Available online at www.sciencedirect.com

ScienceDirect

J. Parallel Distrib. Comput. 67 (2007) 1256–1269

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

The impact of wrong-path memory references in cache-coherent multiprocessor systems

Resit Sendag[a,*], Ayse Yilmazer[a], Joshua J. Yi[b], Augustus K. Uht[a]

[a]Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI, USA
[b]Networking and Computing Systems Group, Freescale Semiconductor, Inc., Austin, TX, USA

Received 24 August 2006; received in revised form 27 December 2006; accepted 9 March 2007
Available online 24 March 2007

Abstract

The core of current-generation high-performance multiprocessor systems is out-of-order execution processors with aggressive branch prediction. Despite their relatively high branch prediction accuracy, these processors still execute many memory instructions down mispredicted paths. Previous work that focused on uniprocessors showed that these wrong-path (WP) memory references may pollute the caches and increase the amount of cache and memory traffic. On the positive side, however, they may prefetch data into the caches for memory references on the correct-path. While computer architects have thoroughly studied the impact of WP effects in uniprocessor systems, there is no comparable work for multiprocessor systems. In this paper, we explore the effects of WP memory references on the memory system behavior of shared-memory multiprocessor (SMP) systems for both broadcast and directory-based cache coherence. Our results show that these WP memory references can increase the amount of cache-to-cache transfers by 32%, invalidations by 8% and 20% for broadcast and directory-based SMPs, respectively, and the number of writebacks by up to 67% for both systems. In addition to the extra coherence traffic, WP memory references also increase the number of cache line state transitions by 21% and 32% for broadcast and directory-based SMPs, respectively. In order to reduce the performance impact of these WP memory references, we introduce two simple mechanisms—filtering WP blocks that are not likely-to-be-used and WP aware cache replacement—that yield speedups of up to 37%.

Keywords: Speculation; Multiprocessor; Wrong-path; Cache

1. Introduction

Shared-memory multiprocessor (SMP) systems are typically built around a number of high-performance out-of-order superscalar processors, each of which employs aggressive branch prediction techniques in order to achieve a high issue rate. During program execution, these processors speculatively execute the instructions after the predicted target of the branch. When a branch is mispredicted, the processor must restore its state to the state that existed prior to the mispredicted branch before the processor can start executing instructions down the correct path (CP). However, during speculative execution, *i.e.*, before the branch outcome is known, the processor speculatively issues and executes many memory references down the wrong-path (WP). Although these WP memory references are not allowed to change the processor's architectural state, they do change the data and instructions that are in the processor's caches, which can affect its performance.

Previous work [1,2,6,9,13,20–25,4,28–30] studied the effects that speculatively executed memory references have on the performance of out-of-order superscalar processors. These papers yield several conclusions. First, WP memory references may function as prefetches by bringing data into the cache that are needed later by instructions on the correct execution path [22,30,29,4]. Unfortunately, these WP memory references also increase the amount of memory traffic (*i.e.*, increased bandwidth consumption) and can pollute the cache with cache blocks that are not referenced by instructions on the CP [21,22,30,28]. Of these two effects, cache pollution—particularly in the L2 cache—is the dominant negative effect [21,22]. The results in

[*] Corresponding author. Fax: +1 401 782 6422.
E-mail addresses: sendag@ele.uri.edu (R. Sendag), yilmazer@ele.uri.edu (A. Yilmazer), joshua.yi@freescale.com (J.J. Yi), uht@ele.uri.edu (A.K. Uht).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.03.005

[22] also show that it is extremely important to model WP memory references, since they have a significant impact on the estimated performance.

In this paper, we focus on the effect that WP memory references have on the memory system behavior of SMP systems, in particular, for both broadcast-based and directory-based cache coherence. For these systems, not only do the WP memory references affect the performance of the individual processors, they also affect the performance of the entire system by increasing the number of cache coherence transactions, the number of cache line state transitions, the number of writebacks and invalidations due to WP coherence transactions, and the amount of resource contention (buffer usage, bandwidth, *etc.*).

To minimize the effect that WP memory references have on the performance of a SMP system, in this paper, we propose and evaluate a simple mechanism to filter out the WP cache blocks that are unlikely to be used on the CP. Our filtering mechanism uses temporal locality and L1 data cache evictions to determine if the corresponding cache block should be evicted from the L2 cache. In addition to this filtering mechanism, we also propose a cache replacement policy that is WP aware. More specifically, we add a field to each cache line to indicate whether or not that cache line was due to an instruction on the CP or the WP. When evicting a cache block from a set, evict the oldest WP cache block. Our results show that both of these simple mechanisms can significantly reduce the negative impact that WP memory accesses have on the performance of SMP systems.

This paper makes the following contributions:

1. It quantifies and analyzes the effect that WP memory accesses have on the performance of SMP systems, in particular, how WP memory accesses affect the cache coherence traffic and state transitions, and the resource utilization.
2. It proposes a filtering mechanism and a replacement policy to minimize the impact that WP memory references have on the performance of SMP systems.

The remainder of the paper is organized as follows—Section 2 describes the effects that WP memory references can have on the memory system behavior of SMP systems. Sections 3 and 4 present the details of the simulation environment and the simulation results, respectively. Section 5 describes our filtering mechanism and the WP aware replacement policy, and how they reduce negative effects of WP memory references. Section 6 describes some related work, while Section 7 concludes and suggests some future work.

2. WP effects

When designing a coherent shared-memory interconnect, the most important design decision is the choice of the cache coherence protocol. Popular protocols include: MSI (Modified, Shared, Invalid), MESI (Modified, Exclusive, Shared, Invalid), MOSI (Modified, Owned, Shared, Invalid), and MOESI (Modified, Owned, Exclusive, Shared, Invalid) [8]. When a processor accesses memory, the coherence state (*i.e.*, M, O, E, S, or I) of the cache lines in the processors' data caches may change. However, although the branch prediction accuracy of modern high-performance processors is high, when a branch misprediction does occur, loads on the mispredicted path access the memory subsystem, which can generate additional coherence traffic. While these extra state transitions do not violate the coherency of the data copies, they may degrade the performance of the cache coherence protocol and, subsequently, the performance of the memory subsystem, and, finally, the performance of the the SMP. In the remainder of this section, we discuss the potential effects that WP memory references can have for each of the aforementioned four cache coherence protocols (MSI, MESI, MOSI, and MOESI).

2.1. Replacements

A speculatively-executed load instruction that is later determined to be on a mispredicted path may bring a cache block into the data cache that replaces another block that may be needed by a load on the CP. As a result of these replacements, WP loads pollute the data cache [21,30], which may cause additional cache misses. Fig. 1, Step 2 shows an example of this situation. In this example, Processor 0 speculatively requests Block A, which causes the replacement.

On the other hand, these speculatively accessed memory references can potentially hide the memory latency for later CP misses, *i.e.* prefetching [22,30,29,4], which can improve the processor's performance.

2.2. Writebacks

In contrast to the writebacks caused by the CP replacements, in a SMP system, the coherence actions caused by WP memory references can also cause writebacks. For example, if the requested WP block has been modified by another processor, *i.e.*, its cache coherence state is M, a shared copy of that block is sent to the requesting processor's cache, which subsequently may cause a replacement. When the evicted block has a cache coherence state of M (exclusive, dirty) or O (shared, dirty) state, this causes an additional writeback, which would not have occurred if the WP load had not accessed memory in the first place. Step 2 in Fig. 1 illustrates this example. Extra writebacks, in addition to what is discussed above, may occur in MSI or MESI coherence SMPs. For these two protocols, if the requested WP block is in the M state in another processor's cache, a shared copy of that block is sent to the requesting processor's cache and also it is written back to the memory. Then the cache coherence state of that cache block is demoted from M to S in the original owner's cache. This additional writeback may not occur without the WP load.

