Speculative Multiprocessor Cache Line Actions Using Instruction and Line History[1]

David M. Koppelman[2]
Department of Electrical & Computer Engineering
Louisiana State University, Baton Rouge
102 EE Building, Baton Rouge, LA, 70803 U.S.A.

Abstract: A technique is described for reducing miss latency in coherent-cache shared-memory parallel computers. Miss latency is reduced by speculatively invalidating and updating (copying an exclusively held line back to memory) cache lines at one processor that might be needed at another processor. A line becomes a candidate for speculative invalidation when another line last accessed by the same instruction is invalidated. A line becomes a candidate for speculative updating under corresponding conditions. The technique can be implemented by constructing linked lists of lines for each memory access instruction. The amount of memory needed by an implementation is less than 50% the size of the cache, roughly comparable to the cost of adding error-correcting memory. No time need be added to cache hits. In execution-driven simulations of such systems running programs from the Splash 2 suite invalidations and updates are reduced by 50% or more. Total time spent servicing misses is reduced by about 20% on 16-processor systems, execution time is reduced as much as 20% for some benchmarks on high latency systems.

Keywords: Multiprocessor, Coherent Caches, Cache Management, Instruction History, Speculative Invalidation

1 Introduction

The time needed to complete shared-memory operations is a significant factor in the performance of coherent-cache shared-memory parallel computers. The messages needed to maintain cache coherence can take hundreds of cycles or more to reach distant processors. Even on the fastest systems in which messages reach their destinations in tens of cycles memory operations take much longer than the several cycles typically needed when the cache is hit [3,13,16].

On systems using directory-based cache-coherence [3,13] an operation that misses the cache must wait for two or four message transit times. (The *number* of messages may be higher.) For example, a read miss to a memory location that is not exclusively held may be satisfied in two message delays: the message sent to the *home memory* and a return message containing the data. Misses can result in four message delays in the case of a write to a location which is cached elsewhere and in the case of a read miss to a location which is exclusively held (by another cache). In the former case the copies held elsewhere are invalidated, in the latter a request is made for the newly written data. (See [3,13] for details.) Cache misses that must wait four message delays will be called *second-cache misses*.

A method is introduced here which reduces the number of second-cache misses by speculatively invalidating or updating (restoring to the shared state) cache lines. Speculative invalidation reduces the number of write misses in which the data is cached elsewhere. Speculative updating reduces the number of read misses in which the data is exclusively held. As a result of these actions miss latency is reduced since fewer messages, on average, are needed.

The choice of lines to speculatively invalidate or update is made assuming that those lines last accessed by an instruction may share the same fate. That is, if one line last-accessed by a particular instruction is invalidated, then other lines last-accessed by the instruction will likely be invalidated. The same idea is used for updating.

This idea is implemented by constructing, for each active memory access instruction, a linked list of accessed lines. A line is added to the list when the access instruction operates on it. A line can belong to only one list at a time, so a subsequent access by another instruction will remove the line from the first list and place it on the new instruction's list.

The updating of the pointers needed to implement the linked list, as well as the speculative actions, need not be completed during a cache access. Therefore, maintaining the lists adds little or nothing to the cache-hit latency. The speculative actions do add to the volume of communication; added volume is only a problem if it slows down other traffic. Execution-driven simulations show that the additional volume only adds a very small amount to the time needed for normal network traffic, while the overall time for shared-memory access is reduced.

Other methods of reducing shared-memory access latency have been reported. Prefetching is in some ways similar to the technique described here. Software prefetch schemes use *prefetch instructions*, which bring data to a cache but otherwise have no effect, inserted by the programmer or compiler. (See [9,15] for prefetching on serial systems and [8,12] for parallel systems.) Software prefetching works well when the data needed can be identified well enough in advance. The scheme described here does not depend upon such identification by a compiler or programmer; no changes at all need be made to object code.

Prefetch instructions are not needed in hardware prefetching schemes. A hardware prefetching scheme is described by Dahlgren, Dubois (in the first reference), and Stenström [6,7] for parallel systems and by Chen and Baer [4] for serial systems. Blocks to be prefetched are identified by guessing the stride of memory accesses. Once determined, prefetch can occur as far in advance as needed. The effectiveness of such schemes depends upon regular access to memory and so may only work for certain programs.

