Speculative Multiprocessor Cache Line Actions Using Instruction History

David M. Koppelman[1]
Department of Electrical & Computer Engineering
Louisiana State University, Baton Rouge
102 EE Building, Baton Rouge, LA, 70803 U.S.A.
koppel@ee.lsu.edu

[1] This work was 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.

Address for correspondence:

David M. Koppelman
102 EE Building
Department of Electrical & Computer Engineering
Louisiana State University
Baton Rouge, LA 70803

Voice: (225) 388-5482
Fax: (225) 388-5200
E-mail: koppel@ee.lsu.edu

Abstract: A technique is described for reducing miss latency in coherent-cache shared-memory parallel computers. Miss latency is reduced by speculatively invalidating and downgrading (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 downgrading under corresponding conditions. The technique can be implemented by constructing linked lists of lines for recent memory access instructions. The amount of memory needed by an implementation is little more than 11% the size of the cache. No time need be added to cache hits. In execution-driven simulations of such systems running programs from the SPLASH 2 suite invalidations and downgrades are reduced by 50% or more.

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 multiprocessors, shared memory parallel computers. The messages between caches and memory needed to send data and otherwise implement the memory model 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 [3,18,22].

On systems using write-invalidate directory-based cache-coherence [3,18] an operation that misses the cache must wait for two or four message transit times. (The actual number of messages may be higher.) For example, a read miss to a memory location that is not cached in the exclusive state at a remote processor will be satisfied in two message delays: the message sent to the *home memory* and a return message containing the data. Misses result in four message delays when there is a write to a location that is cached elsewhere and when there is a read miss to a location that is exclusively held by another cache. In the former case the copies held elsewhere are invalidated, in the latter a *downgrade* request is sent to the cache holding an exclusive copy. (See [3,18] for details.) Cache misses that must wait four message transits will be called *second cache misses (SCMs)*.

Second cache misses are due to data sharing and so are inescapable when using conventional protocols. Because the interconnect contributes to long miss delays, caches in multiprocessors are made large (using multiple levels). With large caches the number of conflict and capacity misses is small and so second cache misses dominate performance.

This can be seen in Figure 1 where the proportion of second cache misses suffered on multiprocessors running SPLASH 2 benchmarks is shown by the bottom row of pie charts; they make up over half the misses for five out of eight benchmarks. (The data plotted is for the base system described in Section 3.) The bar graph segments show the proportion of misses of each type with wide segments showing second cache misses. The lower segments

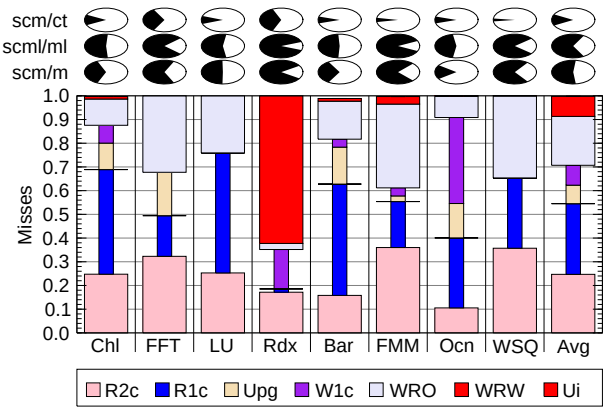

Figure 1. Categorization of misses (bar chart) and impact of second cache misses (pie charts). Wide segments are second cache misses. Plotted from the bottom up, miss types are: *R2c* read exclusive elsewhere; *R1c* read shared elsewhere or not cached; *Upg* write shared locally and not cached elsewhere; *W1c* write not cached; *WRO* write shared; *WRW* write exclusive elsewhere. Bottom row of pie charts: fraction of misses to second cache. Middle row of pie charts: fraction of miss latency due to second cache misses. Top row of pie charts: fraction of computation time stalled due to second cache misses.

show read misses and the upper segments show write misses; both types of second cache misses are common.

The middle row of pie charts shows the fraction of miss latency due to second cache misses, on average nearly 75%. Clearly second cache misses are an appropriate target for miss latency reduction. The top row of pie charts shows the fraction of computation time stalled due to second cache misses. If all second cache misses were avoided this stall time could be cut roughly in half, which would be significant for some benchmarks, such as FFT and Radix.

A method is introduced here which reduces the number of second cache misses by speculatively invalidating or downgrading (converting to the shared state) cache lines. The choice of lines to speculatively invalidate or downgrade 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 same instruction will likely be invalidated. The same idea is used for downgrading.

This assumption holds, for example, when a few instructions in a producer processor write a table which is later read by other processors. The first downgrade received by

the producer processor for an entry in the table will likely be followed by downgrades for other elements of the table as other processors read the newly written table. Later, some processor may re-write the table whose elements are now widely shared. The first invalidation received for a table element will likely be followed by other invalidations. Note that fate is correlated with the last accessing instruction so the data need not be contiguous or otherwise regular.

The SCM reduction technique, called *speculatively linked invalidations and downgrades* (SLID), 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. Speculation scores are maintained for each instruction; if a score is above a threshold lines in the list are invalidated or downgraded.

Fortunately this process need not be perfect. In the implementations simulated a line would not be moved from one list to another if the hardware maintaining the lists is busy with a prior access, and so the cost of the linking hardware is much lower that it would be if it had to keep up with memory access. The added hardware does not affect hit latency and more traffic is avoided by speculation (correct or not) than is added.