2.3. Invalidations

The, loads issued down the WP may cause additional invalidations. For example, assuming a MOESI protocol, when a WP load instruction accesses a cache block that another processor has modified, the state of that cache block changes from M to O in the owner's cache and will have a cache coherence state

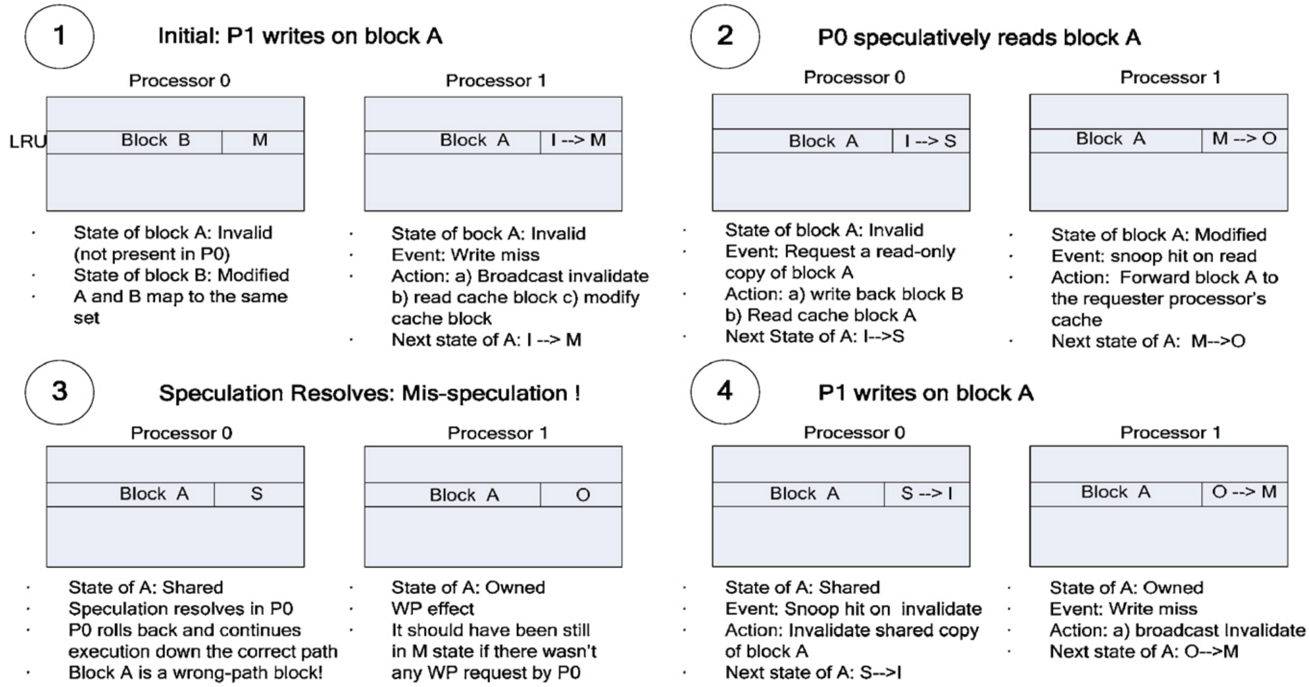
1
Initial: P1 writes on block A
Processor 0
Processor 1
LRU
Block B
M
Block A
I --> M
State of block A: Invalid (not present in P0)
State of block B: Modified
A and B map to the same set
State of bock A: Invalid
Event: Write miss
Action: a) Broadcast invalidate b) read cache block c) modify cache block
Next state of A: I --> M
2
P0 speculatively reads block A
Processor 0
Processor 1
Block A
I --> S
Block A
M --> O
State of block A: Invalid
Event: Request a read-only copy of block A
Action: a) write back block B b) Read cache block A
Next State of A: I-->S
State of block A: Modified
Event: snoop hit on read
Action: Forward block A to the requester processor's cache
Next state of A: M-->O
3
Speculation Resolves: Mis-speculation !
Processor 0
Processor 1
Block A
S
Block A
O
State of A: Shared
Speculation resolves in P0
P0 rolls back and continues execution down the correct path
Block A is a wrong-path block!
State of A: Owned
WP effect
It should have been still in M state if there wasn't any WP request by P0
4
P1 writes on block A
Processor 0
Processor 1
Block A
S --> I
Block A
O --> M
State of A: Shared
Event: Snoop hit on invalidate
Action: Invalidate shared copy of block A
Next state of A: S-->I
State of A: Owned
Event: Write miss
Action: a) broadcast Invalidate
Next state of A: O-->M

Fig. 1. Summary of the wrong-path effects on a SMP system for MOSI (Modified, Owned, Shared, Invalid) or MOESI (Modified, Owned, Exclusive, Shared, Invalid) coherence protocols. Blocks A and B map to the same cache: (1) initially, block B is in the Modified (M) state in P0's cache and it is the LRU (least recently used) block in the set, while block A is in P1's cache in the M state; (2) P0 speculatively reads block A. A Shared (S) copy of the block replaces block B and causes a writeback. The copy in P1's cache changes its state to O; (3) speculation turns out to be incorrect. Note the extra cache transactions and state transitions; and (4) P1 writes on block A and gets the exclusive ownership (state of block A is M now). This causes invalidation to be sent to the caches sharing block A.

of shared, S, in the requester's cache. If the owner of that cache block needs to write to it, the owner changes the state of that block from O to M and invalidates all other copies of that cache block. Therefore, as this example shows, changes in the cache coherence state of a cache block due to a WP load can cause additional invalidations. Fig. 1, Step 4 illustrates this example.

2.4. *Cache block state transitions*

In addition to causing additional replacements, writebacks, and invalidations, WP memory references can also cause transitions in the cache coherence state of a cache block. For example, when a WP memory reference accesses a modified cache block in another processor's cache, under the MOESI protocol, the cache coherence state of that block changes from M to O in the owner's cache. The state of that cache block changes back to M when the owner writes to that block. These changes in the cache coherence are due solely to the WP access. Therefore, in this case, a WP memory access in another processor results in two extra cache state transitions in the owner's cache (see Steps 2 and 4 in Fig. 1).

The extra cache block state transitions caused by WP memory references may degrade the performance. For example, when implementing a snooping coherence protocol, the operation of detecting a write miss, obtaining the bus, getting the most recent value, and updating the cache cannot be done as if it took a single cycle. This requires adding a number of transient states for pending write misses and write-backs (for a write-back cache). The controller will leave those states when the bus is available. A WP memory reference, which causes this type of extra transitions, competes with other CP requests to acquire the bus. The processor will also stall when it requests a block that is in transient state due to an earlier WP request. Such problems are slightly worse in a directory-based system that does not have a broadcast mechanism like a bus, which can be used to order all requests.

2.5. *Data/bus traffic and coherence transactions*

Due to these extra replacements, writebacks, invalidations, and changes in the cache coherence state, WP memory accesses increase the amount of traffic due to L1 and L2 cache accesses, as well as increasing the number of snoop and directory requests.

2.6. *Power consumption*

In the best case, even if WP memory references do not affect the performance of the SMP system, they still may increase system's overall power consumption [19].

Several previous studies proposed methods to reduce the power in snoop-based systems [19,18,27,10,11] by filtering

Table 1
Benchmarks and input data sets

Benchmark	Description	Input data set
fft	Complex 1-D FFT	64 K points
radix	Integer radix sort	2 M integers, radix 1024
ocean	Simulates large-scale ocean movements	128 × 128 ocean
water-spatial	Simulation of water molecules	512 molecules
em3d	Electromagnetic force simulation	400 K nodes, degree 2, span 5, 15% remote

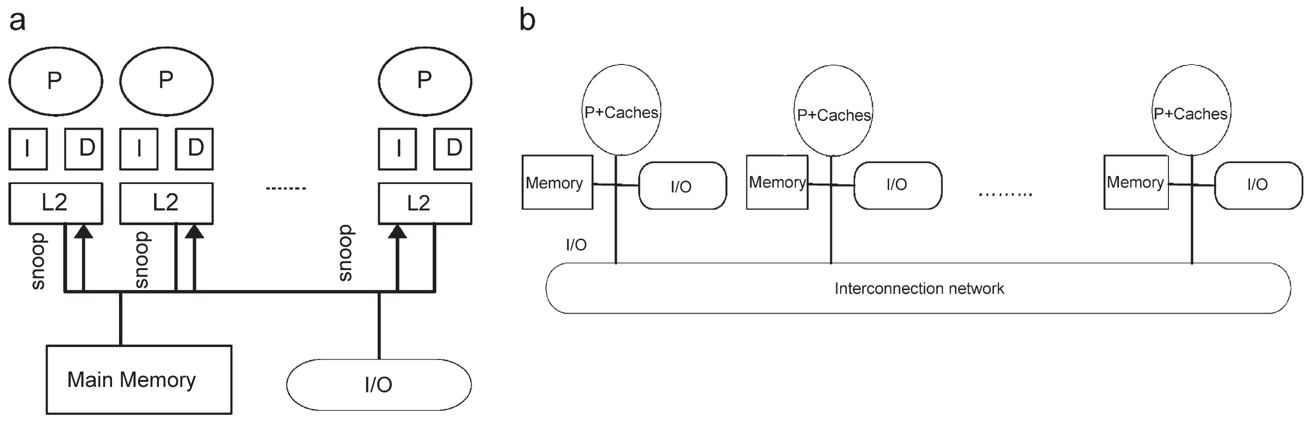

Fig. 2. (a) Broadcast and (b) directory based SMPs.

unnecessary snoops. In particular, Moshovos *et al*. [19] showed that filtering unnecessary snoops can reduce the total L2 cache power by 30%. Accordingly, reducing the cache line transitions and cache coherence traffic due to WP memory accesses should also reduce the power consumption. However, in this paper, we defer a detailed examination of the attendant power consumption implications to future work.