Other schemes attempt to determine the behavior of accesses to a location [1,5,11,17]. In an *adaptive caching* scheme described by Bennett, Carter, and Zwaenepoel, data sharing behavior is divided into classes. In a system based on this idea, the history of memory accesses to a location would be used to determine its class. A coherence mechanism appropriate to the class would then be chosen for the location. Trace driven simulations show that classes can be detected and that performance gains are possible.

Cox and Fowler [5] and in a similar paper Stenström, Brorsson, and Sandberg [17], observe that a pattern of read/invalidate/miss observed at a line might indicate that the data is *migratory*, exclusively read and written over a span of time by one processor at a time. A read by a processor to such a line might fetch an exclusive copy (rather than a shared copy), anticipating a write

[1] To appear in the proceedings of *The 10th International Conference on Parallel & Distributed Computing Systems*, New Orleans, Louisiana, October 1997.
[2] This work is supported in part by the Louisiana Board of Regents through the Louisiana Education Quality Support Fund, contract number LEQSF (1993-95)-RD-A-07 and by the National Science Foundation under Grant No. MIP-9410435.

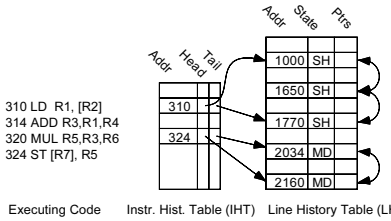

Figure 1. Relationship between code, IHT, and LHT: IHT has an entry for the two memory access instructions in illustrated code. Each entry points to linked list of lines last-accessed by the instruction, stored in LHT.

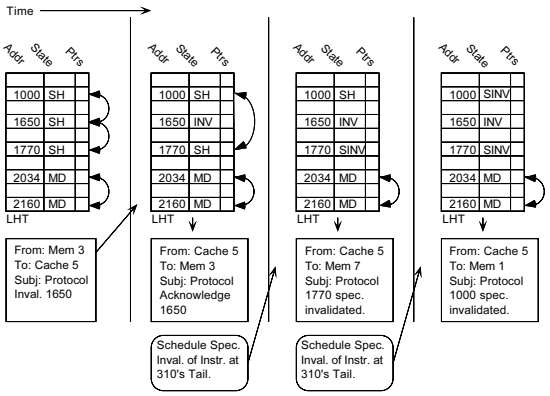

Figure 2. Example of speculative invalidation: 1) Normal invalidation message received from memory. 2) Line invalidated and speculative invalidation initiated. 3) Tail of list speculatively invalidated. 4) New tail of list speculatively invalidated, list is empty, ending these invalidations.

by the processor. Lebeck and Wood describe a scheme in which the cache directory (located at the memory module) observes a block's behavior. Based on this behavior it might speculatively invalidate the block in certain caches [11].

One shortcoming with these methods is that the behavior is associated with a line—one line. The behavior at a line must first be detected before any performance benefit is realized in accesses to the line. In the scheme described here behavior is associated with an instruction; once the behavior is detected the performance benefit is realized in all lines accessed by the instruction. A big improvement since the ratio of number of lines to number of instructions can be very large.

Lilja describes a scheme in which the policy chosen for a write instruction, write invalidate or write update, is based upon its execution history, using software assistance [14]. It is similar to the method described here in that the fate of a line is tied to the instruction that last accessed it. It is however less flexible, since only instructions that were the last to write a line could initiate an update (write back). In the method described here, a line is speculatively updated only if another line last-accessed by the same instruction is updated. This could happen long after the write executed or after some lines accessed by the write were subsequently written by other instructions.

2 Speculative Invalidation Hardware

2.1 Hardware

Speculative actions are added to multiprocessor, cached shared memory parallel systems; for some background see [3,13]. Speculative invalidation and updating can be implemented by adding two tables and some control hardware to each cache and by making minor modifications to the memory controllers. The cache has two tables added, an *instruction history table* (IHT) and a *line history table* (LHT). The IHT stores the head and tail of active instructions' linked lists, as well as other data. The LHT stores the pointers needed to implement the linked list. A controller is used to maintain the lists and to perform the speculative actions.

Each cache line has an associated LHT entry. (The LHT entry could be located on the same physical devices used for the line, however there may be cost and performance benefits when they are kept separate.) Figure 1 illustrates the relationship between the tables and executing code. The code fragment in the left part of the figure contains two memory access instructions; entries for the two are in the IHT in the center of the figure. The LHT in the right of the figure holds the linked list. An IHT entry may also contain information used to determine if speculative action is warranted.

