Speeding up the Memory Hierarchy in Flat COMA Multiprocessors[1]

Liuxi Yang and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
lyang,torrella@csrd.uiuc.edu
http://www.csrd.uiuc.edu/iacoma

Abstract

Scalable Flat Cache Only Memory Architectures (Flat COMA) are designed for reduced memory access latencies while minimizing programmer and operating system involvement. Indeed, to keep memory access latencies low, neither the programmer needs to perform clever data placement nor the operating system needs to perform page migration. The hardware automatically replicates the data and migrates it to the attraction memories of the nodes that use it. Unfortunately, part of the latency of memory accesses is superfluous. In particular, reads often perform unnecessary attraction memory accesses, require too many network hops, or perform necessary attraction memory accesses inefficiently. In this paper, we propose relatively inexpensive schemes that address these three problems. To eliminate unnecessary attraction memory accesses, we propose a small direct-mapped cache called Invalidation Cache (IVC). To reduce the number of network hops, the IVC is augmented with hint pointers to processors. These hint pointers are faster and have more applicability than in older hint schemes. Finally, to speed up necessary accesses to set-associative attraction memories, we optimize the locality of windows in page-mode DRAMs. We evaluate these optimizations with 32-processor simulations of 8 Splash and Perfect Suite applications. We show that these optimizations speed up the applications by an average of 20% at a modest cost.

1 Introduction

In scalable shared-memory multiprocessors, long-latency memory accesses can cause crippling slowdowns. A processor issuing a read request may have to remain stalled for many cycles before the data is finally supplied by the memory system. One class of cache-coherent shared-memory machine that is designed for reduced memory access latencies while minimizing programmer and operating system support is called Flat Cache-Only Memory Architecture (Flat COMA) [16].

Flat COMA machines, like all other COMA machines [6, 8, 10, 14, 16], organize the distributed memories as caches called attraction memories. Like caches, attraction memories are composed of data array and tag array. A tag keeps information on the memory line that is currently stored in this location: enough of its physical address to identify the line, and its sharing state, for example shared, invalid, or exclusive. In addition, to reduce the number of conflicts, attraction memories are usually associative, often 4- to 16-ways. Examples of COMA machines, also called *All-Cache*, are the Swedish Institute of Computer Science's DDM1 [6] and Kendall Square Research's KSR1 [10].

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098, DARPA Contract No. DABT63-95-C-0097, NASA Contract No. NAG-1-613, and Intel Corporation.

A major advantage of COMA machines is that data automatically replicates and migrates in the different attraction memories of the machine, moving close to the processors that use it. Therefore, when a processor suffers a cache miss on a line, it first accesses its local attraction memory to see if the line is there. In the case where the line was displaced from the cache due to conflicts, chances are that the attraction memory still has a copy that can be supplied quickly. As a result, cache conflict misses become cheap. This occurs without requiring the operating system to perform page migration or the programmer to cleverly place the application's data in the distributed memory for locality.

From this discussion, it is clear that individual memory lines in COMA machines do not have a fixed location used as a repository. This is unlike non-COMA machines, where each memory line has a repository in an agreed-upon memory module called home. The home keeps a copy of the line unless the line is dirty in one cache, in which case the home knows where the line is. In first-generation COMA machines, there was no direct way to find out where a copy of a line was. Instead, the memory hierarchy had to be searched. Such search, even if done intelligently, was likely to cause some performance degradation.

Flat COMA machines are unlike traditional COMA machines in that each individual memory line has a directory entry in an agreed-upon node [16]. This node is called the home. The home, however, does not have a repository for the data. Therefore, in the general case, a message sent to the home by a processor requesting data will be forwarded to a third node. Note, however, that no search of the memory hierarchy is necessary now to find a copy of the data.

COMAs have several well-known sources of overhead [8]. First, each attraction memory line needs a tag. In addition, the attraction memory needs to be larger than the size of the application data to allow for the latter to replicate. The ratio between the size of the application data and that of the total attraction memory is called the *Memory Pressure*. Obviously, the higher the memory pressure is, the harder it is for the replicated data to find space.

In fact, for high memory pressures, it is likely that, when a processor replicates a line by reading a fresh copy of it into its attraction memory, no space is available in the corresponding set. A space is then forced by overwriting one of the lines in the set. This is fine unless the overwritten line is the last copy in all attraction memories. If so, or we suspect that it might be so, the line is sent to another attraction memory for it to keep. Such event is called a line replacement. The lines that must be sent to another attraction memory are identified in the tag as *master*. Master lines include those in state exclusive and one of the copies of the lines in state shared.