2.7. Resource contention

Finally, in addition to the aforementioned effects, WP memory accesses can also increase the amount of resource contention. More specifically, WP memory accesses compete with CP memory accesses for the multiprocessor's resources, such as request and response queues at the communication interconnect, and interprocessor bandwidth. The additional cache coherence transactions may increase the frequency of full service buffers. In this paper, however, we assume a sufficient network bandwidth to keep the network contention low. With the possible exceptions of *fft*, which uses all-to-all communication, and *em3d*, network contention was not a problem for the benchmarks that we studied in this paper. However, for other workloads, network contention could have a serious performance impact.

3. Experimental methodology

3.1. Benchmarks

Table 2
Broadcast (snoop)-based and directory-based SMP system parameters

Parameter	Value
Processors	16 UltraSPARC III processors
Processor parameters	2 GHz 15-stage pipeline, out-of-order execution
	8-wide dispatch/retirement
	256/128-entry ROB/scheduler
	10 cycle branch misprediction penalty
	GSHARE branch predictor with 4 K PHT
	64-entry return address stack
	32 Entry CAS and CAS exception table
L1 Caches	Split I/D, 32 kB 2-way, 128 Byte Blocks, with 2 ns access latency 32 Entry MSHRs
L2 Caches	Unified, 2 MB 2-way, 20 ns hit latency
	Exclusive L1 and L2s
Main Memory	4 GByte per bank, 240 ns DRAM latency
Interconnect	Hierarchical switch

Table 1 lists the five benchmarks that we used in this paper. The first four benchmarks are benchmarks from the SPLASH-2 benchmark suite [34],[1] while *em3d* [7] is an electromagnetic force simulation benchmark.

3.2. Simulated system configurations

In this paper, we evaluate a 16-processor SPARC v9 system running an unmodified copy of Solaris 9. We simulate both snoop-based and directory-based SMP systems with an invalidation-based cache coherence. We use the MOSI and MOESI cache coherence protocols, respectively, for the

[1] We were not able to simulate all SPLASH 2 benchmarks because we had problems with compilation and/or in adding the checkpoints/breakpoints to the SPLASH benchmark code for simulation on SIMICS.

snooping-based and directory-based SMP systems. Each node includes an aggressive, dynamically-scheduled, out-of-order processor core [16], two-levels of cache, coherence protocol controllers, and a memory controller [17]. Fig. 2 shows the block diagram of simulated directory and broadcast-based SMP systems and Table 2 lists the relevant processor and system parameters.

3.3. Simulation methodology

We collect our simulation results using the GEMS [16] extension to Virtutech's Simics [15], which is a full system simulator. GEMS adds cycle-accurate models of an out-of-order processor core [17], cache hierarchy, various cache coherence protocols, multibanked memory (unified or distributed), and various interconnection networks to the base-version of Simics.

GEMS uses timing-first simulation approach [17], in which functional and timing aspects of the simulators are decoupled. The timing modules interact with SIMICS to determine when SIMICS should execute an instruction. However, what the result of the execution of the instruction is ultimately dependent on SIMICS. The GEMS simulator is reported to be 100% correct, and the worst case performance error is 2.4% [16].

To avoid measuring the time needed for thread-forking, we begin our measurements at the start of the parallel phase by using Simics' functional simulation to execute the benchmarks until the start of the parallel phase. Then, we use first iteration of the loop to warm-up the caches and branch predictors. After the first iteration, we simulate the benchmark for an additional iteration to gather our simulation results.

4. Evaluating the WP effects

In this section, we evaluate the impact that executing WP memory references have on the caches, the communication between processors due to coherence transactions, and the overall performance of the SMP. To measure the various WP effects, we track the speculatively generated memory references and mark them as being on the WP when the branch misprediction is known.

4.1. L1, L2, and coherence traffic

In this section, we quantify the percentage increase in the L1 cache, L2 cache, and coherence traffic due to the WP memory references for 4- and 16-processor SMP systems. Fig. 3 shows the increase in the traffic between the processor and its L1 data cache and between the L1 cache and the L2 cache due to WP memory references, as a percentage of the total number of memory references, for broadcast-based SMPs. Fig. 4 does the same for directory-based SMPs.

Fig. 3 shows that, for a 4-processor broadcast-based SMP, WP loads increase the total number of L1 and L2 cache accesses by an average of 8% and 14%, respectively. For a 16-processor broadcast-based SMP, this increase is 15% for L1 and 35% for L2 cache accesses. For directory-based SMPs, Fig. 4 shows that these loads increase the percentage of L1 and L2 cache accesses by an average of 9% and 14%, respectively, for four processors, and 13% and 32%, respectively, for 16 processors. With 16 processors, for all benchmarks and for both SMP systems, the percentage increase in the number of L2 references is larger than the percentage increase in the number of L1 cache references. With four processors, however, except *em3d*, there is no such a trend. For *em3d*, while the percentage increase in the number of L1 cache accesses is negligible for both 4 and 16 processors and for both systems, the number of L1 misses increases by as much as 45%. Overall, 16-processor SMPs are affected by WP memory references much more than 4-processor SMPs are.

Fig. 5 shows that WP memory accesses increase the number of coherence transactions by an average of 18% and 32%, for 4 and 16 processors, respectively, for both broadcast and directory-based SMPs. For *em3d*, the coherence traffic increases by over 60%.

The results from Figs. 3–5 show that the extra traffic due to WP memory references increases as the number of processors increases. Therefore, for rest of the paper, we only analyze their effects on 16 processor systems.

4.2. Cache line replacements

From a performance point-of-view, WP memory references can have both a positive and negative effect on the processor's performance by either prefetching data into the caches or by polluting them [21,22,30,29], respectively. To determine the potential performance impact that WP memory references have in SMP systems, we categorize the misses caused by WP loads into four groups: *unused*, *used*, *direct miss*, and *indirect miss*. In the *unused WP block* category, the WP cache block is either evicted before being used or is never used by a correct-path. On the other hand, cache blocks in the *used WP block* category are eventually used by a CP memory reference. *Direct miss* cache blocks can severely degrade the systems performance because they replace a cache block that a later CP load accesses, but the WP block is evicted before being used. Finally, since unused WP misses change the LRU state of cache blocks in that set, which may eventually cause CP misses, we call these misses *indirect misses*. For example, A, B, C and D are cache blocks that map to the same cache set. Assume that in this example the: (1) cache is two-way set-associative cache that initially contains blocks A and B, (2) where B is the LRU block, (3) C is the WP reference, and (4) both A and D are on the CP. In this situation, the sequence of operations is as follows: WP block C replaces B, CP miss block D replaces A, CP miss block A replaces C. If WP reference for block C did not occur, then the CP reference for block A would have been a cache hit because block D would replaced block B instead. Fig. 6 illustrates this situation.

Fig. 7 classifies the WP-caused cache misses into the aforementioned four categories. The results show that 55–67% of the WP replacements in the L1 data cache and 12–36% of the WP replacements in the L2 are *used* in broadcast-based

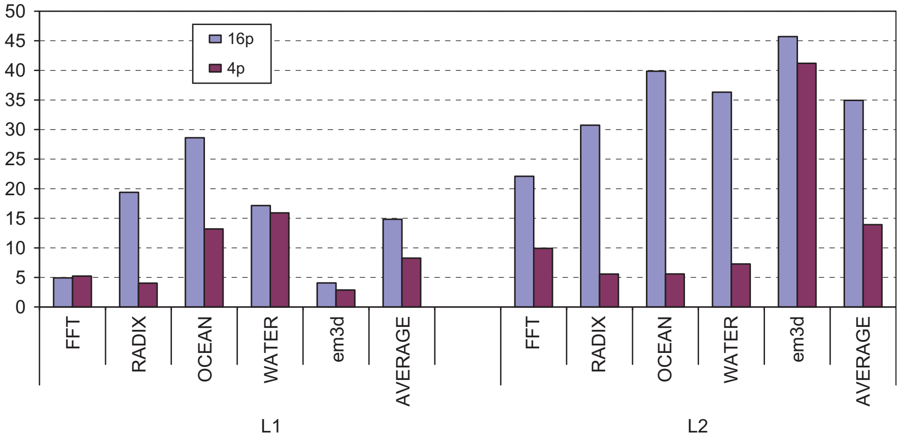

Fig. 3. Percentage of increase in L1 and L2 cache traffic for broadcast-based SMPs.