Whenever a memory access instruction hits the cache, the corresponding line's LHT entry is read. The previous and next pointers are used to remove the line from the list it was in when accessed (if any). The memory accesses instruction's IHT index is written to the line's LHT entry. The IHT entry for the memory access instruction and the line pointed to by the IHT entry's head pointer are updated so that the accessed line is the new list head.

Speculative actions are triggered by invalidation and update messages (sent from the memory to the cache). When an invalidation message is received, the target line is invalidated as usual. In addition, a process called *list traversal* is possibly started. First, the line's LHT entry is read and the identity of the last-accessing instruction is determined. The IHT entry for that instruction is read. Based on information in the IHT entry a decision is made on whether to proceed with list traversal. If positive, the tail pointer is read, and the line to which it points is speculatively invalidated or updated.

A line is speculatively invalidated by sending a *speculative invalidation message* to the home memory (containing data if the line were exclusively held). The IHT's tail pointer is updated, and the next speculative action is scheduled. An example of speculative invalidation is illustrated in Figure 2. The procedure for speculative updates is similar, except lines are not removed from the list and, of course, speculative update messages are sent to memory.

The data in a line that had been speculatively invalidated from the cache is treated no differently than if the memory had sent an invalidation message: a subsequent access to that location will miss the cache. A speculatively updated line is no different than one that had been copied back: to complete a write an exclusive copy would have to be re-obtained.

To avoid network congestion and memory hotspots the rate of list traversal should be controlled. In the simulations, a rate was determined statically using network topology, link widths, and other data. For higher performance, the rate of list traversal could be based on the current state of the network.

2.2 Performance Monitoring Hardware

The effectiveness of speculative invalidation would be reduced if a processor frequently accessed a block that it had speculatively invalidated but that had not been subsequently written by another processor. Such events will be called *false positives*. False positives also occur when there are speculative updates to

lines which are later written at the same processor, before being read elsewhere.

To reduce the number of false positives a record is kept for each memory access instruction of the effectiveness of the speculative actions using that instruction's linked list. List traversal would not be initiated for instructions generating too many false positives.

The damage from false positives is also reduced by limiting the number of lines acted upon during list traversal, avoiding the significant performance degradation that would occur on a long list of false positives. For the experiments described below, the limit was twenty for invalidation, but there was no limit for updating. List traversal resumes if the lines remain unaccessed at the cache and an invalidation arrives for a remaining list member.

2.3 Hardware Cost

Most of the cost of the speculative hardware is in the IHT and LHT and the controller needed at the caches. The cost will be estimated by finding the amount of additional storage needed.

Let n_{IHT} denote the number of entries in the IHT. Let n_{s}, n_{c}, and n_{l} denote the number of sets, the number of lines in the cache, and the size of the lines at each processor, respectively. Each LHT entry contains pointers to two other LHT entries. Since there is one LHT entry for each cache line, the pointers need to be $\lceil \log_2 n_{\mathrm{c}} \rceil$ bits. (If n_{c} is an integral power of two then the pointers must be $1 + \log_2 n_{\mathrm{c}}$ bits, to code a null value.) The size of the last-accessor field must be $\lceil \log_2 n_{\mathrm{IHT}} \rceil$ bits.

The line itself must store the data as well as a tag. The size of the line for a byte-addressable system is given by $8n_{\mathrm{l}} + A - \lceil \log_2 n_{\mathrm{s}} \rceil - \lceil \log_2 n_{\mathrm{l}} \rceil$ bits, where A is the number of bits in an address. The ratio of the amount of storage for the LHT to the amount of storage for the line is given by

$$\frac{2\lceil \log_2 n_{\mathrm{c}} \rceil + \lceil \log_2 n_{\mathrm{IHT}} \rceil}{8n_{\mathrm{l}} + A - \lceil \log_2 n_{\mathrm{s}} \rceil - \lceil \log_2 n_{\mathrm{l}} \rceil}.$$

For a system in which $n_{\mathrm{IHT}} = 1000$ entries, $n_{\mathrm{c}} = 2^{16}$ bytes, $n_{\mathrm{l}} = 16$ bytes, $n_{\mathrm{s}} = 2^{13}$ sets, and $A = 32$ bits, the ratio is .29. Such a cache holds one megabyte. With smaller caches and larger line sizes the fraction of storage used by by the LHT is lower.