Many other schemes have been proposed for reducing delays due to data sharing, several of these are discussed in Section 5. One specifically targeting SCMs is described by Lebeck and Wood. In their *Dynamic Self-Invalidation* (DSI) cache lines are speculatively invalidated (though not downgraded) to avoid second cache misses [16]. Lines are candidates for *self-invalidation* if it appears that second cache misses accessed them; the determination is made at the memory controller by observing line state history. Speculative (self-) invalidation occurs at synchronization points. DSI is described in detail in Section 3.4.

Though both DSI and SLID reduce second cache misses they work very differently. DSI detects self-invalidation candidates on a line-by-line basis, necessitating a warm-up period and large penalties for misprediction. Because (last-accessing) instruction correlation is used SLID can speculatively act on a line that was never accessed by a second cache miss. DSI performs invalidation at all synchronization points, often invalidating too soon. SLID acts on lines only when other lines are invalidated or downgraded, reducing misspeculations. (See Section 4 for details.)

The remainder of the paper is organized as follows. Details of SLID appear in Section 2. Simulation methodology is described in Section 3; experiments and results are discussed in Section 4. Related work is discussed in Section 5 and conclusions appear in Section 6.

2 Speculatively Linked Invalidation and Downgrading

2.1 Linking

SLID is intended for multiprocessors using directory based coherence, for background see [3,18]. The added hardware organizes cache lines into linked lists, each linked list holds lines accessed by a recent memory access instruction (based on the instruction's PC). The linked lists' head and tail pointers, along with other information, is stored in a PC-indexed *instruction history table* (IHT). A *line history table* (LHT), indexed by effective address, holds the linked list pointers for each line. (The LHT may be part of the cache's tag store.) Each processor has its own LHT and IHT. If the IHT is indexed by low-order PC bits, as it is in the simulated system, then an entry can be unintentionally shared by more than one instruction. For clarity, such collisions will be ignored in this discussion but occur in the simulations.

The linked lists are maintained by linker hardware that adds, removes, or moves lines between lists by updating the appropriate LHT and IHT entries. When a line arrives at a cache it is added to the head of the list corresponding to the instruction accessing it. For

example, suppose an instruction accesses lines 1 through 5 in order (line 5 most recent) and no following instruction access them. The list will be **T** → 1 → 2 → 3 → 4 → 5 → **H** assuming the list was empty before accessing line 1. The arrows show the order in which lines will be speculatively invalidated or downgraded. When a line is evicted it is removed from its list. For example, if line 3 from the list above were evicted the result would be **T** → 1 → 2 → 4 → 5 → **H**.

When a line is invalidated it is removed from the list, breaking the list into two pieces. Under the assumption that lines will be invalidated in the same order they were last accessed, the piece following (more recently accessed than) the invalidated line is linked to the tail and the other piece is linked to the head without changing the order within each piece. For example, suppose line 3 in list **T** → 1 → 2 → 3 → 4 → 5 → **H** is invalidated. The list after removal and reordering is **T** → 4 → 5 → 1 → 2 → **H**. A similar reordering is done when a line is downgraded (including speculative downgrades) except the line is not removed and the list is split between the downgraded line and the preceding line. For example, if line 3 in the original list above is downgraded the list will be reordered to **T** → 4 → 5 → 1 → 2 → 3 → **H**. When a line is upgraded it is moved to a new list.

The list adjustments described so far are triggered by misses or upgrades and are therefore infrequent. Lists are also adjusted on a cache hit: a line is moved to a new list or the head of the current one (if it is re-accessed by the same instruction). Because hits occur frequently and list updates require several accesses it is unreasonable to expect the linker hardware to keep up with hits or for hits to wait for the linker hardware. In the simulated systems lists are adjusted on a hit only if the linker hardware is idle, otherwise the accessed line is not moved. (In the simulated systems the linker hardware takes three cycles to remove an evicted line, three cycles to add a line when it arrives at the cache, and six cycles for all other adjustments.)

Figure 2 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

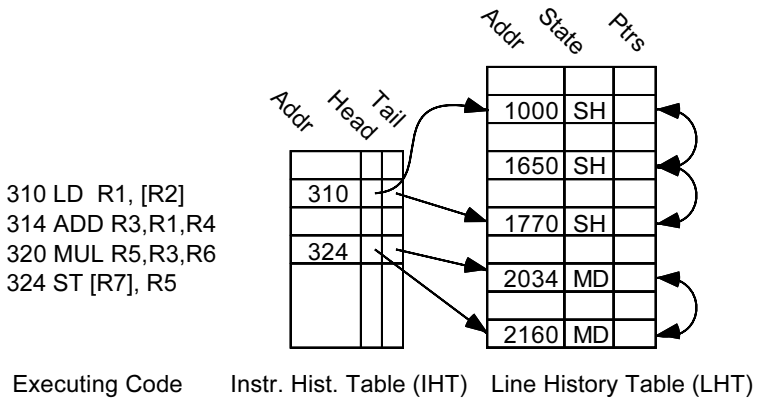

Figure 2. Relationship between code, IHT, and LHT: IHT has an entry for the two memory access instructions in the illustrated code. Each entry points to a linked list of lines last accessed by the instruction, stored in the LHT.