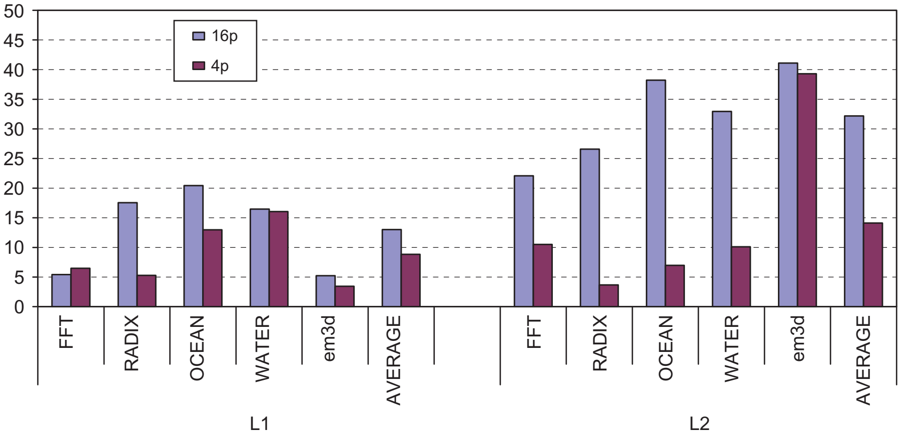

Fig. 4. Percentage increase in L1 and L2 cache traffic for directory SMPs.

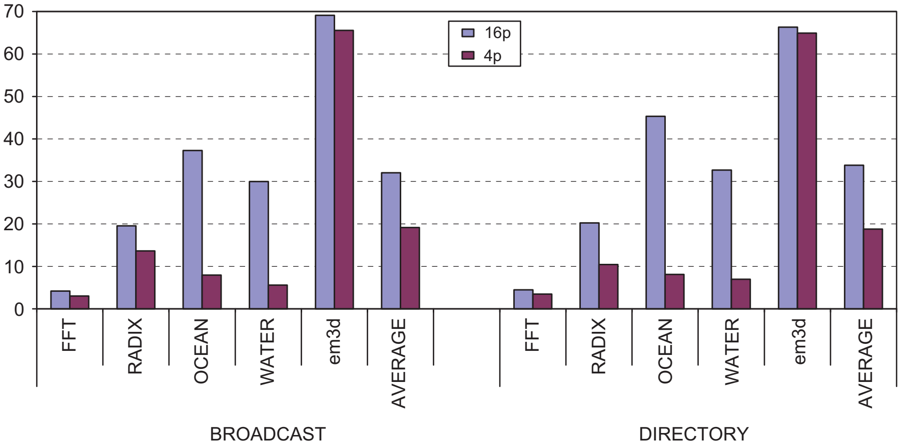

Fig. 5. Percentage of increase in coherence traffic for SMPs.

systems. *Direct misses* account for 5–62% of all WP replacements and account for a higher percentage of WP misses in broadcast-based SMP systems than for directory-based. Finally, *indirect misses* account for less than 5% of all WP misses for most of the benchmarks and systems tested.

It is important to note that *direct* and *indirect* misses are responsible for the pollution caused by the WP memory references. While they have similar effect on the L1 data cache for both broadcast and directory systems, their effects on L2 cache are different between the two SMP systems. For

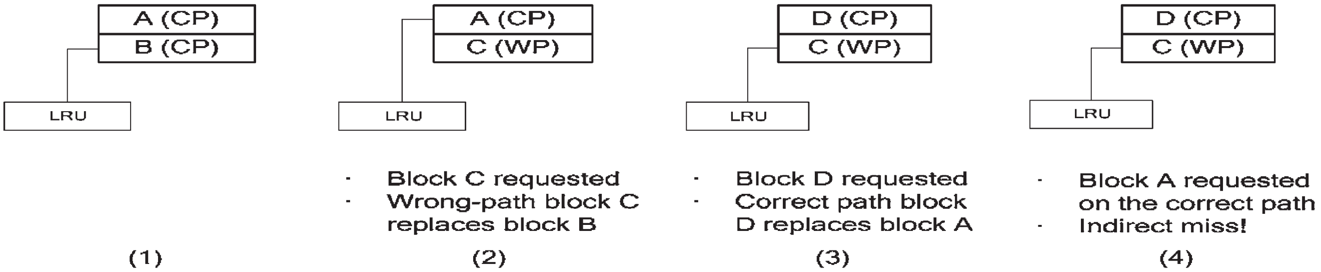

Fig. 6. Illustrating indirect miss caused by wrong-path memory references.

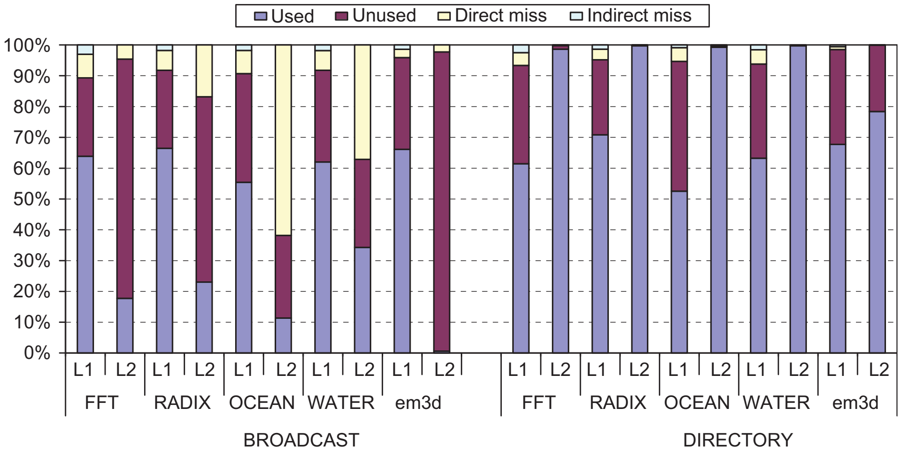

Fig. 7. Replacements due to wrong-path memory references.

directory-based, almost all of the L2 replacements are *used*, while the opposite is true for broadcast-based. This suggests that WP memory references have a greater effect on broadcast-based systems. However, a small number of remote misses caused by WP loads may have a disproportionately large performance impact in a directory-based system, as compared to a broadcast-based system.

4.3. Servicing cache coherence transactions

Broadcast-based cache coherence provides the lowest possible latency to retrieve data since misses can either be served by remote caches or shared memory. In contrast, in a directory-based SMP, misses can be served locally (including the local directory), at a remote home node, or by using both the home node and the remote node that is caching an exclusive copy, *i.e.*, a three-hop miss. The latter case has a higher cost because it requires interrogating both the home directory and a remote cache. Coherence misses account for most of the remote misses.

Figs. 8 and 9 show how the CP and WP cache coherence transactions are serviced for broadcast and directory-based SMP systems, respectively. The figures show that the results are similar for both SMP systems. Namely, remote caches service a greater percentage of the WP misses than for CP misses for all benchmarks except *em3d*. For those benchmarks, the percentage of misses serviced by remote caches varies from 12% to 80% for CP loads and 55–96% for WP loads. For the directory-based SMP, in all benchmarks, local memory services only a very small percentage of both CP and WP memory references.

4.4. Replacements and writebacks

As described in Section 2.2, WP replacements may cause extra writebacks that would not occur otherwise. Figs. 10 and 11 show the percentage increase in the number of replacements and writebacks due to WP memory references. Fig. 10 shows the percentage increase in the number of E (for directory MOESI) and S line replacements. E → I transitions—which increased by 2–63%—are particularly important since the processor loses the ownership of a block and, more importantly, the ability to silently upgrade its value, which can significantly increase the number of invalidations needed for write upgrades. For *em3d*, there is a large increase in both the replacements and writebacks.

Fig. 11 shows that WP memory accesses increase the number of writebacks from 4% to 67%. It is important to note that writebacks may result in additional stall cycles when an L2 cache miss occurs after the processor starts to perform a writeback, since it cannot begin to service the miss until the writeback completes.

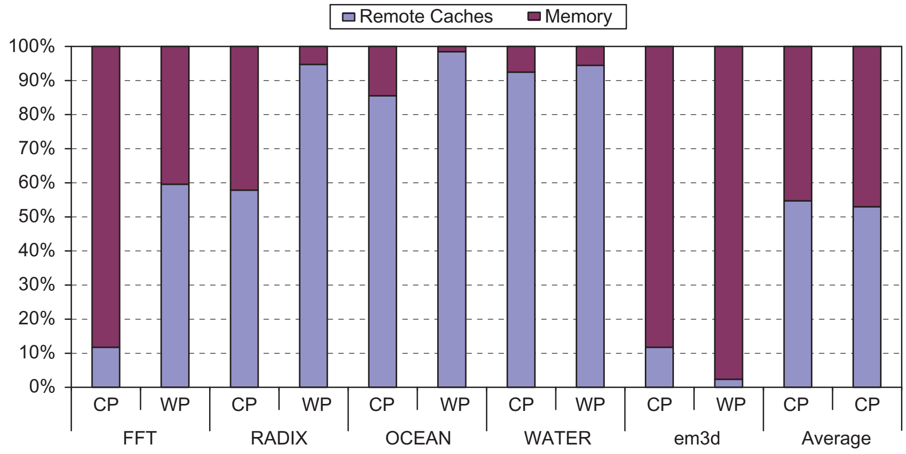

Fig. 8. Servicing coherence transactions for broadcast-based SMPs for the correct path (CP) and the wrong-path (WP).