Line replacements affect processor reads only indirectly: they increase the network traffic and, therefore, slow down all messages. Unfortunately, other characteristics of Flat COMA machines slow down reads directly. Given that processors tend to suffer stalls following read misses, these other charac-

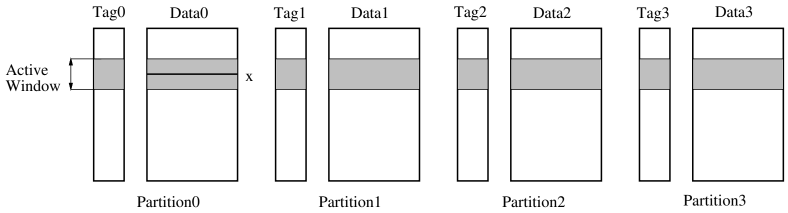


Figure 1: Position of the active tag and data windows in a 4-way set associative attraction memory after location x has been accessed.

teristics are more important. In this paper, we propose relatively inexpensive solutions to three situations where reads take longer than they should because they perform unnecessary attraction memory accesses, require too many network hops, or perform necessary attraction memory accesses inefficiently. To eliminate unnecessary attraction memory accesses, we propose a small direct-mapped cache called Invalidation Cache (IVC). To reduce the number of network hops, the IVC is augmented with hint pointers to processors. Finally, to speed up necessary accesses to set-associative attraction memories, we optimize the locality of windows in page-mode DRAMs. We evaluate these three optimizations with 32-processor simulations of 8 Splash [15] and Perfect Suite [1] applications. We show that these optimizations speed up the applications by an average of 20% at a modest cost.

This paper is organized as follows: Section 2 describes the inefficiencies that increase the read latency in Flat COMA machines; Section 3 presents our solutions to address them; Section 4 describes the methodology used to evaluate our solutions; Section 5 evaluates them; and Section 6 discusses related work.

2 Inefficiencies in Flat COMA

Some processor reads in Flat COMA machines are inefficient because they perform unnecessary attraction memory accesses, require too many network hops, or perform necessary attraction memory accesses inefficiently. In this section, we analyze each of these effects in turn. Note that some of these effects may also occur in non-COMA machines.

2.1 Unnecessary Attraction Memory Reads

In COMA machines, a secondary cache miss is always followed by a check in the node's attraction memory. This is unlike in non-COMA machines, where a secondary cache miss triggers a message to the corresponding directory, which may be in a remote node. Clearly, the local attraction memory access in Flat COMA is beneficial if the data is found in the attraction memory. However, if the attraction memory access misses, we have unnecessarily delayed the transaction by a time equal to the memory access time. Clearly, we would like to skip the accesses that will miss and proceed directly to check the corresponding directory.

The latency of this failed attraction memory access depends on the technology used to implement the attraction memory tags. To keep memory costs down, suppose that we use DRAM. In the architecture examined in this paper (Section 4.1), processor reads are serviced in an average of 138 or 188 cycles depending on whether they need to visit one or two remote nodes respectively. Out of these latencies, the failed access to the local attraction memory takes an average of 24 cycles, or about 15% of the time.

2.2 Too Many Network Hops

In Flat COMA machines, a secondary cache miss on a line shared by several nodes may require a three-hop transaction. This is because the home node may not keep a copy of the line. As indicated above, in non-COMA machines, no more than two hops are necessary in such a case.

The cost of this extra hop depends on the speed of the network and the latency of a directory check. The latter is relevant because, when the message is received by the home node, the directory is checked before forwarding the message. If, instead, the message could be sent from the requesting processor directly to a node with a copy of the data, we would save one network hop and one directory access. For the machine examined in this paper (Section 4.1), this would change the latency of the transaction from 188 to 118 cycles on average. This is a 37% savings.

2.3 Inefficient Attraction Memory Reads

In COMA machines, when a read request finally reaches the correct attraction memory, it often performs the memory access inefficiently. To see why, consider page-mode DRAMs [9]. These memories effectively have a cache inside. After a given memory location x in the DRAM is accessed, an active window of contiguous locations surrounding x can be accessed in very short times. These active windows are aligned at fixed memory boundaries and each of them can easily be 2-64 Kbytes in size, depending on the memory organization.

Consider a very simple model of memory organization such that an A-way set-associative attraction memory has one active window in each of its A memory partitions. For example, Figure 1 shows the active windows for a 4-way set-associative attraction memory right after location x has been accessed. While the organization of real memories may be different and is usually more complicated, this paper uses this simple model because it is enough to grasp the problem and the solution. Clearly, an A-way set-associative attraction memory built as in Figure 1 has, at any given time, A localities that can be accessed with high speed. These localities can be used to exploit the spatial locality of the references to several objects.

Unfortunately, these localities are often wasted for two reasons. The first one is that, when the memory is accessed, the active windows in all partitions end up moving to the same sets (Figure 1). This is a limitation because the several active objects that exist at any given time are unlikely to be all in the same group of a few sets. It would seem that we could put *A-1* of these active windows to better use. The second reason is that external events disturb the state of the active windows. The most obvious example is when an attraction memory is forced to accept a line that was displaced from a second attraction memory. In this case, the address of such a line is unrelated to the addresses of the requests that the first attraction memory was receiving. Therefore, after the attraction memory accepts the line, all its A active windows are likely to be left in a useless position. The next access to the attraction memory is likely to miss all active windows.