An IHT entry contains a tag identifying the instruction address, head and tail pointers, state, and performance monitoring fields. The tag size is no larger than A; the head and tail pointers are each $\lceil \log_2 n_{\mathrm{c}} \rceil$ bits. Allowing 32 bits for state and performance monitoring data, and using the cache parameters above, an IHT entry would be 12 bytes. Since there are many fewer IHT entries than cache lines the IHT storage would only be a small fraction of total storage.

The cost of the hardware at the memory modules is insignificant since only a single bit per entry is needed to store a *speculatively invalidated* state. Based on the estimated IHT and LHT storage requirements, adding speculative actions increases cache storage requirements by less than 50%. The controller needed would be no more complicated than the controller used to maintain cache coherence; some or all of this hardware might be shared.

3 Methodology

The effectiveness of speculative actions was tested using execution-driven simulation. Shared-memory multiprocessor systems using a write-invalidate, limited-directory cache coherence protocol were simulated. The experiments, using programs from the Splash 2 shared-memory benchmark suite [18], were designed to determine values of certain parameters (for example, time between speculative invalidations), to determine sensitivity to system configuration variations, and to determine the performance difference between a normal system and one using speculative invalidation and updating.

3.1 Proteus

The simulations were performed using a modified version of the Proteus simulator [2]. Modifications were made to simulate the speculative scheme described here, other modifications were made that are unrelated to the speculative hardware; that is, they impact the reported performance of systems that do not use the speculative hardware. For details on the changes see [10].

Proteus is an execution-driven parallel computer simulator which simulates a network, memory system, and processors running a parallel program. The modified version of Proteus runs on Sparc systems, and was run on Solaris 2.5.1.

The simulated system runs parallel programs written in C and which include some parallel programming functions and operations, including shared memory operations. The C programs are pre-processed and compiled using a host-system compiler, in this case gcc 2.7.2.1. The compiled code (in assembly form) is augmented at least every basic block with cycle-counting and shared-memory access code. The simulated system runs user (the benchmarks), library, and some OS code. The OS code includes a TLB miss handler and other virtual memory management procedures, so VM management timing is accurate.

The augmentation process inserts code for cycle counting, simulator context switches, and shared memory accesses. The cycle counting code keeps track of time on the simulated system and initiates context switches. Time is advanced at the rate of about one cycle for every two instructions except for load, store, and certain floating-point instructions. Load and store instructions that might access shared memory are replaced with code that tests the address and calls simulator procedures if shared memory is indeed accessed. Cache hit latency is approximately one cycle, the time to complete an access that misses is determined by network interface, network, memory, and cache latencies, and the protocol actions needed to complete the accesses. Further information can be found in [10].

The interconnection network is simulated at the packet-transfer level; k-ary n-cubes (n-dimensional meshes) are used for the work reported here. Network nodes effectively consist of a single shared infinite buffer.

The simulated system provides virtual memory, using 2^{12}-byte pages and 64-entry, fully associative TLBs. Caches are physically mapped, color matching is used for physical page assignment. Memory allocation routines can return a single contiguous block distributed over all memory modules. Stores are nonblocking but complete in program order with respect to other stores; up to five stores per processor can be simultaneously active. Loads do not complete in order with respect to stores, but of course can read values to be written by pending stores, maintaining thread-specified data dependence. The simulated memory system uses a full-map directory similar to the one described in [3], for differences see [10]. Each processor has an associated memory module, sharing the network interface queue with messages bound for the processor.

