThrough` brings the cell data closer to the next reader, reducing the average latency seen by prefetches from 202 to 154 cycles. This reduces the number of late prefetches from 640 to 324. Furthermore, we can now use shared prefetching for cells, which eliminates early invalidations at other processors and results in fewer misses than PF. PF+RW eliminates substantially more read and barrier overheads than PF alone, and reduces overall execution time by 16% relative to PF. Because of the highly unpredictable accesses to cells, however, some part of the read latency could not be eliminated or hidden.

4.4 Water

Water is an N-body molecular dynamics simulation that evaluates forces and potentials in a system of water molecules. The most time-consuming phase is the intermolecular force computation, in which each processor executes an outer loop examining molecules assigned to it and an inner loop examining half the molecules in the system. If the distance between the pair of molecules examined is greater than a predefined value, the force vectors of the two molecules are updated in separate critical sections.

PF prefetches the displacement vectors of the inner-loop molecules, one inner-loop iteration ahead. If the forces must be updated, PF uses exclusive prefetches for the forces of the inner-loop molecule. The outer-loop molecule is prefetched once every outer-loop iteration. We find that read stall time is almost eliminated. The few remaining L2 cache misses are mostly due to the outer-loop molecule which is occasionally modified by other processors or replaced due to a cache conflict. Network traffic improved slightly because of exclusive prefetching.

RW uses `WriteThrough` for the force update of the inner-loop molecule and `WriteThroughInv` for the release of the corresponding (Test&Test&Set based) lock. We do not use remote writes for the outer-loop molecule or its lock since these are likely to be reaccessed by the same processor. The remote writes reduce the read and lock synchronization latency seen by the next reader, reducing execution time by 11% compared to Base.

PF+RW combines the optimizations used in PF and RW, although the overall improvement in execution time relative to PF is small. Like MP3D, the additional gain in read stall time comes from fewer L2 misses due to not requiring exclusive prefetch, and from reducing the latency seen by the prefetch from 144 to 90 cycles. The impact on read time is much smaller than in MP3D because there are few late prefetches in PF. The use of `WriteThroughInv` for releases reduces lock stall time slightly compared to PF.

4.5 Radix

Radix performs a radix sort on integer keys, using one iteration for each radix-r digit. Most of the communication occurs in three phases. The first is a logarithmic prefix sum computation where each processor collects and sums rankings for the keys from processors to its left. The second is a permute phase, where a processor copies its keys from the current key array to their newly sorted positions in another key array, from where they will be read in the next iteration. This phase has extensive false sharing and random all-

to-all communication, causing high contention, write buffer overflows, and substantial processor write stall time. Finally, many of the subsequent reads to the new key array incur misses because the keys are likely to have been last written by another processor.

PF prefetches non-local partial sums in the prefix sum, the local keys to be permuted in the permute phase, and other local data in other loops, resulting in reduced read stall time. PF also reduces barrier time by speeding up the computation in the prefix sum phase. PF does not address the high write stall time in the permute phase because (i) exclusive prefetches are not used for the writes since their addresses are not known enough in advance, and (ii) write stall time occurs due to high contention caused by false sharing which is not addressed by prefetching. Some L2 read misses also remain in the permute phase since their addresses cannot be generated early enough to prefetch.

RW uses `WriteSend` in the prefix sum phase for the writes of the partial sums and the flag synchronization, moving the data directly to the consumer as soon as it is produced. This reduces read latency and load imbalance. The major gains in RW, however, come from using `WriteThrough` to reorder the keys in the permute phase. Because `WriteThrough` does not acquire ownership, it greatly alleviates false-sharing which in turn substantially reduces network traffic and directory contention, and results in faster completion of writes.[5] By not getting ownership, `WriteThrough` also eliminates some capacity misses in the L2 cache since it does not bring in a line if it is not present in the cache. Finally, `WriteThrough` converts dirty misses to clean misses for subsequent reads in the next iteration, reducing read stall time. Overall, RW reduces execution time by 35% compared to Base.

PF+RW inherits the reduction in L2 cache misses from PF and RW, and the reductions in write stall time from RW. Furthermore, `WriteThrough` reduces average prefetch latency from 159 to 126, which results in fewer late prefetches at the L2 cache (80 in PF+RW instead of 271 in PF). Overall, PF+RW gives the highest benefits for Radix, reducing total execution time by 28% relative to PF.