The time that can be saved by improving the use of the active windows depends on the speed of the DRAM used. For the DRAM simulated in this paper, a RAS cycle takes 50 ns, a CAS cycle takes 10 ns, and to perform the tag match and put the data out takes a further 10 ns. Furthermore, to perform a row address match to check whether the access falls inside the active window takes 10 ns. Therefore, an access inside the active window, which we call a *Window Hit*, takes 20 ns (row address match in parallel with the CAS plus the time to match the tag and put the data out). An access outside the active window, which we call a *Window Miss*, takes 80 ns (row address match that fails, RAS, CAS, and the time to match the tag and put the data out). Furthermore, an access that does not check the active window takes 70 ns (RAS, CAS, and the time to match the tag and put the data out). Therefore, every time that we transform a window miss into a window hit we save 80-20=60ns, or 12 cycles for the system described in Section 4.1. This cuts down the processor stall time in a read miss satisfied by the local attraction memory by about 30%. Note that for a window-based system to be better than a non window-based system, the window hit rate must reach a certain threshold. For our parameters, this threshold is 16.7% because $20ns * 0.167 + 80ns(1 - 0.167) = 70ns$.

3 Improving Flat COMA

These three inefficiencies can be addressed with the following three optimizations.

3.1 The Invalidation Cache

To prevent unnecessary accesses to the attraction memory we propose the *Invalidation Cache* (IVC). The IVC is a small direct-mapped cache that is accessed at the same time as the secondary cache and that contains the addresses of some of the lines that are *not* in the local attraction memory. The IVC does not contain any data. The IVC is faster than the secondary cache and does not slow it down: before the secondary cache detects the miss, the IVC determines whether or not it contains the address of the line. If the IVC has the address of the line, after the secondary cache detects the miss, the attraction memory access is skipped.

A set of lines that are surely not in the local attraction memory are those that have been invalidated from it. Lines can be invalidated by invalidation messages sent by a directory to keep data coherent (real invalidations) or by line displacements due to space conflicts in the attraction memory (self invalidations).

The IVC can easily detect all real and self invalidations by snooping on the local bus (Figure 2). Indeed, a real invalidation message reaches the attraction memory via the bus; the self invalidation of a line in state master generates a bus message to send the line to another node; finally, the self invalidation of a line in state non-master still has to generate a bus message to invalidate the entry from the secondary cache. We could, therefore, record the addresses of all these invalidations in the IVC. However, because we want to keep the IVC relatively small, we record only a subset of them. We would like to record only the invalidations for lines that (1) are likely to be referenced soon by the processor and (2) we have an idea of what other processors keep a copy of the line. The first requirement is designed to increase the chances that the IVC entry is used before being displaced from the IVC. The second requirement is necessary as will be shown in the next section.

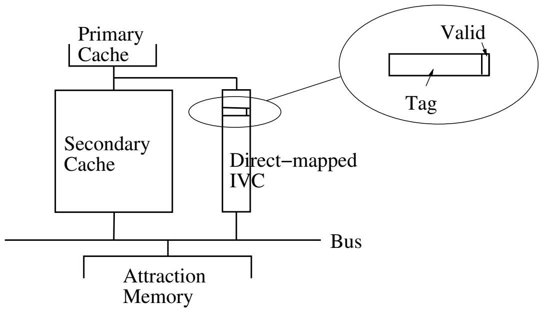


Figure 2: IVC organization.

Clearly, the self invalidation of lines in state non-master does not fulfill the second requirement: we do not know what other node has the line. Furthermore, of all the self-invalidated lines, those in state master have a higher chance of being referenced sooner. This is because, in our protocol, when a processor reads a line from a remote node, it gets the master copy. Therefore, for locality reasons, if a node has the master copy, it has a high chance of being the next processor to access the line. Overall, therefore, our IVC allocates an entry only when it detects a real invalidation or a self invalidation of a line in state master. These entries can later be removed from the IVC due to conflicts in the IVC or when the IVC, which continuously snoops the bus, detects that the line is being inserted back again into the attraction memory.

Adding IVCs to a system is relatively inexpensive. This is because the cache coherence protocol is minimally affected, the secondary cache controller is not affected at all, and IVCs are small. Indeed, each IVC entry contains only an address tag and a valid bit (Figure 2). To compute the size of a typical IVC, consider a 64-node machine with 64 Mbytes of attraction memory per node. In such a machine, a physical address is represented with 32 bits. Suppose that the size of the secondary caches is 1 Mbyte and that cache lines are 64 bytes long. A direct-mapped IVC with as many entries as the secondary cache will need about 26 Kbytes.

The effectiveness of the IVC is difficult to predict. On the one hand, the IVC, by collecting real invalidations, attempts to capture the coherence working set of the application. The coherence working set is the fraction of the working set composed of write-shared data. The coherence working set is clearly smaller than the working set, which suggests that the IVC needs fewer entries than the secondary cache. On the other hand, the IVC also captures a number of lines displaced from the attraction memory. Overall, therefore, the effectiveness of the IVC depends on the locality of the coherence working set and the lines displaced in state master, and on the combined size of these two sets of lines. Note, however, that the IVC can not possibly eliminate all unnecessary attraction memory accesses: start-up effects and displacement of lines from the attraction memory in state shared will still cause unnecessary memory accesses.