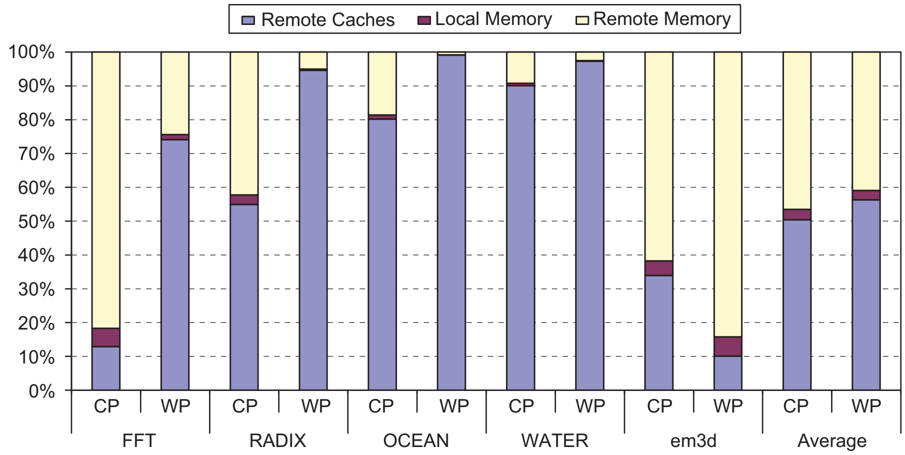

Fig. 9. Servicing coherence transactions for directory-based SMPs for the correct path (CP) and the wrong-path (WP).

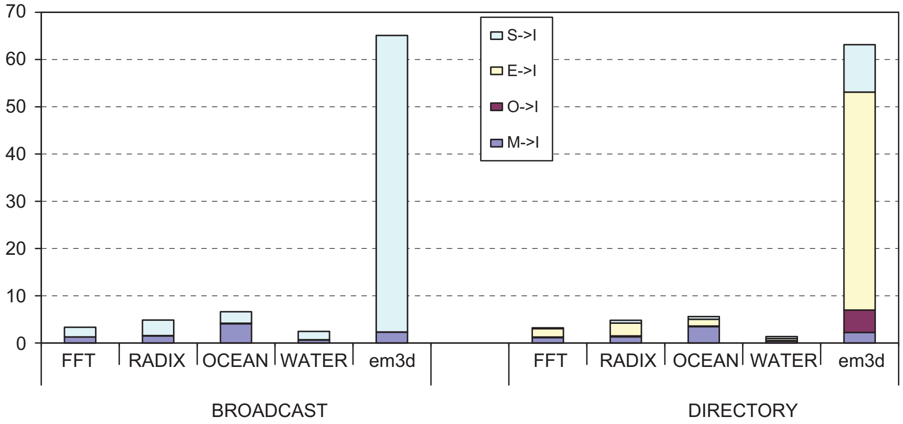

Fig. 10. Percentage increase in the number of replacements due to wrong-path references in broadcast and directory-based SMPs.

4.5. Cache line state transitions

Fig. 12 shows the impact that WP memory references have on the number of cache line state transitions. The results show that the number of cache line state transitions increase by 20–24% for a broadcast-based SMPs and by 27–44% for directory-based. Although the percentage increase is smaller for the broadcast-based system, the number of cache line state transitions is much higher to begin with.

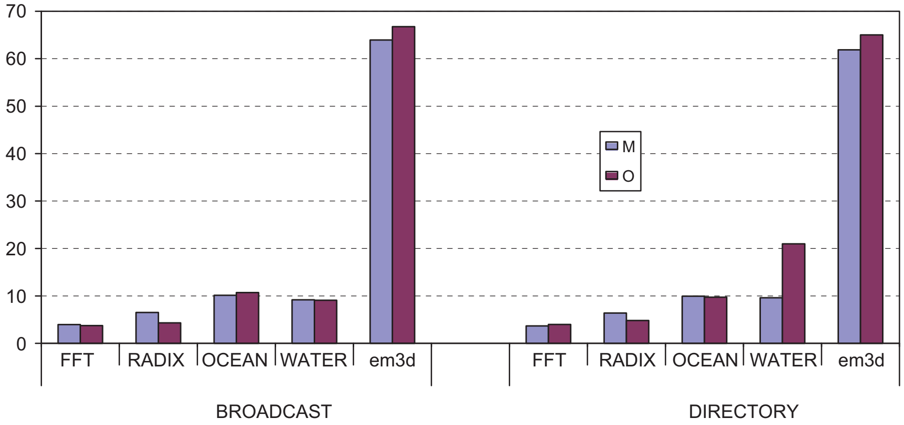

Fig. 11. Percentage increase in the number of writebacks due to wrong-path references in broadcast and directory-based SMPs.

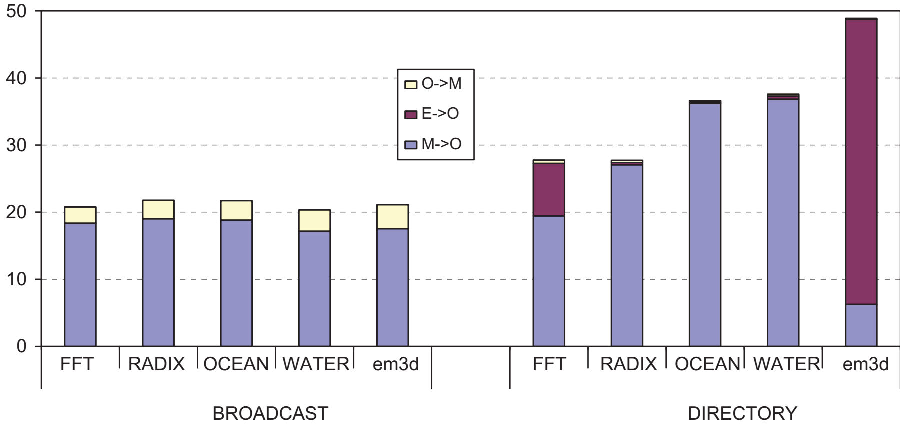

Fig. 12. Percentage increase in the number cache line transitions for MOSI broadcast-based and MOESI directory-based SMPs.

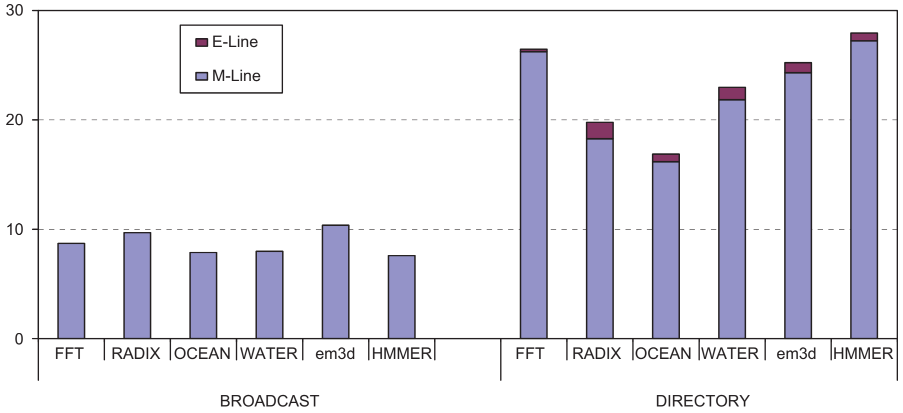

Fig. 13. Increase in the write misses and extra invalidations due to wrong-path references in broadcast and directory-based SMPs.

A processor loses ownership of an exclusive cache block (M or clean E) when another processor references it. In order to regain ownership, the processor has to first invalidate all other copies of that cache block, *i.e.*, S → I, for all other processors. Fig. 13 shows that there is 8–11% increase in the number of write misses due to WP references—each of which subsequently causes an invalidation—for broadcast-based SMPs; this percentage is higher, 15–26%, for the directory-based SMPs.

5. Filtering and replacement policies for WP memory references

In Section 4, we described the effects that WP memory references can have on the memory subsystem behavior of broadcast and directory-based SMP systems. In this section, we evaluate two enhancements that try to minimize the negative effects of WP memory references, while retaining their positive effects (*i.e.*, prefetching), to improve the performance of an SMP system without significantly increasing the complexity of the memory subsystem.