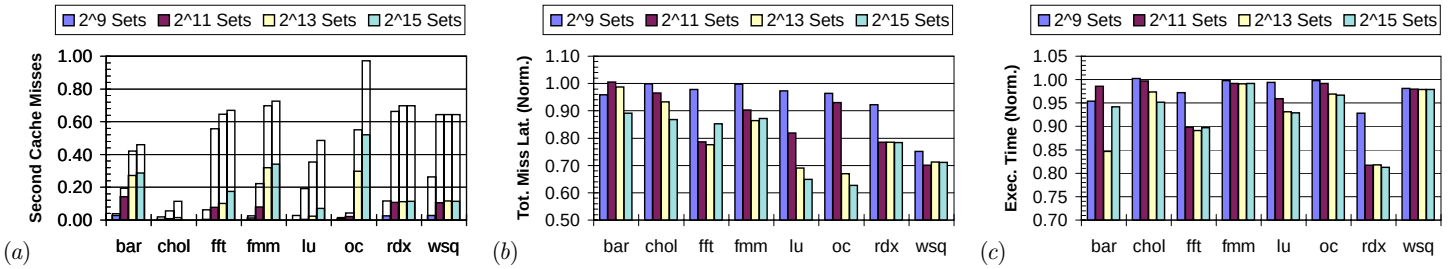

Figure 4. Effect of cache size on (*a*) fraction of second cache misses with (shaded) and without (outline) speculative actions, (*b*) total miss latency, and (*c*) execution time. Bars normalized to conventional system.

Table I: Basic Configuration Parameters

Simulation Parameter	Value
System Size	16 processors
Network Topology	$4^2 = 4 \times 4$ mesh
VM Page Size	2^{12} bytes
TLB Capacity	64 entries
TLB Replacement	LRU, fully assoc.
Cache Size	2^{13} sets
Cache Associativity	8
Cache Line Size	16 bytes
Cache Capacity	1,048,576 bytes
Cache Hit Latency	1 cycle
Directory Size	full map
Completion Buffer	5 stores
Memory Latency	10 cycles
Protocol Message Size	$n_{\text{pr}} = 6$ bytes (plus data)
Network Interface Width	8 bytes
Network Link Width	4 bytes
Hop Latency	20 cycles (plus waiting)

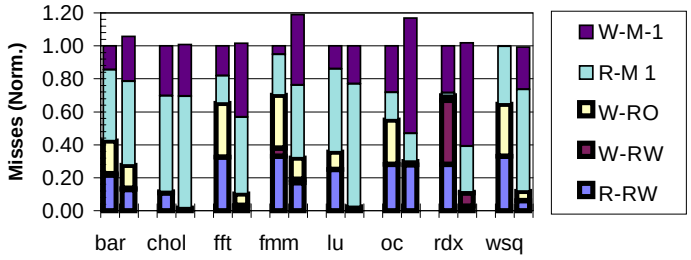

Figure 3. Normalized number of misses with (right member of pair) and without (left member of pair) speculative invalidation. Bold indicates second-cache misses, segments indicate type of miss (see text).

3.2 Splash 2 Suite

The Splash 2 suite consists of a representative sample of scientific shared-memory parallel programs for use in testing shared memory systems [18]. Three Splash 2 kernel programs ran for the results reported here are Cholesky, FFT, and LU. The fourth kernel, Radix, was used in modified form. (The modifications improved the efficiency of the prefix sum used in the kernel.) Radix is an integer sorting program, Cholesky factors matrices, FFT performs a 1-dimensional fast-Fourier transform using a "radix-$\sqrt{N}$, six-step" algorithm, and LU is a dense-matrix LU factorization program. Four Splash 2 applications were also run, Barnes, FMM, Ocean (contiguous partitions), and Water N^2. Barnes simulates particle interactions in three dimensions using the Barnes-Hut method and FMM simulates particle interactions in two dimensions using the Adaptive Fast Multipole Method; both use tree-based data structures, though of different types. Water N^2 simulates forces on water molecules and Ocean simulates ocean currents [18]. The programs were run using the base problem sizes specified in the distribution. The programs' comments specify where statistics gathering might start and stop; the statistics described below are collected in those intervals.

3.3 Configurations

The experiments tested several different parallel-computer configurations. Variations were made in network characteristics, memory timing, and cache structure. The table gives the simulation parameters describing the *basic configuration*. The differences from the basic configuration will be noted for each experiment. Some parameters are explained below; see [10] for a detailed explanation of the network- and memory-related parameters.

3.4 Simulated Speculative Hardware

In the implementation modeled by the experiments, speculative invalidation occurs at two rates: fast, for short lists, and slow, for long lists. After a speculative invalidation occurs the list length is checked; if it is less than n_{slow} then the next action (if any) is scheduled for $t_{\text{a-fast}}$ cycles later. Otherwise, it is scheduled for $t_{\text{a-slow}}$ cycles later. The fast rate is based on the maximum amount of traffic a single link could carry (multiplied by a tuning coefficient) when one cache is in the process of list traversal and there are no other message sources. The slow rate is based on the maximum amount of traffic a single link could carry (multiplied by a tuning coefficient) when all caches are in list traversal and there are no other message sources.