3.2 Invalidation Caches with Hints

To address the problem discussed in Section 2.2, Gupta *et al* [4] and Bjorkman *et al* [2] have proposed the *hint* schemes. The idea is for the lines in the attraction memory to store the ID of a processor that is likely to have an up-to-date copy of the line. This ID is called *hint*. In the scheme proposed by Bjorkman *et al*, when a processor misses on a line, it sends two requests: one to the home of the line and one to the hint processor. If the hint processor has the data, it will supply it and, in parallel, inform the home; the home will not reply to the original message because it knows that the hint processor

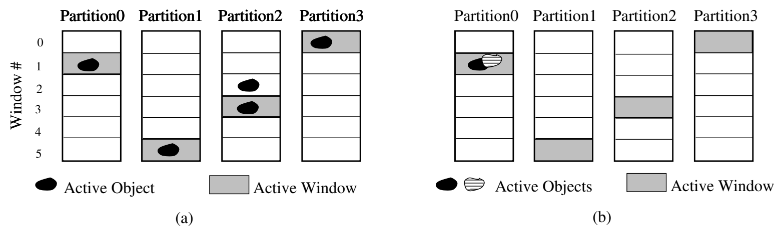


Figure 3: Active windows in an optimized 4-way set-associative attraction memory.

will reply directly. As a result, the requesting processor gets the data in 2 hops. If, instead, the hint processor does not have the data, the home node, which knows it, will forward the request to a fourth processor that has the data. The latter node will supply the data. In this case, the requesting processor gets the data in 3 hops.

Gupta *et al* [4] and Bjorkman *et al* [2] have proposed two types of hints, namely *invalid* and *shared*. An invalid hint is stored when a line is invalidated. The hint is the ID of the invalidating processor and it replaces the data in the line. A shared hint is stored when a line in the attraction memory is read by another processor: the hint is the ID of the reading processor and it is stored in a special field close to the line; no action is then taken when the line is invalidated. Overall, since these hints are useful for lines that have been invalidated, we call them *coherence* hints.

While coherence hints are good, we also need to handle lines that have been displaced from the attraction memory by conflicts, not just by real invalidations. Furthermore, we cannot afford to access memory to check hints like Gupta *et al* and Bjorkman *et al* suggested. That would be an unnecessary memory access. These two problems can be addressed by including a hint in each entry of the IVC. The attraction memory stores no hints. For the IVC entries corresponding to real invalidations, we store a *coherence* hint. For the IVC entries corresponding to self invalidations of master lines, we store a *conflict* hint: the ID of the processor to which the master copy was sent to. If the line replacement algorithm is such that the master copy is always sent and stored in its home node, the hint is the ID of the home node. Otherwise, if the local node decides what node to send the line to, the IVC is updated accordingly. Finally, if the algorithm used is Joe and Hennessy's [8], the IVC stores the ID of the processor that supplies the line causing the displacement.

Overall, the cost of this optimization is the extra complexity of the cache coherence protocol [2]. The size of the IVC, however, changes only slightly. For the 64-processor machine described in Section 3.1, the hint adds 6 bits to each entry. The size of the IVC is now about 38 Kbytes.

3.3 Improving Window Locality

To optimize accesses to attraction memories with page-mode DRAMs we exploit three ideas: we let the active windows in different memory partitions drift to different groups of sets, we encourage memory lines that belong to the same object to gather together in the same memory partition, and we prevent external events from disturbing the position of the active windows. In the following, we describe these ideas first and then consider some related issues.

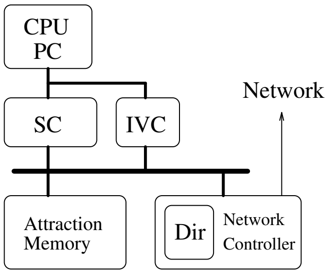
3.3.1 Description of the Scheme

The first idea, letting active windows in different memory partitions move independently to include different sets (Figure 3-(a)), helps capture more localities of active objects. In our scheme, we decouple the active windows in the different memory partitions and let them follow the localities as they are dynamically outlined by the requests that reach the memory. As we will see, this decoupling is accomplished by issuing extra RAS cycles to "restore" the previous position of the active windows. Intuitively, with this approach, in an A-way set-associative attraction memory, we can capture A objects with high speed. This is much better than the thrashing of active windows that would occur in the unoptimized memory of Section 2.3 when the objects that are being accessed are spread as in Figure 3-(a).

The second idea, namely encouraging the lines that belong to the same object to gather together in the same memory partition, is necessary because the attraction memory itself is a cache. A window in a memory partition does not necessarily contain lines with contiguous physical addresses. Instead, it contains lines whose addresses modulo the size of a memory partition are contiguous. Consequently, we have to be careful when we insert a line in an attraction memory after an attraction memory miss. If we insert the line in a randomly-chosen memory partition, it is likely that the different lines of an object will end up scattered in different memory partitions. As a result, one active window in one memory partition will not be able to capture the complete object and the memory accesses will be slower.

To group the different lines of an object in one memory partition and, therefore, capture them with a single active window, we use the principle of locality. In the memory controller, we add, for each memory partition, four registers organized as a stack. These registers we call the *RecentWindow* registers. The registers for a given memory partition contain the IDs of the four most recently-accessed windows in the memory partition. In particular, the register at the top of the stack has the ID of the active window for its memory partition. Then, when, after an attraction memory miss, we have to insert a line in the attraction memory, we select the memory partition based on the *RecentWindow* registers. Specifically, given that the line can be inserted in one and only one window per memory partition, we select the memory partition where the corresponding window is closest to the top of its *RecentWindow* stack. Given that references to the different lines of an object tend to happen clustered in time, this algorithm will encourage the convergence of the lines of the object into one single memory partition.