4.6 Erlebacher

Erlebacher is a 3-D partial differential equation solver based on Alternating-Direction-Implicit (ADI) integration. The data and computation for the main arrays are distributed by assigning blocks of consecutive X-Y planes to each processor. Communication occurs only in one phase, which consists of one loop with coarse-grain communication followed by two fine-grain pipelined loop nests with an intervening short reduction. In the pipelines, flags signal completion of each pipeline stage. Multiple values typically need to be communicated per pipeline stage to achieve the optimal balance between communication overhead and start-up cost of the pipeline. The optimal number of values is 16 (one cache line) in all four systems.

In Erlebacher, the fine-grain pipelines are the only cases (other than tree barriers) where remote writes are applicable. Furthermore, these pipelines are typical of the communication patterns that arise in loops containing recurrences. We therefore first focus on results for the two pipelined loop nests along with the intervening reduction (Figure 4(d)), before discussing the full application.

PF prefetches the local values in the pipelines, which reduces some of the read stall time. We cannot prefetch data that is produced in the previous pipeline stage or in the reduction which occurs between the two pipelined loop nests because the data is produced immediately before it is read. Therefore, a significant part of the read stall time and all of the synchronization stall time remains.

RW uses `WriteSend` for the boundary data and flag writes in the pipelines since these locations are read soon after they are written and the next consumer is known to the producer. Consequently, both read and flag stall times are reduced compared to Base. Nevertheless, significant flag time remains because: (i) there is startup cost and load imbalance in the pipeline, and (ii) the flag release sees the full latency of the buffered `WriteSend`s, potentially delaying the subsequent acquire. Overall, RW reduces execution time by 12% compared to Base.

PF+RW combines the gains of PF and RW because they affect different sources of overhead. The only remaining L2 cache misses are in the reduction operation. We did not optimize the reduction since it did not form a significant part of the application. Overall, PF+RW reduces execution time by 13% compared to PF.

Figure 4(e) shows the corresponding results for the entire application. In PF, all the local reads are prefetched. The prefetch instructions result in some overhead which is reflected in higher busy times. There is only a small reduction in L1 cache misses because we do not prefetch data that is expected to be in the L2 cache. There are more L2 cache misses

[5] The extra buffering provided by the coalescing buffer was not responsible for these improvements since there is almost no spatial locality in the permute phase. To confirm this, we ran Base and PF with larger write buffers and found a negligible improvement in performance [1].

than in the pipeline phase because of conflicts within the L2 cache. The improvements from remote writes are small because remote writes apply only to one phase in the application and the original application itself does not have much communication or synchronization overhead. Nevertheless, the combination of PF+RW eliminates most of the overhead (other than cache conflicts and load imbalance) in this application.

4.7 MICCG3D

MICCG3D solves Laplace's equation on a three dimensional grid using the preconditioned conjugate gradient method. The Solve phase performs forward and backward substitution producing fine-grain recurrences. This phase is parallelized by pipelining the computation, using flags to synchronize the pipeline stages, similar to the pipelines of Erlebacher.

We use prefetches and remote writes as in Erlebacher. There are, however, two key differences. First, there are severe cache conflicts in our direct mapped L1 cache, but the method of data access made it too complex to pad the array without a significant rewrite of the code. Prefetching is unable to address these conflict misses. Second, RW is unable to reduce the L2 cache misses relative to Base because fewer misses at the consumer due to `WriteSend`s are offset by extra misses at the producer. The extra misses occur on reads that follow a remote write to the same line, because a remote write does not bring the line into the L2 cache. This is a drawback to not acquiring ownership on a remote write. In PF+RW, prefetching is used to eliminate these misses.

Overall, due to the large number of conflict misses, there is not much impact of our optimizations on the full application. Within the pipelines, we find that our optimizations are able to eliminate virtually all L2 cache misses in the PF+RW version, but again due to conflict misses, the decrease in execution time for the pipelines is only 5% relative to PF.

4.8 Summary

Our results highlight the benefits and limitations of remote writes, used alone or in combination with prefetching. The principal benefits of remote writes are as follows. First, with irregular sharing, `WriteThrough` is effective at reducing the latency of a subsequent read. When combined with prefetching, `WriteThrough` reduces the number of late prefetches by reducing the latency that must be hidden by the prefetches. Second, since remote writes do not acquire ownership, they reduce both network traffic and false sharing. For the same reason, in many cases, `WriteThrough` removes the need for exclusive prefetching; this reduces cache misses by delaying invalidations. Third, `WriteSend` is effective in handling fine-grain, producer-consumer sharing patterns that are not amenable to prefetching. Finally, remote writes, unlike prefetching, usually do not require many additional instructions.