5.1. Marking the WP blocks

The first step to reduce the negative effects of WP memory references is to detect the WP requests. Mispredicted branches are usually resolved in the branch execution unit (BEU) before most of the WP L1 misses, and before almost all of the WP L2 misses complete [22]. Therefore, whether an L1 or L2 cache miss is down the WP is usually known before the block is placed into the cache. Most of the current processors use miss status holding registers (MSHRs) to track outstanding memory requests. Each MSHR entry stores the speculative tag for the missed load instruction. When a branch misprediction is signaled by the BEU, the speculative tag for the corresponding branch can be matched with the tags in MSHRs and marked as WP. In order to implement a WP-aware replacement policy, each cache block also needs a 1-bit to specify whether the block is brought into the cache due to WP or CP. This bit is set to CP by default. The WP loads which miss in L1 and that are already completed (*e.g.*, serviced by a L2 hit) before the branch resolution are not detected and thus marked as CP. However, with the help of a simple mechanism, almost all of the blocks brought by WP loads can be marked as WP. If a speculative load request misses in the L1 and is serviced by the L2, the L1-missed address may be kept in a small first-in–first-out (FIFO) queue (4–8 entries speculative load miss queue, SLMQ) when removed from the MSHRs giving more time to BEU to signal the misprediction. The WP addresses in this queue matching the mispredicted branch tag can then be used to access the data cache and mark the WP blocks. This operation can be done by probing the cache whenever there is available access port to the cache, thus it does not compete with the ordinary memory requests. Most of the blocks brought by WP loads can be marked as WP even without this FIFO queue. However, such a queue is useful when a predicted branch instruction's operands depend on a long latency operation (such as a load that misses in L2) to produce the operand value. In this case, we may not be able to capture the WP loads in the MSHRs because of the late branch resolution. Therefore, an SLMQ will be beneficial. On the other hand, another scenario may help on-time marking of load misses as WP. When the branch is resolved but cannot be committed because it is waiting for a long latency operation which is at the reorder buffer head to complete execution and commit, the out-of-order core continues execution. The WP loads in the load queue which were not ready to be issued can become ready after the branch resolution and can simply be marked as WP when placed in the MSHRs if they miss in L1. Figs. 14 and 15 show the basic operation of how to mark cache blocks as being from the WP and the implementation in the SMP.

5.2. A WP aware replacement policy

Based on the results in Section 4, we propose a WP-aware cache block replacement policy. To make the cache replacement policy WP aware, when a block is brought into the cache, it is marked as being either on the CP or on the WP. (There are several possible ways to design such a mechanism, one of which was discussed in the previous section.) Later, when a block needs to be evicted from that set in the cache, assuming that all cache blocks are valid (if not, an invalid block is "replaced" first), WP blocks are evicted first, on a LRU basis if there are multiple WP blocks. The WP block evicted from L1 data cache will now be written into L2 cache (exclusive L1–L2), however, when placed in L2, it will stay as LRU. This will ensure that the WP block that was not used in L1 will not reside in the L2 cache for very long (unless it is used). On the other hand, a WP block that services a CP reference is marked as if it was on the CP, thus excluding it from the WP replacement policy. If all cache blocks originated from a CP reference, then the LRU block in that set is chosen for eviction. Fig. 16 shows the algorithm for WP-aware replacement.

5.3. Reducing cache pollution via filtering

Our second proposed enhancement is a filtering mechanism that reduces the cache pollution due by *direct* and *indirect miss* WP references, and by evicting the *unused* WP blocks early. We apply our filtering mechanism to the L2 cache due to the long latency of L2.

We base our filtering mechanism on the observation that if a speculatively fetched cache block is not used while it resides in the L1 cache, then it is likely that block will not be used at all or will not be used before being evicted from the L2 cache [21].

In this paper, we evaluate exclusive L1 and L2 caches. A block that misses both in L1 and L2 allocates a line only in the L1 cache. Then, when a block is evicted from the L1 cache, it is written to L2.

Our filtering mechanism works as follows: If a WP block is evicted from the L1 cache before being used by a CP memory reference, it is allocated to the L2 cache only if its L2 set has an empty way, *i.e.*, at least one cache way is invalid. If not, then that cache block is discarded, *i.e.*, not allocated to the L2 cache, but written to memory only. A WP block that services a CP reference is handled in the same way as a CP block.

We can further filter WP blocks from being placed in L2 cache by canceling the WP references in the L2 cache request queue as soon as the misprediction is known. For example, if a requested block is an L1 cache miss, a request is sent to the L2 cache controller and placed in a request queue. At the time that the L2 cache controller processes this request, if it is known that the load instruction was on a mispredicted branch path, then this request is simply discarded without being serviced. (If this request were not discarded, it would cause an L2 miss and could possibly replace a valid block in the L2 cache.) However, if there is an invalid line in the set, the L2 cache controller services that WP memory reference and overwrites the invalid line. Otherwise, the L2 cache controller processes this request as usual.

5.4. Performance evaluation

Fig. 17 shows the speedup results for the enhancements described in Sections 5.2 and 5.3. In this figure, there are a total

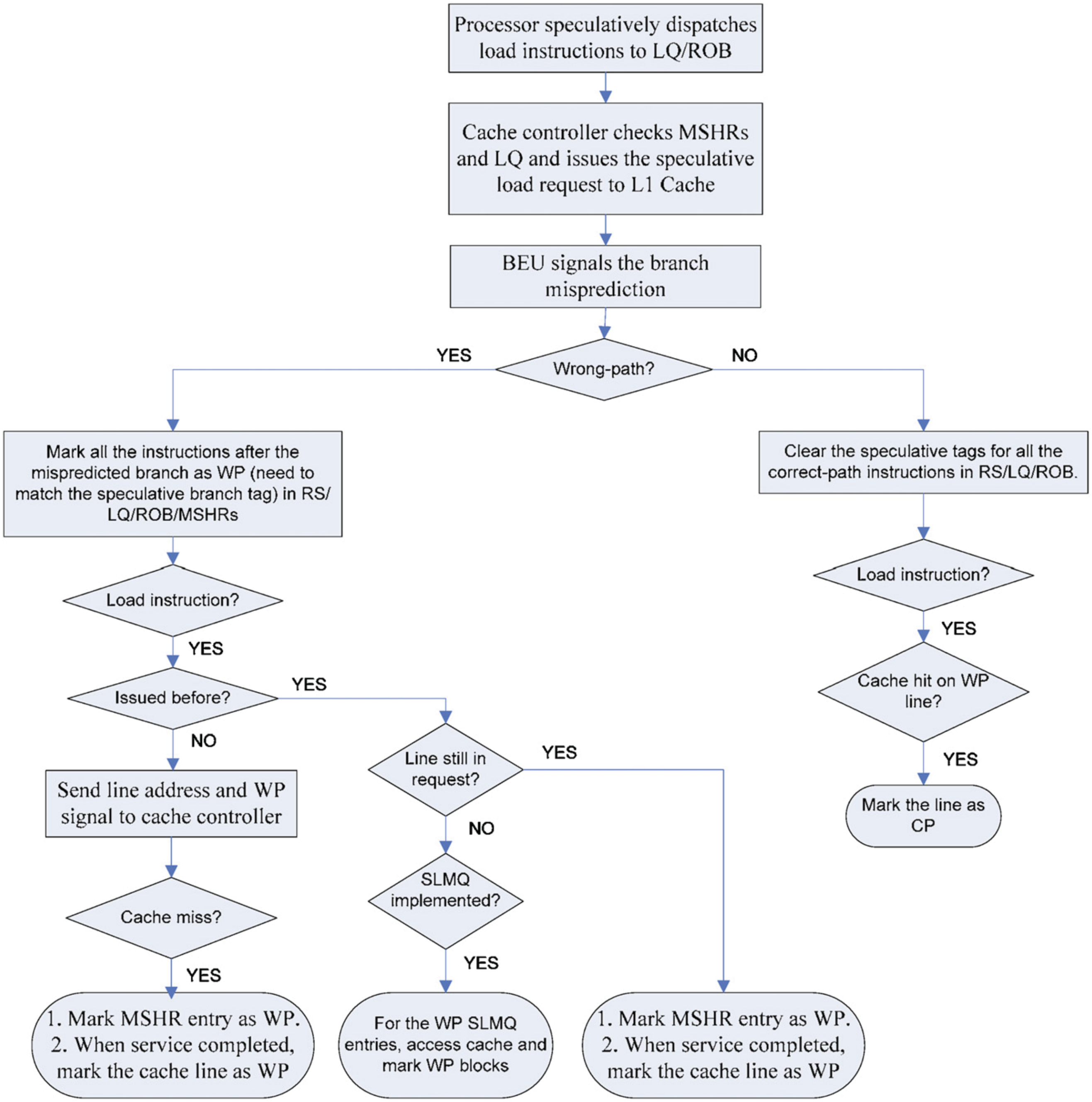
Processor speculatively dispatches load instructions to LQ/ROB
Cache controller checks MSHRs and LQ and issues the speculative load request to L1 Cache
BEU signals the branch misprediction
Wrong-path?
YES
NO
Mark all the instructions after the mispredicted branch as WP (need to match the speculative branch tag) in RS/LQ/ROB/MSHRs
Clear the speculative tags for all the correct-path instructions in RS/LQ/ROB.
Load instruction?
YES
Issued before?
YES
NO
Send line address and WP signal to cache controller
Cache miss?
YES
1. Mark MSHR entry as WP.
2. When service completed, mark the cache line as WP
Line still in request?
YES
NO
SLMQ implemented?
YES
For the WP SLMQ entries, access cache and mark WP blocks
1. Mark MSHR entry as WP.
2. When service completed, mark the cache line as WP
Load instruction?
YES
Cache hit on WP line?
YES
Mark the line as CP

Fig. 14. Basic operation of marking lines as WP in data cache. In figure, LQ: load queue; ROB: reorder buffer; MSHRs: miss status handling registers; SLMQ: speculative load miss queue; BEU: branch execution unit; WP: wrong-path; CP: correct-path.

of three enhancements: *replacement*, *filter*, and their combination, *i.e.*, *filter* + *replacement*.