4 Experiments

4.1 Basic Configuration

The effectiveness of speculative actions at reducing the number of second-cache misses is illustrated in Figure 3 for the basic configuration. There is a pair of bars, indicating misses, for each benchmark; the bar on the left is for the conventional system; the bar on the right is for systems using speculative actions. The size of the bar segments indicate the number of each type of miss. Second cache misses, outlined in bold, are labeled W-RO for a write to a block which is shared in another cache; W-RW for a write to a block which is exclusive in another cache; and R-RW for a read to a block which is exclusive in another cache. Other read and write misses are denoted R-M-1 and W-M-1, respectively. The segment sizes are scaled so that the number of misses in the conventional system is one. As can be seen, the number of second-cache misses is reduced substantially, sometimes to less than 10% its original value. There is a diversity of miss behavior and reaction to speculative actions. While the number of second-cache misses is reduced, the total number of misses remains the same or increases only slightly, as can be seen in Figure 3.

The impact on total miss time (the sum of all miss latencies)

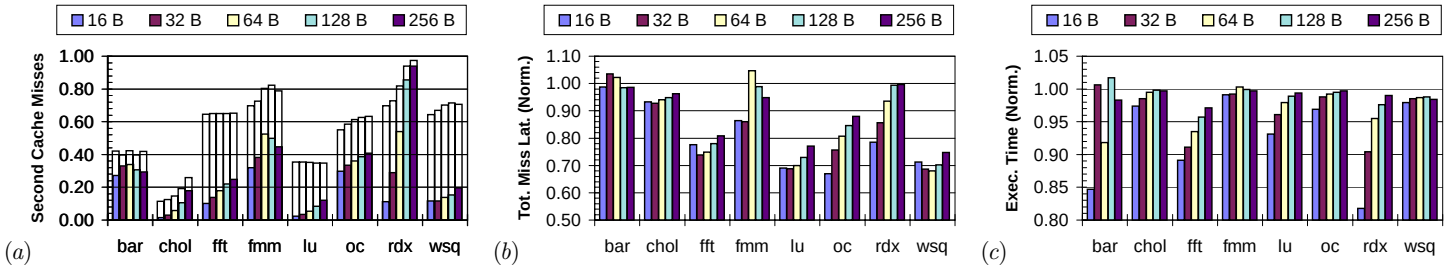

Figure 5. Effect of line size on (*a*) fraction of second cache misses with (shaded) and without (outline) speculative actions, (*b*) total miss latency, and (*c*) execution time. Bars normalized to conventional system.

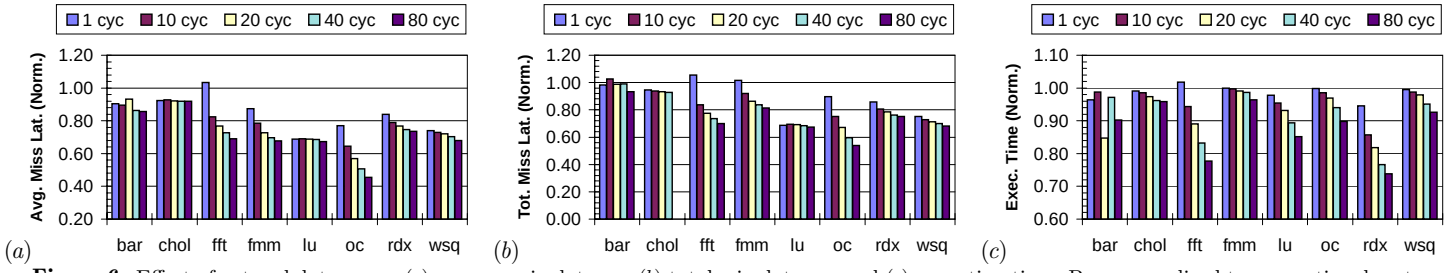

Figure 6. Effect of network latency on (*a*) average miss latency, (*b*) total miss latency, and (*c*) execution time. Bars normalized to conventional system.

can be seen in Figure 4(*b*); (the bars for 2^{13} sets are for the basic configuration). Impact on miss time varies with how many second cache misses there were and with additional congestion caused by speculative actions. Miss time is reduced by 25% or more in four benchmarks but less than 10% in two. The impact on execution time can be seen in Figure 4(*c*). Some applications, such as FMM and Water N^2 show negligible improvement, others show 10% or more improvement. The small improvements are due to an increase in miss rate or a small number of second-cache misses.