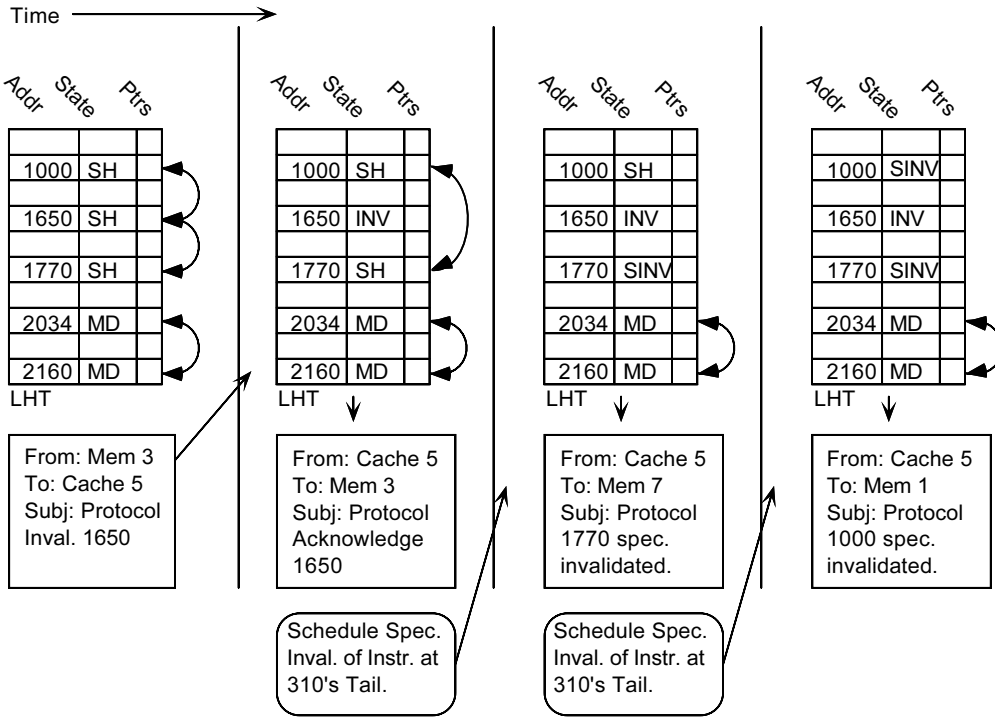

Figure 3. Example of speculative invalidation: 1) Normal invalidation message received from memory. 2) Line invalidated and speculative invalidation initiated. 3) Tail of list speculatively invalidated. 4) New tail of list speculatively invalidated, list is empty, ending these invalidations.

the two are in the IHT in the center of the figure. The LHT in the right of the figure holds the linked list.

2.2 List Traversal

Lines are speculatively invalidated and downgraded during a process called *list traver-*

sal. Scores are maintained for recent memory access instructions (see below) and kept in the IHT; list traversal for an instruction starts when its score exceeds a threshold. A list traversal step consists of speculatively invalidating or downgrading the line at the tail and scheduling the next step if the list is non-empty, the score is above the threshold, and, for downgrades, an exclusive line was found in this or in the previous speculative downgrade attempt. Note that because invalidated lines are removed from a list and downgraded lines are moved to the list head, a list can be "traversed" by always acting on the line found at the tail. In the systems simulated there is no limit to the number of lists simultaneously being traversed.

A line is speculatively invalidated by sending a *speculative invalidation message* (containing data if the line were exclusively held) to the home memory. The line is set to a speculatively invalidated state and the data in the line is treated no differently than if the memory had sent an invalidation message: a subsequent access to that location will miss the cache. An example of speculative invalidation is illustrated in Figure 3. The procedure for speculative downgrades is similar. A speculatively downgraded line is no different than one that had been normally downgraded: to complete a write after the downgrade an exclusive copy would have to be re-obtained. A separate state is also used for speculatively downgraded lines.

After initiating the speculative action the next step is scheduled by enqueuing the IHT index of the list's instruction in a speculative action queue. The cache controller dequeues and performs the actions; to avoid congestion in the simulations a minimum time between steps is imposed.

2.3 Scoring

Proper scoring is a critical factor in speculation effectiveness: If scores are too high many lines that are needed in the cache will be speculatively invalidated or downgraded

(*false positives*) and if scores are too low many preventable SCMs will occur. The scoring scheme described below was developed for the simulated system and is used in all simulations.

The score, a 5-bit signed integer, is based on the number of detected false positives and correct speculations and on the number of invalidations and downgrades. Scoring by outcome, correct prediction or false positive, ensures that effective speculation continues to be performed while harmful misspeculation is quickly stopped. Scoring by invalidations and downgrades (both speculative and normal) allows speculation to occur at a low rate in the absence of recent outcome information. Separate scores are used for invalidation and downgrading. The score is initialized to zero; speculative actions are taken if the score is non-negative.

Correct predictions are detected when a cache receives a would-have-invalidated message for a speculatively invalidated line (that is still present). Because an invalid line can be replaced, not all correct predictions will be detected. A false positive is detected on a miss or upgrade request to a line in a speculatively invalidated or downgraded state; for the same reason detection of false positives is also imperfect.

The score is incremented by one on non-speculative invalidation or downgrading, tending to turn speculation on. The score is decremented by one when a speculative action is taken, tending to turn speculation off and thereby limiting the number of "unverified" speculative actions. When a false positive is detected the respective score is decremented by 8; at most two consecutive false positives will turn off speculation. The invalidation score is incremented by 4 on a correct prediction and the downgrading score is incremented by 1 on a correct prediction.

These scores were arrived at by an informal parameter space search. Attempts to derive parameters using simple memory system models were less effective.

2.4 Hardware Details and Cost

Most of the cost of the speculative hardware is in the LHT and the controller needed at the caches. The controller hardware is no more elaborate than the hardware used to process coherence messages, and may be realized as an extension to that hardware. Because of the difficulty of making a useful cost estimate it will be assumed that the additional cost for the controller is moderate to small. The cost of the tables will be estimated by finding the amount of additional storage needed.