Every time that an access hits in the attraction memory or an access misses and, as a result, new data has to be inserted, the *RecentWindow* stack of the corresponding memory partition is updated: the window accessed becomes the active window and moves to the top of the stack. In addition, however, we need to prevent the pollution of the stacks and the active windows of the other memory partitions. This is the third idea suggested above. Indeed, a memory partition should modify neither its active window nor its stack in the following three cases. The first case is when the memory partition misses and no data will be inserted in the memory partition as a result of the miss. This occurs to *A-1* partitions of an A-way set-associative memory on a miss. The second



Transaction	Cycles
Primary Cache	1
Secondary Cache	8
Attraction Memory	
Window Hit	32
Window Miss	44
2–Hop (Data in AM)	108–132
3–Hop (Data in AM)	167–187

Figure 4: Node organization and typical latencies of round trip transactions without contention. In the figure, PC, SC, and AM stand for primary cache, secondary cache, and attraction memory.

case is when the attraction memory has to accept a line that was displaced from another attraction memory. In this case, we insert the line in a memory partition such that the window where the line goes to, is not in the partition's *RecentWindow* stack. A final case is when the attraction memory receives a write back. In neither of these three cases the access is an indicator of a new locality that the active window should move to include. In such cases, the *RecentWindow* stack is not modified and we issue a "correcting" RAS cycle to return the active window to the position that it had before the access.

3.3.2 Other Issues to Consider

One problem of this scheme could be that, in some pathological cases, it could induce additional attraction memory misses. This could occur when, in an attraction memory miss, a line must be inserted in a set that has some lines in state master and others in state invalid. In traditional schemes, the line would probably be inserted so that it overwrites an invalid line. In our scheme, however, we may decide to overwrite one of the lines in state master. The latter line, which we call the displaced line, would need to be moved. In a naive scheme, the displaced line would be sent to another node and, consequently, would cause a miss if it were referenced later by this processor. Intuitively, this is what occurs when two active objects overlap in the same window of the same memory partition (Figure 3-(b)) and the processor references the objects in an alternate manner. To solve this problem to some extent, when our algorithm displaces a master copy, it does not send it to another node if it can avoid it. Indeed, if it finds a non-master line in another memory partition of the same set, it overwrites it with the displaced line. Note that this approach is cheaper than sending the displaced master line to a remote node: the bus is not used and the operation can be partially performed in the background.

A second issue to consider is the overhead of the "correcting" RAS cycle after a miss with no data fill, after the acceptance of a displaced line, and after a write back. While this RAS cycle is completely performed in the background, it increases the utilization of the memory. Our simulations, however, show that the resulting increase in memory busy time is modest and does not slow down the other requests noticeably. Note that, sometimes, this "correcting" RAS cycle can start very soon. For example, suppose that, in a memory access, after 10 ns, one of the memory partitions discovers that its active window is in the right place while the other memory partitions find that theirs are not. The former memory partition proceeds to check the tag, while the latter memory partitions start a RAS cycle to reposition their active windows. However, 10 ns later, the memory partition that had the right active window finds that it can supply the requested line. Right at this point, we can issue the "correcting" RAS cycles to the memory partitions that are in the process of moving their active windows. In the meantime, the bus is busy with the data transfer.

Finally, a third issue has to do with accesses to invalidated data. Our experiments show that such accesses often have much spatial locality. Consider, for example, *Mp3d*'s cells [15]. These are small objects that account for many attraction memory accesses. The fields in each cell are accessed in sequence, which should be good for hitting in active windows. However, cells are heavily shared and, therefore, an access to the attraction memory often finds the line marked invalid. If every time that this happened we had to take a window miss and spend a total of 80 ns searching all the other memory partitions in vain, some applications would run slower. Instead, we want our window algorithm to exploit this spatial locality. Consequently then, when, after the first 20 ns, an attraction memory access finds the requested line in an active window and marked invalid, we consider that a window hit has occurred and that the requested data is not in this attraction memory. Therefore, we complete the whole access in 20 ns and we can start a remote request. In this way, we can exploit the spatial locality present in invalidated data and complete all these memory accesses very fast. Of course, this requires that a set never keep, in two different memory partitions, a valid and an invalid copy of the same line. To ensure this, the insertion of a line into the attraction memory after a miss needs to be handled with some care: if there already is an invalid copy of the line, it has to be eliminated or overwritten.

All these changes to the line insertion and replacement algorithm certainly complicate the system. However, all our modifications only require changes to the state machine of the memory controller. We modify neither the DRAM data path nor the memory read and write primitives. In this sense, we do not complicate the memory interface. In addition, the traditional line replacement scheme that we compare against ours is already quite sophisticated. The traditional scheme performs random selection among tags in state invalid, then LRU selection among tags in state shared non-master, and then LRU selection among tags in state master.

4 Evaluation Approach

To evaluate these optimizations, we use detailed simulations. In this section, we describe the architecture modeled, the simulation environment used, and the applications run.