4.2 System-Configuration Effects

System configuration can determine the effectiveness of speculative actions. In systems using smaller caches lines are more likely to be evicted, so there will be fewer lines to speculatively invalidate. There is a greater chance of unrelated data sharing a line in systems with larger line sizes, possibly confounding speculation. Also with larger lines, an individual action uses more network and memory bandwidth. In systems having higher network latency more run time is spent waiting for accesses to complete, so reductions in miss latency have a greater impact on performance. Speculative actions can hurt performance on systems with bandwidth constraints in the network or memory system.

To test the effect of cache size, experiments were performed in which the number of sets in the cache was varied from 2^9 to 2^{15}. Figure 4(*a*) shows the number of second-cache misses with (shaded) and without (outline) speculative actions, scaled to the number of misses without speculative actions. Second-cache misses are only a small fraction of total misses when caches are small. With only a few second-cache misses to eliminate, speculative actions have almost no effect, this can be seen in Figure 4(*b*) and (*c*). As cache sizes increases the fraction of second cache misses becomes significant, the effect of increasing the cache size beyond a certain point is small. For most of the benchmarks, the change in absolute run time between 2^{15} and 2^{17} sets was small. The number of misses encountered by Water N^2 changed little above 2^9 sets. These results show that speculative actions are not effective when the cache is small and that beyond a certain cache size, adding speculative actions will have a greater performance impact than a further increase in cache size.

The effect of line size on speculative actions is shown in Figure 5. In systems using longer lines speculative actions are less effective. The impact on absolute execution time varies with benchmark. Kernels LU, FFT, and Cholesky run well with long lines (due to contiguous data) while Radix suffers greatly (due to small randomly ordered data). The others also suffer to some extent at longer line sizes.

The results of varying network latency are plotted in Figure 6, latency is given in CPU cycles per hop. (Since CPU clocks run faster higher than external components the numbers may appear high.) As expected, speculative actions perform better on systems with higher latency. The effect is large, FFT goes from a slowdown with a 1-cycle latency to a 20% drop in execution time at 80 cycles.

The additional traffic due to speculative actions can congest a system with limited bandwidth. To test this memory bandwidth, given in cycles per access, was varied, the results are in Figure 7. On systems with the fastest memory access latency was a smaller fraction of execution time and so speculative actions had less of an effect, on systems with the slowest memory speculative actions congested the system. The best speedups were attained at middle values. On real systems memory might be banked so that latency would be high but so would bandwidth.

5 Conclusions

A method of speculative invalidation and updating of cache lines was described. Using these speculative actions, the number of second-cache misses is reduced significantly. These speculative actions have only a minor impact on miss rate. The average time needed to service cache misses is reduced by over 20% in some cases. The cost of adding this hardware is approximately the same as the cost of increasing cache size by 50%. (In systems having an ample amount of cache, increasing cache size will have little or no effect on performance, so that increasing cache size by 50% would be more cost-effective in systems with undersized caches while adding speculative actions would be the better route

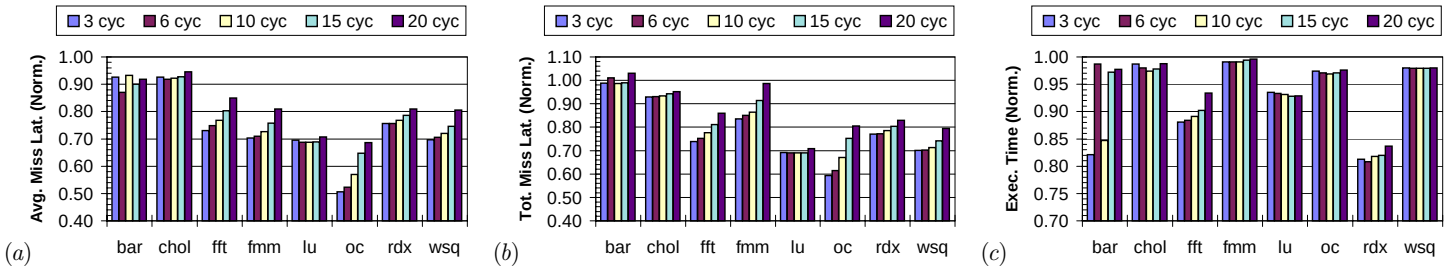

Figure 7. Effect of non-pipelined memory latency on (*a*) average miss latency, (*b*) total miss latency, and (*c*) execution time. Bars normalized to conventional system.