The simulated parameters were chosen for the SPLASH 2 benchmarks, these programs are used with smaller input sizes than are normally encountered in scientific work but the applications themselves (though not the kernels) are full-size programs [24]. To account for the scaled up input sizes in real systems cost will be found for a large cache size of 16MB. If the code size of the SPLASH 2 applications are representative of actual scientific applications then there should be little need to scale up the IHT size. In fact, larger input sizes would lead to instruction use frequencies that possibly favor an even smaller IHT since a smaller number of instructions would account for a given fraction of the dynamic instruction count. Nevertheless, to be conservative, the cost will be based on a system using a 1024-entry IHT, four times the simulated size.

Table 1: Line History Table Fields

Field Name	Size / bits	Description
Instruction ID	10	IHT index of instruction last accessing line. (Based on 1024-entry IHT.)
Downgrader	10	IHT index of speculatively downgrading instruction. (Used to detect false positives.)
Next Line	18	Cache address of next line in linked list. (Address bits needed to index the tag store and bits to select the set element (way) in a set associative cache.)
Previous Line	18	Cache address of previous line in linked list.

The LHT, since it must hold pointers for each cache line, is responsible for most of the cost. The size of a pointer is based on a 16MB cache with 64-byte lines (which holds

2^{18} lines). (If the cache is set associative the pointer includes bits to select the proper set element.) The table above gives the size (in bits) of each field in an LHT entry, along with a name and description. The total size of an entry is 56 bits, or about 11% of a 64-byte cache line.

Table 2: Instruction History Table Fields

Field Name	Size / bits	Description
Instruction ID	10	IHT index of instruction last accessing line.
Head	18	Address of list head. (Last line to be acted on.)
Tail	18	Address of list tail. (Next line to be acted on.)
Invalidation Score	5	Speculative invalidation score, a signed integer. Speculate if non-negative.
Reversion Score	5	Speculative downgrading score, a signed integer. Speculate if non-negative.
Invalidating	1	Indicates whether list traversal for invalidation in progress.
Downgrading	1	Indicates whether list traversal for downgrading in progress.
Shared	1	When list traversal starts, initialized to zero. Set to one if line found in the shared state during speculative downgrade list traversal.

The fields used in an IHT entry are listed above. The total entry size is 59 bits. Because the IHT has only 1024 entries its total size, 7552 bytes, is only a small fraction of the cache size.

Tags (to verify a hit) are not included in the IHT or LHT. The LHT does not need tags because either it is accessed in parallel with the tag store or it is accessed using a pointer which holds an element number. Pointers are always removed when a line is replaced or invalidated, so every pointer will retrieve a valid entry. The IHT is indexed using the low-order bits of an instruction address. Experiments were performed in which the IHT held tags; such systems performed only slightly better. Note that the result of an undetected IHT miss may be misspeculation but not incorrect execution.

The cost of the hardware at the memory modules is small 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 SLID increases cache storage requirements by little more than 11%.

3 Simulation Methodology

The effectiveness of SLID was tested using execution-driven simulation. Shared-memory multiprocessor systems using a write-invalidate, full-map directory cache coherence protocol were simulated. The experiments, using programs from the SPLASH 2 shared memory benchmark suite [24], were designed to determine values of certain parameters, to determine sensitivity to system configuration variations, and to determine the performance difference between a conventional system one using SLID, and one using DSI.

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 [15]. 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 and 2.6.

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 four 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 [15]. 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 [15]. Each processor has an associated memory module, sharing the network interface queue with messages bound for the processor.

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 [24]. 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 [24]. The problem sizes used are Radix, 262144 keys and radix 1024; LU, 256×256 matrix; FFT, 65536 points; Ocean, 258. The following input files were used: Cholesky, tk14.O; FMM, iput.16384; Water N^2, input.512; and Barnes, input. 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 described below tested several different cache configurations. The table gives the simulation parameters describing the *base configuration*. The differences from the base configuration will be noted for each experiment. Some parameters are explained below; see [15] for a detailed explanation of the network- and memory-related parameters.

A large cache size was chosen to reflect the size of a cache in a multiprocessor, where remote misses are expensive. At the size chosen, 2^{19} bytes, further increases in cache size have little effect on hit ratio (using the benchmark's scaled input sizes). The line size was chosen to minimize average execution time. A single-level cache was chosen to simplify simulations and interpretation of results.

Table 3: 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^{19} bytes
Cache Associativity	8
Cache Line Size	64 bytes
Cache Hit Latency	1 cycle
Directory Size	full map
Completion Buffer	5 stores
Memory Latency	10 cycles
Protocol Message Size	6 bytes (plus data)
Network Link Width	4 bytes
Hop Latency	20 cycles (plus waiting)
IHT Size	256 entries
Add new list element	3 cycles
Re-link list element	6 cycles
Minimum time between spec.	18 cycles

3.4 Dynamic Self Invalidation

For comparison, systems employing dynamic self invalidation were simulated. The implementation closely follows Lebeck and Woods' variation that uses version numbers and linked lists [16]. Like SLID, DSI speculatively invalidates lines that are expected to be normally invalidated before their next use. Candidate lines for speculative invalidation are identified based on their past behavior.