The results in Fig. 17 show that a simple WP aware replacement policy may perform very well for some benchmarks. For example, for *water*, all three enhancements yield speedups of about 30% or more for the broadcast-based SMPs. Overall, the performance of the enhancements varies across benchmarks and systems. On average, filtering yields higher speedups than WP aware replacement, while also outperforming replacement for all benchmarks for directory-based SMPs. For broadcast-based SMPs, filtering performs better than WP aware replacement for *radix*, *water* and *em3d*. Employing a simple WP replacement policy does not significantly improve the performance of *ocean* and *fft*.

WP-aware replacement policy degrades the performance in some cases: 1% for *radix* (broadcast), 4% for *fft* (directory) and 8% for *water* (directory). The performance degradation is mainly due to the WP-aware replacements that are not useful because they reduce prefetching effect of useful WP blocks by replacing them first (this blocks may be used later by CP

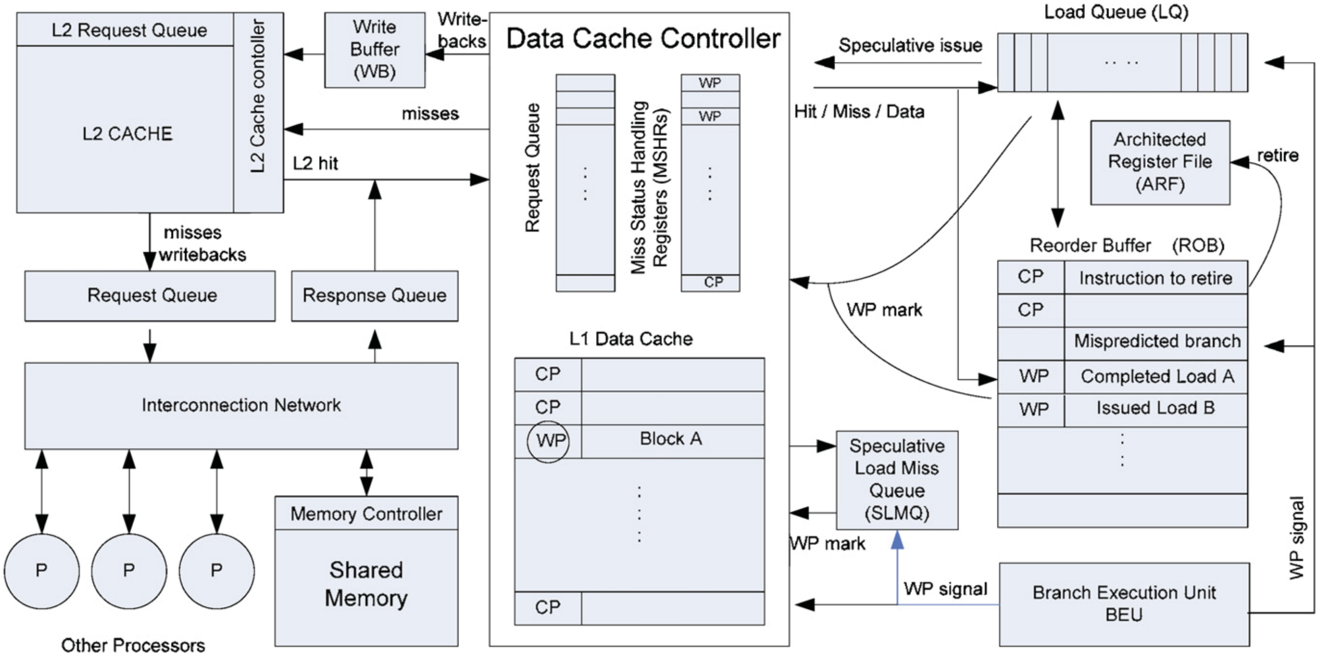

Fig. 15. The block diagram of the WP-aware SMP system. Some of the details are not shown for the sake of simplicity. The WP-aware cache system can also be designed for uniprocessor systems.

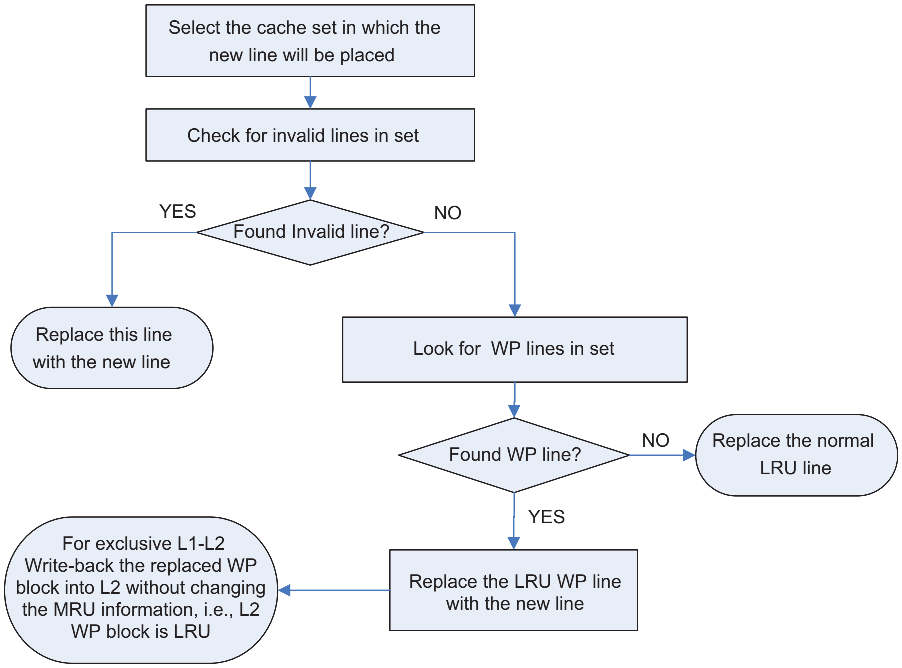
Select the cache set in which the new line will be placed
Check for invalid lines in set
Found Invalid line?
YES
NO
Replace this line with the new line
Look for WP lines in set
Found WP line?
NO
Replace the normal LRU line
YES
Replace the LRU WP line with the new line
For exclusive L1-L2 Write-back the replaced WP block into L2 without changing the MRU information, i.e., L2 WP block is LRU

Fig. 16. Wrong-path aware replacement policy.

if they were not replaced). For a WP filter mechanism, only *fft* (for directory-based SMP) is negatively affected (3% performance degradation for directory-based SMP). This performance degradation is due to the decision that if a WP block is not used in L1 cache it is more likely not to be used in L2 cache. Even if the WP block is not used by a later CP block while it resides in L1 cache, it might have been used later when it resides in L2 cache. A filter policy, which is based on this general observation, therefore, may not work for all applications. For both mechanisms, the performance degradation is due to reducing prefetching effect of some useful WP blocks. However, performance degradation does not occur often as most of the results are positive. When we combine filter and the WP aware replacement, their advantages may cancel out each other in some cases (*radix* and *ocean* in broadcast-based SMPs).

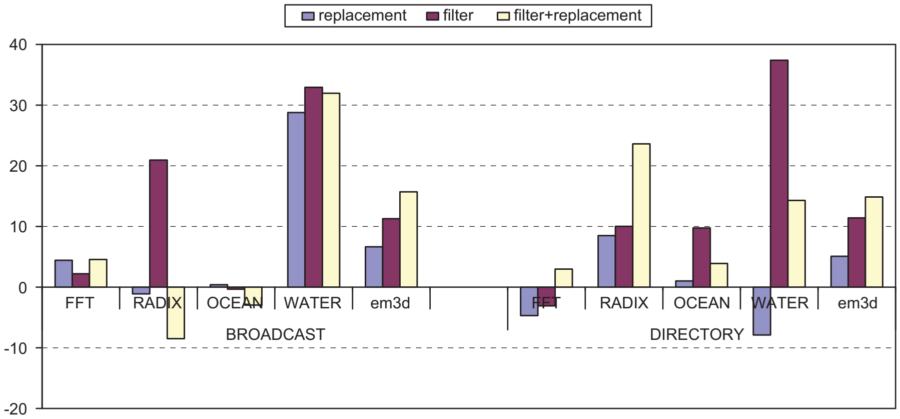

Fig. 17. Percentage speedup in execution time for wrong-path aware replacement, L2 wrong-path filter, and for combination of both, *i.e.*, filter + replacement.