4.1 Architecture Modeled

We model a 32-processor Flat COMA multiprocessor with the nodes connected in a 16-bit wide 4x8 mesh. Each node has a 200-MHz processor, a 4-Kbyte primary cache, a 16-Kbyte secondary cache, a fraction of the distributed attraction memory and directory, and a network controller (Figure 4). The caches are direct-mapped and have 32-byte lines. We selected these small cache sizes because the working sets of the applications studied here are small. Using real-sized caches for these ap-

Table 1: Applications analyzed.

Application	Set	What It Does
Barnes	Splash	Simulates the evolution of galaxies using 1024 bodies for 4 steps.
Cholesky	Splash	Cholesky factorization of the bcsstk14.0 sparse matrix .
Mp3d	Splash	Simulates a rarefied hypersonic flow with 50,000 molecules for 20 steps.
Ocean	Splash	Simulates eddy currents in an ocean basin using a grid size of 98x98.
Dyfesm	Perfect Suite	Performs an analysis of symmetric anisotropic structures.
Qcd	Perfect Suite	Solves lattice gauge problems.
Flo52	Perfect Suite	Performs an analysis of 2-D transonic airflow.
Arc2d	Perfect Suite	Solves a 2-D fluid dynamics problem.

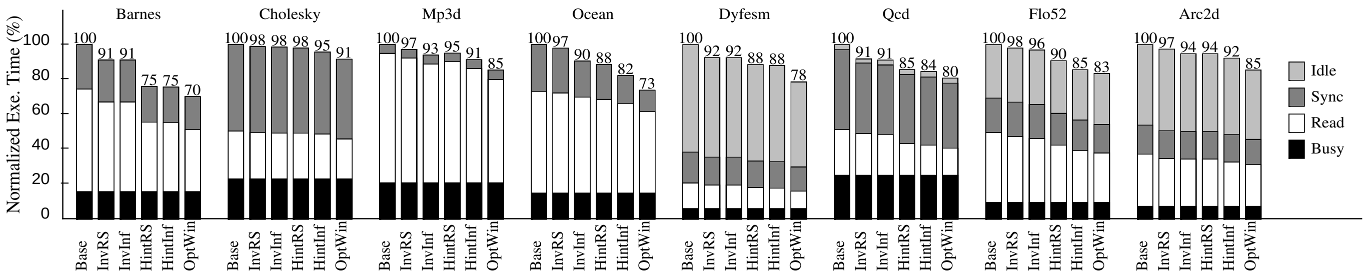


Figure 5: Execution time for different architectures. For each application, the bars are normalized to *Base*.

plications would produce unrealistic results. The bus between the secondary cache and the CPU runs at 66 MHz and is 64-bit wide, while the bus between the attraction memory and the secondary cache runs at 33 MHz and is 128-bit wide. The attraction memory is 4-way set-associative. Processors stall on read misses and use release consistency. In most cases, we run the applications under 0.8 memory pressure.

We use a DASH-like cache coherence protocol [11] with COMA extensions. Because of the small working sets of the applications, instead of using large active windows, we use 2-Kbyte active windows. In later experiments, we use even smaller windows (256 bytes). The small size prevents the active window from caching an unrealistically large fraction of the applications' data. In the baseline architecture, when a line needs to be inserted in a set in the attraction memory, we try to overwrite an invalid line first. If there are no invalid lines, we overwrite the LRU line in state non-master. Finally, if all lines are in state master, we select the LRU line for replacement. When a master line is displaced from the attraction memory of node i and there are no sharers of the line, then the line is sent to its home node. If node i is already the home, then the line is sent to node $i+1$. A node always accepts a line.

For the experiments with IVCs, the IVC is direct-mapped and, like the example of Section 3.1, contains as many entries as the secondary cache (512 entries). Its total size is slightly over 1 Kbyte, with or without hints. The latencies of round trip transactions without contention are shown in Figure 4. In the figure, the 2-hop and 3-hop transactions assume that the data is in the attraction memory. The latency of these transactions includes two attraction memory accesses. Finally, each network hop takes 30 cycles. In our simulations, we model all the contention for resources.

4.2 Simulation Environment and Applications

The architecture simulator described is connected to two trace generation systems, namely Tangolite [3] and EPG-sim [13], to form two execution-driven simulation systems. Tangolite is used for the Splash applications [15], while EPG-sim is used for the Perfect Suite applications [1]. The resulting tightly-coupled execution-driven simulation systems ensure correct interleaving of memory accesses. To make the evaluation as representative as possible, we evaluate the architecture with eight parallel applications from the Splash and Perfect Suite sets. The applications are shown in Table 1 and are all run with 32 processes. While the Splash applications are hand-parallelized, the Perfect Suite applications are parallelized using a parallelizing compiler and later by hand.

5 Evaluation

We now use detailed architecture simulations to evaluate the impact of IVCs, IVCs with hints, and optimized window locality in page-mode DRAMs.