A line sent to a processor is a candidate for speculative invalidation if it is being returned to the processor after being written elsewhere or if it is to be re-written by the processor after other processors have read it [16]. In both cases the line being accessed had been normally invalidated or downgraded from the cache and it is assumed that it will be invalidated or downgraded again. The simulations here use 4-bit version numbers as described in [16] to detect these behaviors. Briefly, each line has a version number which is incremented whenever the memory controller grants exclusive access to a processor. Caches store the version numbers for cached lines, including those in invalid states. On a miss to an invalid line the cache controller will return the version number to the memory

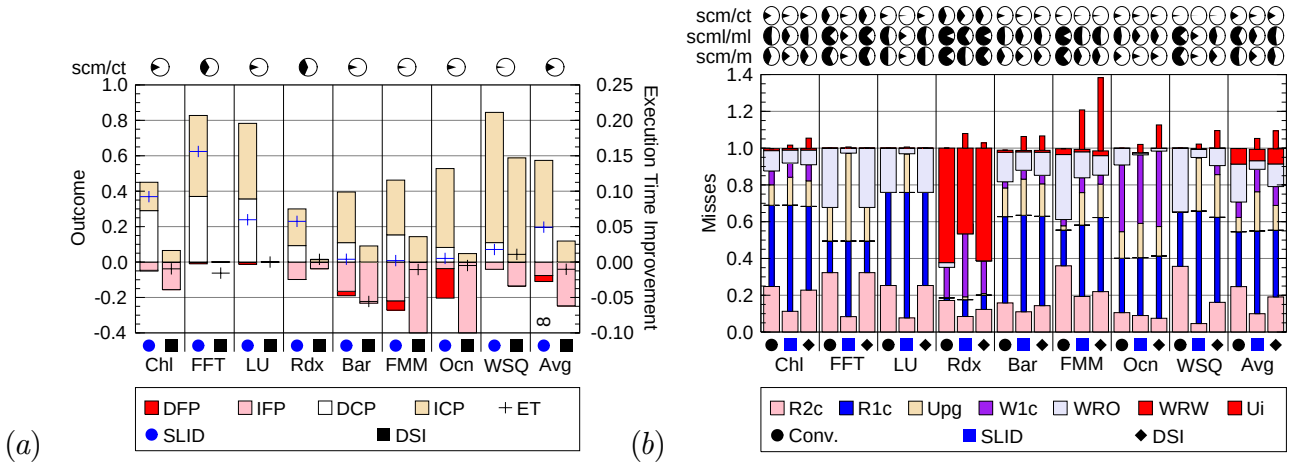

Figure 4. (a) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (b) categorization of misses (bars), fraction of misses to remote (second) cache (bottom row of pies), fraction of miss latency due to second cache misses (middle row of pies), and fraction of computation time stalled due to second cache misses (top row of pies). In (b) *Ui* indicates misses added by misspeculation. See Figure 1 for other segments. Segments scaled to number of misses except those added by misspeculation.

controller. Based on the latest version number and the line's state, the memory controller may return an ordinary or speculatively invalidate-able (SI) line.

The cache maintains a linked list of SI lines it has received. At synchronization points lines in the linked list will be speculatively invalidated. In the simulations here speculative invalidation is initiated when a processor enters a barrier and on a semaphore release. Once DSI speculative invalidation is initiated it proceeds at the same rate as SLID speculative actions (to avoid congestion).

4 Experiments

4.1 Base Configuration

SLID was first evaluated by simulating it alongside DSI and a conventional system, all using the base configuration parameters and running SPLASH 2 benchmarks. Results show that usable speedup is obtained and that about 60% of second cache misses are avoided; speedup is higher for the kernels because they spend more time stalled by cache misses. These results can be seen in Figure 4(a) where speculation outcome and execution time improvement (speedup minus one) for SLID and DSI are plotted. (Let t_x denote the

execution time of system x; the *execution time improvement* of x is $t_{\text{conv}}/t_x - 1$, where t_{conv} is the execution time of a comparable conventional system.) Execution time improvement is shown with a "+" for each benchmark and system and for an average (arithmetic mean of improvements) over all benchmarks. The average execution time improvement using SLID is 5% and is as high as 15%. (The choice of benchmarks was made before the experiments were run, all benchmarks initially chosen were used.) DSI improves the performance of Radix and Water N^2 but reduces the performance of other benchmarks.

The fraction of avoided second cache misses is shown by the bar segments above the axis in Figure 4(a). The segments labeled *DCP* show misses avoided by downgrading and those labeled *ICP* show misses avoided by invalidation. For example, the leftmost bar indicates that just over 40% of the second cache misses were avoided by SLID running Cholesky. Overall, SLID eliminates almost 60% of second cache misses for these benchmarks, and so it is quite effective. DSI eliminates many misses on Water N^2 and some on three other benchmarks but is not as uniformly effective.

Avoiding second cache misses helps improve execution time by halving access time. The amount of improvement depends upon how often execution was stalled by second cache misses, how many misses were added by misspeculation, and by memory access pattern changes (*e.g.*, speculative actions interfering with normal memory operations).

Each added miss approximately undoes the benefit of one correct speculation. The number of added misses is shown in Figure 4(a) by segments below the axis (normalized to the number of second cache misses that would occur); the segments individually show misses added by speculative downgrading and speculative invalidation. The number of added misses is usually small for SLID, and always smaller than the number of correct predictions. DSI suffers from many false positives, which may explain the lack of performance improvement in some cases.

The pie charts in Figure 4(a) show the fraction of non-idle CPU time in which execution was stalled waiting for second cache misses in conventional systems. (Execution

stalls on all loads and on stores when the write buffer fills.) The first four benchmarks spend more of their time waiting for memory than the last four, this partly explains the difference in performance improvement.

Details on misses appear in Figure 4(b). The data for conventional systems is the same as Figure 1; for the speculative systems the number of added misses is shown by segments above the $y = 1$ plane. The misses shown below $y = 1$ include all misses except those added by speculation. This arrangement makes it easy to see which misses are being avoided. (The length of individual segments are not exact because false positives were not collected by miss type.)