The principal limitations of remote writes uncovered by our experiments are as follows. Not acquiring ownership on a remote write becomes a disadvantage if the producer reads the same data later; the resulting misses, however, can be hidden with prefetching. `WriteSend` is most effective for tightly synchronized fine-grain communication, but in these cases the latency of buffered remote writes can delay the subsequent release and hence the following acquire. This reduces the effectiveness of `WriteSend`. To address both of the above limitations, regular writes (which get ownership) can be used in the main computation. Then, a "dummy" remote write immediately *after* the release can be used to send the data to the reader. A final limitation of remote writes is that, unlike prefetching, they are not applicable to uniprocessors.

Overall, however, remote writes are able to combine with prefetching to address virtually all the sources of memory system overhead, except for conflict misses, in the applications and kernels we studied.

5 Related Work

Three primitives have been proposed that provide functionality similar to `WriteSend`: the DASH *deliver* [11], the *forwarding write* [16, 17], and the *PSET_WRITE* [19]. The most extensively studied is the *forwarding write* primitive, but it was evaluated as the sole producer-initiated primitive. A performance study of applications from the Perfect Club Benchmarks examined codes containing *doall* loop parallelism [16]. Surprisingly, the study found that when forwarding and prefetching were combined, the performance of each application fell in between that of forwarding alone or prefetching alone. The two principal reasons appear to be that the study used single-word cache lines, and used forwarding even in cases where conflicts might arise at the destination processor(s). The former increased the overhead of prefetching and caused the combined system to underperform forwarding on some applications. Conversely, the latter caused the combined system to underperform prefetching on some other applications. In contrast, we find that the proper use of a combination of prefetching and remote writes consistently provides the best performance, compared to using either primitive alone. In addition to removing the above two limitations of the previous study, there are two key reasons that its results differ from ours. First,

that study uses a single producer-initiated primitive, whereas we show that the combination of `WriteSend` and `WriteThrough` is key to providing consistently improved performance over prefetching alone. Second, that study focuses on applications with coarse-grain *doall* parallelism, whereas we also examine fine-grain parallelism (*doacross* loops and two important synchronization kernels) and irregular sharing patterns.

Koufaty et al. proposed a detailed compiler algorithm for forwarding, and showed significant improvements due to forwarding [8]. Although this work discusses how forwarding compares with prefetching, it does not include prefetching in the performance evaluation. That study also mentions, but does not evaluate, several optimizations used in this paper, namely coalescing forwarded writes, forwarding only to the L2 cache, and forwarding only to memory for dynamically scheduled loops.

Another study examined the performance impact of forwarding and prefetching for critical sections alone [24]. For five applications from SPLASH, including Water and MP3D, the study concluded that forwarding provides little additional benefit over prefetching. Our experiments, however, show that using `WriteThrough` operations for such irregular sharing patterns provides significant additional benefits when combined with prefetching.

The DASH *deliver* [11] and the *PSET_WRITE* [19] primitives, are both similar to the forwarding write. No performance results were presented for the former. The latter was studied for one application for which it provided insignificant gains, and was not compared with fine-grain prefetching.

A difference between `WriteSend` and the above primitives is that the latter use a bit vector to specify multiple destinations, while `WriteSend` only specifies a single processor, as discussed in Section 2.1.

Write-update and hybrid update-invalidate protocols (under hardware or software control) are another mechanism for producer-initiated communication. Software primitives for updates include the KSR1 *poststore* [21], the DASH *update* [11], the *SEL_WRITE* [19], and *notify* [5]. The *competitive-update* protocol is a hardware-controlled hybrid update-invalidate protocol [7]. Dahlgren and Stenstrom have shown that the addition of a write-cache (similar to a coalescing buffer) can reduce coherence traffic, allowing hybrid update-invalidate protocols to significantly outperform write-invalidate protocols [3]. All of the above schemes use an update operation to transfer a written value directly into the caches of all the processors that currently contain a copy of the line; caches periodically get invalidated based on various runtime heuristics. Raynaud et al. have proposed a different variation, called *distance adaptive protocols*, where hardware keeps track of a superset of current caches holding the line; on a write, a fairly complex runtime heuristic determines whether to send updates or invalidates to the recorded set of caches [20].

The primary difference between the above update schemes and remote writes is that most update schemes do not give the programmer or compiler direct control to update a specific processor cache for selectively exploiting static communication and synchronization patterns. For example, for programs like Erlebacher that sweep through the entire cache in each major loop nest, an update would not be useful, whereas `WriteSend` is effective when data is read soon after it is written by another processor. Finally, update protocols by themselves are not effective in handling unpredictable and irregular accesses, but these are partially optimized using `WriteThrough`.