5.1 Invalidation Caches

To evaluate the impact of IVCs, we compare the execution time of the applications on the baseline architecture of Section 4.1 (called *Base*) to the execution time on two different architectures: *InvRS* and *InvInf*. These architectures are like *Base* plus IVCs: *InvRS* has 512-entry direct-mapped IVCs while *InvInf* has infinite IVCs. The measured execution times are shown in Figure 5. In the figure, the leftmost three bars in each application correspond to *Base*, *InvRS*, and *InvInf*. The execution time is normalized to *Base* and divided into execution of instructions (*Busy*), processor stall due to read misses in the memory hierarchy (*Read*), stall due to synchronization and write buffer overflow (*Sync*) and, for the Perfect Suite applications, stall when another processor is executing the serial section of the application (*Idle*).

Overall, the impact of IVCs is modest: applications run, on average, 5% faster in *InvRS* than in *Base*. It could be suspected that IVCs do not have a higher impact mainly because their entries suffer frequent replacements. The impact of replacements, however, is small. This can be seen by examining the *InvInf* bars, which correspond to infinite IVCs: *InvInf* is, on average, about 7% faster than *Base*.

To understand the modest impact of IVCs, we recall that IVCs directly address only the *Read* time. Figure 6 decomposes the *Read* time of Figure 5 into its components: stall due

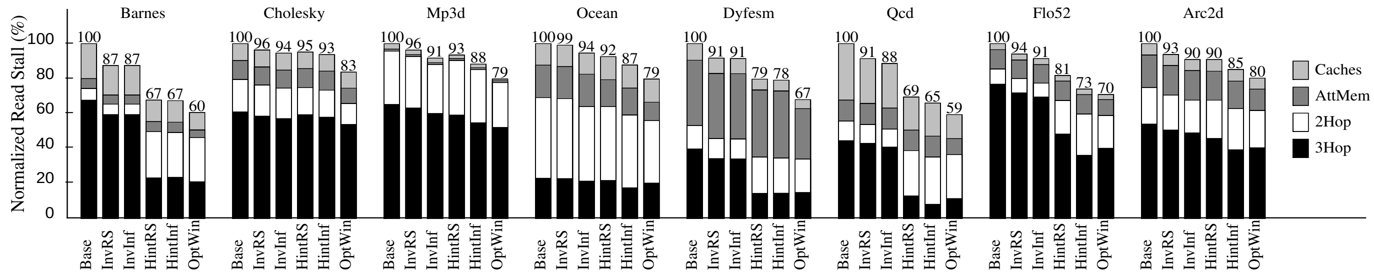


Figure 6: Processor stall time due to read misses for different architectures. For each application, the bars are normalized to *Base*.

to reads satisfied by the caches (*Caches*), stall due to reads satisfied by the local attraction memory (*AttMem*), and stall due to reads satisfied by remote nodes via two or three hops (*2Hop* and *3Hop* respectively). We focus on the three leftmost bars of each application, namely *Base*, *InvRS*, and *InvInf*. As in Figure 5, the bars are normalized to *Base*.

As discussed in Section 2.1, IVCs can at most eliminate 15% of the *2Hop* and *3Hop* time. We can see from Figure 6 that IVCs get close to this 15% reduction in *Barnes*, *Dyfesm*, *Flo52* and, to a lesser extent, *Arc2d*. However, they are unable to decrease the *2Hop* and *3Hop* time much in the rest of the applications. This means that, in the latter applications, IVCs are often unable to prevent secondary cache misses from performing unsuccessful attraction memory accesses. This may occur in three situations. The first one is at start-up time, when memory lines are referenced for the first time. The second one may occur for lines that have been removed from the attraction memory by a real invalidation: the IVC recorded the event in an entry that was later displaced from the IVC. The third situation may occur for lines that have been removed from the attraction memory by a self invalidation: either the IVC recorded the event in an entry that was later displaced or, in the case of the self invalidation of a non-master line, the IVC did not even record the event. These three situations we call *MissMissCold*, *MissMissInv*, and *MissMissConf* respectively, where *MissMiss* stands for a miss in both IVC and attraction memory.

To see how often these situations occur, Figure 7 shows the frequency of the different possible outcomes of 2- or 3-hop transactions. We focus on the two leftmost bars in each application. Transactions can either hit in the IVC (*HitSucc* plus *HitFail* in the figure) and therefore skip an unnecessary attraction memory access, or miss in the IVC (*MissMissCold*, *MissMissInv*, and *MissMissConf*) and perform an unsuccessful attraction memory access. In the figure, the number of 2- and 3-hop transactions is normalized to 100 for each architecture. The difference between *HitSucc* and *HitFail* is not relevant to *InvRS* and *InvInf*. Therefore, to simplify the charts, we lump the two categories in these two bars into *HitFail* and refer to it as *Hitxxx*.