The data shows that far more writes encounter second cache misses (as one might expect) and that SLID has a greater impact on second cache write misses than on read misses. DSI is roughly half as effective and also adds more misses. Note that the proportion of reads and writes can vary between the conventional, SLID, and DSI systems, as with FMM. This is due to execution speed affecting spin lock iterations, work distribution, etc.

The pie charts in Figure 4(b) show the impact of second cache misses on execution time. The bottom row shows the fraction of all misses that are second cache misses, the middle row shows the fraction of miss latency due to second cache misses, and the top row shows the fraction of computation time stalled due to second cache misses (approximately). For benchmarks such as Cholesky and FFT, second cache misses have a significant impact on performance and SLID eliminates most of them. For others, such as Water N^2, SLID has a large impact on miss latency but miss latency has only a small impact on execution time.

4.2 System-Configuration Effects

The effect of cache size is plotted in Figure 5 for caches ranging from 2^6 sets (2^{15} bytes) to 2^{13} sets (2^{22} bytes). In Figure 5(a) each pair of bars shows the average over all

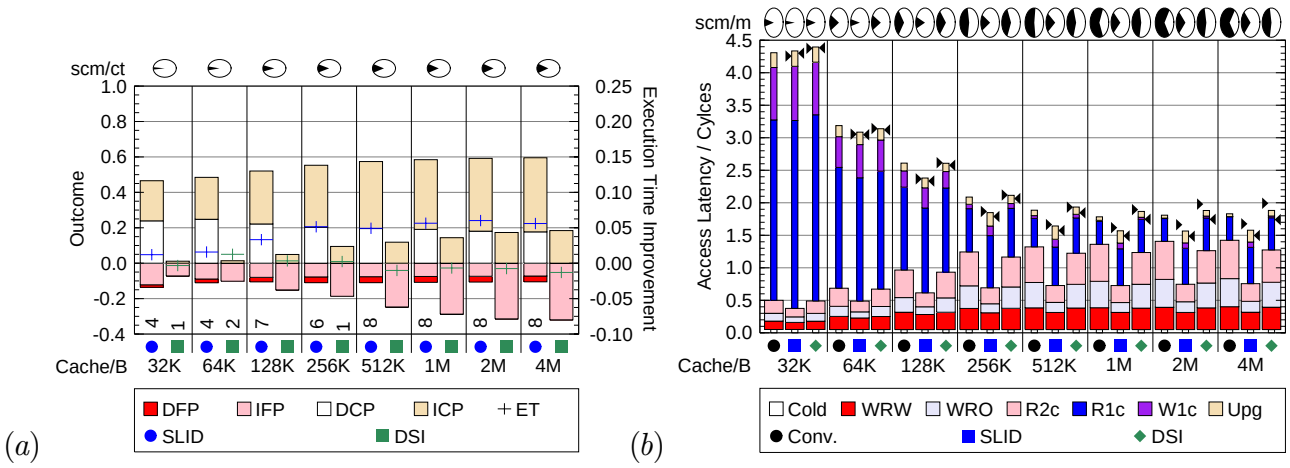

Figure 5. Effect of cache size on (*a*) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (*b*) miss latency per access (bars) and fraction of misses that are to a second cache (pies). See caption in Figure 4(*a*) for a description of the segments used in (*a*). Segments in (*b*) show contribution of miss types to access latency; the segment labels are similar to Figure 4(*b*) except cold misses are shown in separate segments (rather than in *R1c* and *W1c*).

benchmarks and the numbers along the bottom of the plot show the number of benchmarks for which the respective system is best. For example, at the smallest cache size execution was fastest for four benchmarks using SLID, one benchmark using DSI, while three benchmarks ran fastest on a conventional system. DSI is more effective at smaller sizes, while SLID is more effective at larger sizes. DSI may improve at smaller cache sizes due to fewer added misses. SLID is better at larger cache sizes because second cache misses have a greater impact on the execution time of systems with larger caches, as shown by the pie charts.

The effect of cache size on miss latency per access is plotted in Figure 5(*b*). Each bar shows the miss latency per memory access, segments indicate the contribution by miss type. Second cache misses are shown with wide segments. Increasing cache size clearly reduces access latency, but the absolute contribution of second cache misses increases as more shared data remains in remote caches.

The impact of added misses and congestion are shown by the small triangles. The triangle to the left of each bar shows what latency would be (approximately) using miss latencies from the conventional system as follows: An average miss latency is found for