to improved performance when caches are large.) To put the cost in perspective, the added cost is roughly equivalent to the cost of adding error-correcting-code memory.

The technique works best in systems with large caches and having high network latency. Performance is low on systems using small caches because lines are frequently evicted in such systems; there is less benefit in speculatively invalidating a line that will likely be evicted. In contrast, on systems using larger cache sizes, speculative actions would not be undermined by eviction. Performance is also lower on systems in which network latency is low because the penalty for a cache miss is smaller, and so avoiding second-cache misses is less worthwhile.

The technique could be used in place of, or in combination with, prefetching and adaptive cache coherence schemes. The information collected in the IHT and LHT might be useful for other purposes, such as an improved line-replacement algorithm.

If future systems adopt simultaneous multithreading or other schemes in which high cache miss rates can be tolerated then speculative actions will be less useful. On the other hand, there is little doubt that in future systems communication latency with respect to clock speed will be higher, favoring speculative actions.

6 References

[1] John K. Bennett, John B. Carter, and Willy Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures," *ACM Computer Arch. News*, vol. 18, no. 2, pp. 125–134, May 1990.

[2] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, "Proteus: a high-performance parallel-architecture simulator," in *Proc. of the ACM SIGMETRICS conference*, May 1992.

[3] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal, "Directory based cache coherence in large–scale multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 49–59, June 1990.

[4] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.

[5] Cox, A.L., and Fowler, R.J. Adaptive cache coherency for detecting migratory shared data.*Proc. of the Intl. Symp. on Computer Arch.* May 1993, pp. 98–108.

[6] Fredrik Dahlgren, Michel Dubois, and Per Stenstroem, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733-746, July 1995.

[7] Fredrik Dahlgren and Per Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385-398.

[8] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. of the Intl. Conference on Parallel Processing*, August 1991, vol. I, pp. 355–364.

[9] Klaiber, A.C., and Levy, H.M. An architecture for software-controlled data prefetching.*Proc. of the Intl. Symp. on Computer Arch.* May 1991, pp. 43–53.

[10] D.M. Koppelman, "Ver. L3.11 Proteus Changes" Department of Electrical and Computer Engineering, Louisiana State University, (simulator documentation), http://www.ee.lsu.edu/koppel/proteus/proteusl_1.html and http://www.ee.lsu.edu/koppel/proteus.

[11] Lebeck, A.R., and Wood, D.A. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors.*Proc. of the Intl. Symp. on Computer Arch.* May 1995, pp. 48–59.

[12] Lee, R.L., Yew, P.C., and Lawrie, D.H. Data prefetching in shared memory multiprocessors.*Proc. of the Intl. Conference on Parallel Processing.* August 1987, pp. 28–31.

[13] David J. Lilja, "Cache coherence in large–scale shared–memory multiprocessors: issues and comparisons," *ACM Computing Surveys*, vol. 25, no. 3, pp. 303–338, September 1993.

[14] Lilja, D.J. Compiler assistance for directory-based cache coherence enforcement.*Proc. of the Intl. Conference on Parallel Processing.* August 1995, Workshop, pp. 133–138.

[15] Mowry, T.C., Lam, M.S., and Gupta, A. Design and evaluation of a compiler algorithm for prefetching.*Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems.* October 1992, pp. 62–73.

[16] Stenström, "A survey of cache coherence schemes for multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[17] Stenström, P., Brorsson, M., and Sandberg, L. An adaptive cache coherence protocol optimized for migratory sharing.*Proc. of the Intl. Symp. on Computer Arch.* May 1993, 20th, pp. 109–118.

[18] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A. The SPLASH-2 programs: characterization and methodological considerations.*Proc. of the Intl. Symp. on Computer Arch.* May 1995, pp. 24–36.