Our `WriteThrough` primitive is similar to the *check-in* primitive of the CICO model [6]. Unlike writes with CICO, however, `WriteThrough` does not require ownership. The CICO model does not provide any operation similar to `WriteSend`. Dynamic self-invalidation is a hardware-controlled mechanism analogous to check-in [10]. Skeppstedt and Stenstrom developed compiler algorithms to insert instructions similar to `WriteThrough` to update memory and reduce coherence overhead [23]. Performance studies of these primitives have not evaluated the additional benefits of the primitives over prefetching.

6 Conclusions

This study evaluates the performance of fine-grain producer-initiated communication, both with and without fine-grain software prefetching in a base cache-coherent shared-memory system. We use a combination of two producer-initiated primitives, `WriteSend` and `WriteThrough`, collectively referred to as remote writes. Our principal results are as follows.

- For all our kernels and applications, remote writes consistently improved performance relative to the base system, both with and without prefetching. These performance benefits were accomplished with lower network traffic, reduced L2 cache miss rates, and lower miss latencies.

- `WriteThrough` is highly effective at reducing read stall time for irregular applications, where it cooperates well with prefetching to reduce the latency that must be hidden by a prefetch.

- `WriteSend` is effective for key synchronization primitives and tightly synchronized computations like pipelined loops. These did not form a significant part of our applications, however, and the benefits of `WriteSend` in full applications were relatively small. Nevertheless, the communication overheads addressed by `WriteSend` are difficult, if not impossible, to address with prefetching.
- The combination of remote writes and prefetching eliminated most of the memory system overhead in our applications except cache conflicts in the direct-mapped L1 cache.

In our experiments, we model a fairly conservative processor. Current processors aggressively exploit instruction-level-parallelism; e.g., using multiple issue, dynamic scheduling, and non-blocking reads. Recent work has shown that such techniques make memory stall time more dominant in shared-memory systems [15]. This would make techniques such as remote writes even more important for future systems.

Finally, our suggested implementation of remote writes adds little complexity to the base system. In particular, it does not introduce new classes of races in the coherence protocol, and does not need additional mechanisms for deadlock avoidance.

7 Acknowledgements

We thank Todd Mowry for a version of MP3D with prefetching, Preston Briggs and Steve Scott for advice on network latencies, and Parthasarathy Ranganathan for valuable feedback on earlier drafts of this paper.

References

[1] H. Abdel-Shafi. Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. Master's thesis, Rice University, 1997.

[2] R. G. Covington et al. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1, 1991.

[3] F. Dahlgren and P. Stenstrom. Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors. *JPDC*, 26, 1995.

[4] J. Edmondson et al. Internal Organization of the Alpha 21164. *Digital Technical Journal*, 7(1), 1995.

[5] J. R. Goodman et al. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS-III*, 1989.

[6] M. D. Hill et al. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors. *ACM TOCS*, 11(4):300–318, 1993.

[7] A. R. Karlin et al. Competitive Snoopy Caching. *Algorithmica*, (3), 1988.

[8] D. Koufaty et al. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *ICS*, 1995.

[9] D. Kranz et al. Integrating Message-Passing and Shared-Memory: Early Experience. In *PPoPP*, 1993.

[10] A. Lebeck and D. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *22nd ISCA*, 1995.

[11] D. Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3), 1992.

[12] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *ASPLOS-VI*, 1994.

[13] J. M. Mellor-Crummey and M. L. Scott. Synchronization Without Contention. In *ASPLOS-IV*, 1991.

[14] T. C. Mowry et al. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS-V*, 1992.

[15] V. S. Pai et al. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. In *HPCA-3*, 1997.

[16] D. Poulsen. *Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[17] D. Poulsen and P.-C. Yew. Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors. In *ICPP*, 1994.

[18] U. Rajagopalan. The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors. Master's thesis, Rice University, 1994.

[19] U. Ramachandran et al. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. In *Supercomputing'95*, 1995.

[20] A. Raynaud et al. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors. In *HPCA-2*, 1996.

[21] E. Rosti et al. The KSR1: Experimentation and Modeling of Poststore. In *SIGMETRICS*, 1993.

[22] J. P. Singh et al. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1), 1992.

[23] J. Skeppstedt and P. Stenstrom. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *PACT*, 1995.

[24] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Supercomputing '96*, 1996.

[25] S. C. Woo et al. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *ASPLOS-VI*, 1994.

[26] S. C. Woo et al. The SPLASH-2 programs: Characterization and Methodological Considerations. In *22nd ISCA*, 1995.