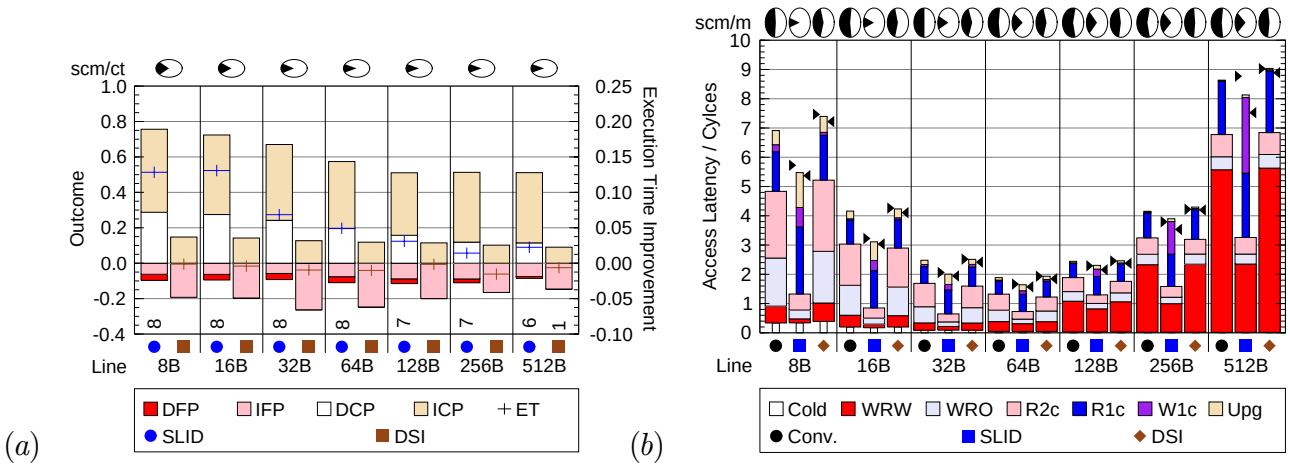

Figure 6. Effect of line size on (a) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (b) miss latency per access (bars) and fraction of misses that are to a second cache (pies). Cache size held constant at 2^{19} bytes. See caption of Figure 5.

one-cache and second cache misses on the conventional system, using these and the number of accesses, one-cache, and second cache misses on the test (DSI or SLID) system a new miss latency per access is computed for the test system. If the triangle is above the bar there is less congestion (lower miss latency) on the test system. This analysis indicates that on smaller caches SLID adds to congestion, lowering performance, while on larger cache sizes congestion is reduced, yielding higher performance. The triangle to the right of a segment indicates what the miss latency per access would be (approximately) without the added misses. The impact is relatively small, indicating added misses are not a major problem.

The effect of line size is plotted in Figure 6 for lines ranging from 8 bytes to 512 bytes with the cache size held constant at 2^{19} bytes. Figure 6(a) shows that SLID is more effective at smaller line sizes, eliminating nearly 80% of the second cache misses when 8-byte lines are used. A possible reason is that the instruction correlation used is more effective on sequentially accessed data. As the line size is increased, a greater proportion of misses are to non-sequential data. As can be seen in the pie charts above Figure 6(b), the proportion of second cache misses increases only slightly with line size. Nevertheless, even at large line sizes SLID can eliminate over 50% of second cache misses. A system, of

course, would be designed with the line size that minimizes access latency. The base line size was chosen that way, as can be seen in Figure 6(b) where miss latency per access is plotted. The increase in *WRW* misses is due primarily to Radix.

5 Related Work

Many approaches have been suggested or used to reduce multiprocessor access latency, including second cache misses. Improved data layout and algorithms help, but of course can't eliminate all sharing or second-cache misses. Hardware approaches have also been used, for this discussion they will be split into two types: prefetching and cache management.

5.1 Prefetch

Software prefetch schemes use *prefetch instructions,* inserted by the programmer or compiler, which bring data to a cache but otherwise have no effect. (See [14,20] for prefetching on serial systems and [9,17] for parallel systems.) Software prefetching works well when the data needed can be identified far enough in advance. SLID 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. Lines 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 prefetch schemes are purely sequential [6,8] or based on previous address sequences [13,11].

The speculation performed by SLID is perhaps easier than that performed by prefetch because a prefetch scheme predicts that a line will be used at a particular processor while

SLID only predicts that a line will be used by some other processor. Thus SLID can reduce the latency of many accesses that a prefetch mechanism cannot predict, and so the two schemes are complementary.

5.2 Cache Management

The cache management schemes reduce access latency by adapting the cache management and coherence protocol to a program's dynamic behavior. These include SLID, DSI, and many others. DSI, discussed in Section 3.4, is most similar to SLID in that it eliminates second cache misses. Other approaches are discussed below.

Several investigators developed systems that speculatively fetch a line in an exclusive state on a read miss. This reduces the number of message transit delays associated with sharing, though does not reduce second cache misses as directly as DSI and SLID. Such schemes were described by Cox and Fowler [5] and in a similar paper Stenström, Brorsson, and Sandberg [23]; they observe that a pattern of read/invalidate/misses 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 by the processor. Cox and Fowler show execution time reductions as high as 19%.

Along similar lines Kaxiras and Goodman [12] predict a store following a load miss using instruction addresses. Such loads are identified using a PC-indexed table holding load effective addresses and a prediction counter. On a store miss the table is associatively searched using the effective address, if a matching entry is found the counter is incremented. Based on simulations of systems using a SCI [10] shared memory interconnect, they show speedups of 1.13 and 1.30 for two benchmarks and little effect on five others, including Barnes and Ocean. (The results are not directly comparable because of cache differences, network latency, and benchmarks other than Barnes.)

These migratory schemes are only effective for load misses followed by stores while SLID and DSI are potentially effective on all sharing-related misses.

Spec. Inv. and Updating Based on Instruction History

Some cache management schemes use profiling or software assistance to determine how the cache will handle each instruction. 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 [19]. 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 a downgrade. In the method described here, a line is speculatively downgraded only if another line last-accessed by the same instruction is downgraded. This could happen long after the write executed or after some lines accessed by the write were subsequently written by other instructions.

Other schemes attempt to determine the behavior of accesses to a location [1,5,16,23]. In an *adaptive caching* scheme described by Bennett, Carter, and Zwaenepoel [1], 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, for example write invalidate or write update. Trace driven simulations show that traffic volume can be reduced if a delayed update is used to handle writes for appropriate classes. (When delayed update is used writes to a line are propagated to other processors caching the line when a "synchronization point" is crossed [1].) The work reported was preliminary and so did not include dynamic methods for classifying data or evaluating performance.