6. Related work

To best of our knowledge, no previous work examined the effects that WP memory references have on SMPs. However, several papers examined the effect that speculative execution had on the performance of uniprocessor systems. Mutlu *et al.* [22,23] analyzed the performance impact that WP references have for different memory latencies and instruction window sizes. Their results showed that the major reason for performance degradation due to WP memory references is L2 cache pollution.

Sendag *et al.* [30] proposed using the fully-associative *wrong-path cache* (WPC) to eliminate the cache pollution caused by WP references. The WPC stores data brought into the processor by WP load instructions and evicted from the L1 cache. The processor accesses the WPC and the L1 data cache in parallel. Hence, the WPC functions both as a victim cache [13] and a buffer to store data fetched by WP references. This approach eliminates the pollution caused by WP references in the L1 cache. Sendag *et al.* [29] also studied the effects of incorrect speculation on the performance of a concurrent multithreaded architecture. They analyzed how wrongly-forked threads affected the memory system performance in addition to the known effects by the WP load instructions in a uniprocessor.

Mutlu *et al.* proposed using the L1 caches as filters to reduce the pollution in the L2 cache caused by speculative memory references, including both WP and prefetched references [21]. Their mechanism takes advantage of the observation that pollution in the L1 cache caused by speculative references has less impact on the performance than pollution in the L2 cache. Their approach reduces the L2 cache pollution due to speculative references for both for out-of-order and runahead processors, without requiring extra storage to hold the data fetched by the speculative references.

Finally, Pierce and Mudge studied the effect of WP memory references on cache performance [24]. Their study used trace-driven simulation, where they injected a fixed number of instructions to emulate the WP. However, this is not very realistic because the number of instructions executed on the WP is not fixed in a real processor [6]. They also introduced an instruction cache prefetching mechanism, which shows the usefulness of WP memory references to the instruction cache [25]. Their mechanism fetches both the fall-through and target addresses of conditional branch instructions.

7. Conclusion

In this paper, we evaluate the effects of executing WP memory references on the memory behavior of cache coherent multiprocessor systems. Our evaluation reveals the following key conclusions:

1. It is important to model WP memory references in cache coherent SMPs. Neglecting to model them may result in incorrect design decisions, especially for future systems with longer memory interconnect latencies and processors with larger instruction windows.
2. For SMP systems, not only do the WP memory references affect the performance of the individual processors due to prefetching and pollution, they also affect the performance of the entire system by increasing the number of cache coherence transactions, the number of cache line state transitions, the number of writebacks and invalidations due to WP coherence transactions, and the amount of resource contention (buffer usage, bandwidth, *etc.*).
3. For a workload with many cache-to-cache transfers, WP memory references can significantly affect the coherence actions.
4. Finally, simple mechanisms such as filtering unlikely to-be-used WP blocks from being placed into L2 or making the replacement policy WP aware can significantly improve the SMP performance.

Acknowledgments

This research is supported in part by US National Science Foundation grant CCF-0541162. We would like to thank Babak Falsafi and Thomas Wenisch for supplying us with the *em3d* benchmark. A preliminary version of this work was presented

at the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006) [31].

References

[1] R. Bahar, G. Albera, Performance analysis of wrong-path data cache accesses, Workshop on Performance Analysis and its Impact on Design, 1998.

[2] R. Bhargava, L. John, F. Matus, Accurately modeling speculative instruction fetching in trace-driven simulation, IEEE Performance, Computers and Communications Conference, 1999.

[4] Y. Chen, R. Sendag, D. Lilja, Using incorrect speculation to prefetch data in a concurrent multithreaded processor, International Parallel and Distributed Processing Symposium, 2003.

[6] J. Combs, C. Combs, J. Shen, Mispredicted path cache effects, Euro-Par, 1999.

[7] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, Parallel programming in Split-C, Supercomputing, 1993.

[8] D. Culler, J. Singh, Parallel Computer Architecture, Morgan Kaufmann, Los Altos, CA, 1999.

[9] J. Dundas, T. Mudge, Improving data cache performance by pre-executing instructions under a cache miss, International Conference on Supercomputing, 1997.

[10] M. Ekman, F. Dahlgren, P. Stenström, TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors, International Symposium on Low-Power Electronics and Design, 2002.

[11] M. Ekman, F. Dahlgren, P. Stenström, Evaluation of snoop-energy reduction techniques for chip-multiprocessors, Workshop on Duplicating, Deconstructing, and Debunking, 2002.

[13] N. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, International Symposium on Computer Architecture, 1990.

[15] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, IEEE Comput. 35 (2) (2002) 50–58.

[16] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, D. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, Comput. Architecture News 33 (4) (2005).

[17] C. Mauer, M. Hill, D. Wood, Full-system timing-first simulation, Joint International Conference on Measurement and Modeling of Computer Systems, 2002.

[18] A. Moshovos, RegionScout: exploiting coarse grain sharing in snoop-based coherence, International Symposium on Computer Architecture, 2005.

[19] A. Moshovos, G. Memik, B. Falsafi, A. Choudhary, JETTY: filtering snoops for reduced energy consumption in SMP servers, International Symposium on High-Performance Computer Architecture, 2001.

[20] M. Moudgill, J. Wellman, J. Moreno, An approach for quantifying the impact of not simulating mispredicted paths, Performance Analysis and its Impact in Design, 1998.

[21] O. Mutlu, H. Kim, D. Armstrong, Y. Patt, Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance, Symposium on Computer Architecture and High-Performance Computing, 2004.

[22] O. Mutlu, H. Kim, D. Armstrong, Y. Patt, Understanding the effects of wrong-path memory references on processor performance, Workshop on Memory Performance Issues, 2004.

[23] O. Mutlu, J. Stark, C. Wilkerson, Y. Patt, Runahead execution: an alternative to very large instruction windows for out-of-order processors, International Symposium on High-Performance Computer Architecture, 2003.

[24] J. Pierce, T. Mudge, The effect of speculative execution on cache performance, International Parallel Processing Symposium, 1994.

[25] J. Pierce, T. Mudge, Wrong-path instruction prefetching, International Symposium on Microarchitecture, 1996.

[27] C. Saldanha, M. Lipasti, Power efficient cache coherence, Workshop on Memory Performance Issues, 2001.

[28] R. Sendag, Y. Chen, D. Lilja, The effect of executing mispredicted load instructions on speculative multithreaded architecture, Workshop on Multi-threaded Execution, Architecture and Compilation, 2002.

[29] R. Sendag, Y. Chen, D. Lilja, The impact of incorrectly speculated memory operations in a multithreaded architecture, IEEE Trans. Parallel Distributed Systems 16 (3) (2005) 271–285.

[30] R. Sendag, D. Lilja, S. Kunkel, Exploiting the prefetching effect provided by executing mispredicted load instructions, Euro-Par, 2002.

[31] R. Sendag, A. Yilmazer, J.J. Yi, A.K. Uht, Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems, International Parallel and Distributed Processing Symposium, April 2006.

[34] S. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, International Symposium on Computer Architecture, 1995.

Resit Sendag received the Ph.D. degree in Electrical Engineering, from the University of Minnesota in Minneapolis, an M.S. in Electrical Engineering from Cukurova University in Adana, Turkey and a B.S. in Electronics Engineering from Hacettepe University in Ankara, Turkey. He is currently an Assistant Professor of Electrical and Computer Engineering at the University of Rhode Island in Kingston. His research interests include high-performance computer architecture, memory systems performance issues, and parallel computing. He is a Member of the IEEE and the IEEE Computer Society.

Ayse Yilmazer received a B.S. degree in Computer Science and Engineering from Hacettepe University in Ankara, Turkey. Currently, she is an M.S. candidate in Electrical and Computer Engineering at University of Rhode Island. Her main research interests include multiprocessor and chip multiprocessor systems, memory systems performance analysis, and parallel computing.

Joshua J. Yi received the Ph.D., M.S., and B.S. degrees, all in Electrical Engineering, from the University of Minnesota in Minneapolis. He is currently a performance analyst at Freescale Semiconductor, Inc. in Austin, Texas. His research interests include high-performance computer architecture, simulation, benchmarking, low power design, and reliable computing. He is the Organizer of the Workshops on Modeling, Benchmarking, and Simulation (MoBS) and Computer Architecture Research Directions (CARD). He is a Member of the IEEE and the IEEE Computer Society.

Augustus K. Uht is a Professor-in-Residence of the College of Engineering at the University of Rhode Island, and is primarily interested in computer microarchitecture. Dr. Uht received Bachelor's (1977) and Masters (1978) degrees from Cornell University, and a Ph.D. (1985) from Carnegie-Mellon University, all in Electrical and Computer Engineering. He was on the UCSD Faculty, and was an Engineer at both IBM and Cornell University. Dr. Uht has served as Journal Associate Editor, Workshop Chair, and Program Committee Member. He holds several US patents with other patents pending. Dr. Uht is a licensed Professional Engineer, and is a Senior Member of the IEEE.