Mukherjee and Hill [21] propose a coherence protocol message predictor analogous to a two-level PAp (local history) branch predictor [25]. The first-level tables are indexed by effective address, first-level table entries hold the last message (or last few messages) arriving for the address and its sender. These are hashed with effective address to index the second level table, holding predictions. Next message predictions are made at cache and memory directories; they obtain prediction accuracies of 65% to 86%. Applications for the predictor were suggested but none were simulated, so comparison with other schemes

is not possible. This scheme may well share two weaknesses of DSI: the behavior of each block must be learned and there is no information on when to act on predictions.

6 Conclusions

A method of speculative invalidation and downgrading of cache lines, SLID, was described. Using these speculative actions the number of second cache misses is reduced significantly with only a minor impact on the miss rate. The cost of adding this hardware is approximately the same as the cost of increasing cache size by about 11%. (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 11% would be more cost effective in systems with undersized caches while adding SLID would be the better route 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 miss 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.

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.

7 References

[1] Bennett, J.K., Carter, J.B., and Zwaenepoel, W. Adaptive software cache management for distributed shared memory architectures. *ACM Computer Architecture News.* **18**, 2 (May 1990), 125-134.

[2] Brewer, E.A., Dellarocas, C.N., Colbrook, A., and Weihl, W.E. Proteus: a high-performance parallel-architecture simulator. *Proceedings of the ACM SIGMETRICS.* 1992.

[3] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A. Directory-based cache coherence in large-scale multiprocessors. Massachusetts Institute of Technology. 1989, VLSI Memo, 1989.

[4] Chen, T.F., and Baer, J.L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers.* **44**, 5 (May 1995), 609-623.

[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] Dahlgren, F., Dubois, M., and Stenström, P. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems.* **6**, 7 (July 1995), 733-746.

[7] Dahlgren, F., and Stenström, P. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems.* **7**, 4 (April 1996), 385-398.

[8] Fu, J., Patel, J.H., and Janssens, B.L. Stride directed prefetching in scalar processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture.* 1992, pp. 102–110.

[9] Gharachorloo, K., Gupta, A., and Hennessy, J.L. Two techniques to enhance the performance of memory consistency models. *Proceedings of the International Conference on Parallel Processing.* August 1991, vol. I, pp. 355–364.

[10] Gustavson, D.B. The scalable coherent interface and related standards projects. *IEEE Micro Magazine.* **12**, 1 (February 1992), 10-22.

[11] Joseph, D., and Grunwald, D. Prefetching using Markov predictors.*Proceedings of the International Symposium on Computer Architecture.* June 1997, pp. 252–263.

[12] Kaxiras, S., and Goodman, J.R. Improving CC-NUMA performance using instruction-based prediction. *Proceedings of the Proceedings of the 5th Symposium on High Performance Computer Architecture.* January 1999.

[13] Kim, S.B., Park, M.S., Park, S.H., Min, S.L., Shin, H., Kim, C.S., and Jeong, D.K. Threaded prefetching: an adaptive instruction prefetch mechanism.*Proceedings of the Microprocessors and Microprogramming.* 1993, vol. 30, pp. 1–15.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Lilja, D.J. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Computing Surveys.* **25**, 3 (September 1993), 303-338.

[19] 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.

[20] 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.

[21] Mukherjee, S.S., and Hill, M.D. Using prediction to accelerate coherence protocols. *Proceedings of the 25th Annual International Symposium on Computer Architecture.* June 1998, pp. 179–190.

[22] Stenstrom, P. A survey of cache coherence schemes for multiprocessors. *IEEE Computer.* **23**, 6 (June 1990), 12-24.

[23] Stenstrom, P., Brorsson, M., and Sandberg, L. An adaptive cache coherence protocol optimized for migratory sharing. *Proceedings of the International Symposium on Computer Architecture.* May 1993, pp. 109–118.

[24] 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.

[25] Yeh, T.Y., and Patt, Y.N. A comparison of dynamic branch predictors that use two levels of branch history. *Proceedings of the International Symposium on Computer Architecture.* May 1993, pp. 257–266.

List of Figures

Figure 1. Categorization of misses (bar chart) and impact of second cache misses (pie charts). Wide segments are second cache misses. Plotted from the bottom up, miss types are: *R2c* read exclusive elsewhere; *R1c* read shared elsewhere or not cached; *Upg* write shared locally and not cached elsewhere; *W1c* write not cached; *WRO* write shared; *WRW* write exclusive elsewhere. Bottom row of pie charts: fraction of misses to second cache. Middle row of pie charts: fraction of miss latency due to second cache misses. Top row of pie charts: fraction of computation time stalled due to second cache misses.

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.

Figure 3. 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.

Figure 4. (*a*) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (*b*) categorization of misses (bars), fraction of misses to remote (second) cache (bottom row of pies), fraction of miss latency due to second cache misses (middle row of pies), and fraction of computation time stalled due to second cache misses (top row of pies). In (*b*) *Ui* indicates misses added by misspeculation. See Figure 1 for other segments. Segments scaled to number of misses except those added by misspeculation.

Figure 5. Effect of cache size on (*a*) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (*b*) miss latency per access (bars) and fraction of misses that are to a second cache (pies). See caption in Figure 4(*a*) for a description of the segments used in

(a). Segments in (b) show contribution of miss types to access latency; the segment labels are similar to Figure 4(b) except cold misses are shown in separate segments (rather than in *R1c* and *W1c*).

Figure 6. Effect of line size on (a) fraction of second cache misses avoided (bars), execution time improvement ("+"), and fraction of computation time stalled by second cache misses (pies); and (b) miss latency per access (bars) and fraction of misses that are to a second cache (pies). Cache size held constant at 2^{19} bytes. See caption of Figure 5.