The figure shows the effectiveness of the IVC at intercepting remote transactions and, if the IVC is not successful, why. For *Barnes* and *Dyfesm*, the IVC is very effective: the *Hitxxx* category is the main contributor to the *InvRS* bar. These are the applications where the 2- and 3-hop transaction time in Figure 6 decreases the most. For four other applications, however, namely *Cholesky*, *Mp3d*, *Ocean*, and *Arc2d*, the IVC is ineffective: relatively few remote transactions are intercepted, as can be seen from the low contribution of the *Hitxxx* category to the *InvRS* bar. To understand why, we compare the *InvRS* bars to the *InvInf* bars. In *Cholesky* and, to a lesser extent *Arc2d*, *MissMissCold* dominates. A large fraction of the remote accesses occurred during the start-up time. Therefore,

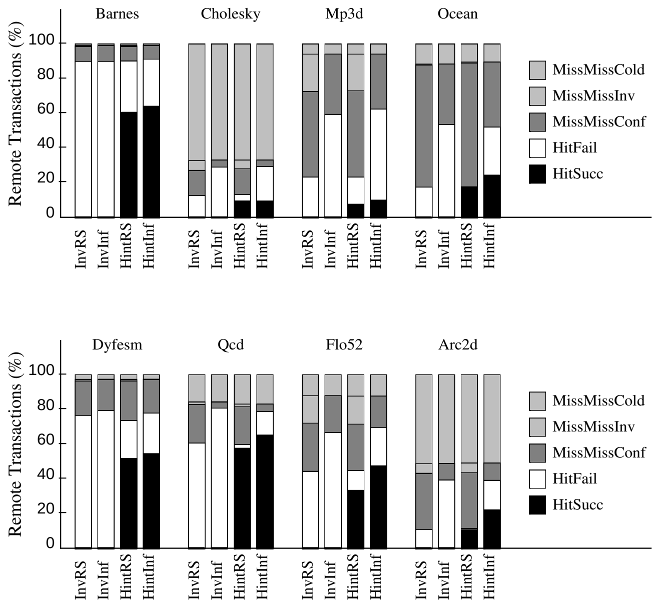


Figure 7: Breakdown of the outcome of remote transactions. For each application and architecture, the bars are normalized to 100 transactions.

little can be done to increase the effectiveness of the IVC. Even an infinite IVC (*InvInf*) does not help much. For *Ocean* and *Mp3d*, *MissMissConf* and *MissMissInv* accesses dominate in *InvRS*, and a good fraction of them disappear for *InvInf*. This means that the IVC is too small for *Ocean*'s and *Mp3d*'s working set: the IVC suffers conflicts that disappear for an infinite IVC. A larger IVC would help. Note that an infinite IVC still induces *MissMissConf* transactions. This is because the IVC does not record the address of non-master lines that are self-invalidated in the attraction memory. Finally, for the remaining applications, namely *Qcd* and *Flo52*, the IVC has a modest effectiveness. It intercepts a good fraction of the remote transactions but it still induces some *MissMissInv* and *MissMissConf* transactions. Larger IVCs would help a bit.

Overall, while the effectiveness of IVCs for a particular application is difficult to predict a priori, we see that small- to medium-sized IVCs are nearly as effective as infinite IVCs. This is probably the case in general even accounting for the fact that our applications are small. Unfortunately, IVCs increase the performance of the machine by only 5% on average. To increase their impact, we now reuse the IVCs for the next optimization.

5.2 Invalidation Caches with Hints

To evaluate the impact of hints, we examine two architectures, namely *HintRS* and *HintInf*. *HintRS* is *InvRS* plus coherence and conflict hints, while *HintInf* is *InvInf* plus coherence and conflict hints. For the coherence hints, we use invalid hints [2, 4]. The execution time of the applications on the *HintRS* and *HintInf* architectures is shown in bars 4-5 of Figure 5. Overall, we see that the impact of IVCs with hints is larger: applications run, on average, 11% faster than in *Base*. As before, infinite IVCs perform only slightly better than finites ones: *HintInf* is 3% faster than *HintRS*. The impact of *HintRS* is largest in *Barnes*, where *HintRS* is 18% faster than *InvRS*. The impact is more modest in *Ocean*, *Dyfesm*, *Qcd*, and *Flo52*. Finally, for the other three applications, the impact is minimal.

HintRS largely attempts to reduce the *Read* time. To see how effective it is in doing so, we examine the *Read* time (Figure 6). We focus on the fourth bar, *HintRS*, and compare it to the second one, *InvRS*. If IVCs with hints are successful, the *3Hop* time in *InvRS* should transform into smaller *2Hop* time in *HintRS*. From the figure, we see that hints work well for *Barnes*, *Dyfesm*, *Qcd* and *Flo52*. For the first three of these applications, it can be seen from the charts that about 60-70% of the *3Hop* transactions become *2Hop* transactions, while for *Flo52* the fraction is 30%. However, for the other four applications, namely *Cholesky*, *Mp3d*, *Ocean* and *Arc2d*, hints do not manage to reduce the *3Hop* time much. This is consistent with the results of Bjorkman *et al* [2], which we discuss in Section 6.

To understand these results, we examine the outcome of 2- and 3-hop transactions (Figure 7). We focus on the *HintRS* bars. In these bars and those to their right, the *HitSucc* and *HitFail* categories denote two types of IVC hits: in *HitSucc* the hint is correct, while in *HitFail* the hint is wrong. In the three applications for which Figure 6 showed the best results (*Barnes*, *Dyfesm*, and *Qcd*), the IVC intercepts many transactions and supplies a correct hint most of the time. Indeed, *HitSucc* accounts for 50-60% of the *HintRS* bars in these three applications. It is interesting, however, to note that *HitFail* transactions are still quite frequent in these applications. Better hints could make a difference.

In three of the applications for which Figure 6 showed little change (*Cholesky*, *Ocean* and *Arc2d*), the low impact of IVCs with hints is not due to incorrect hints. Instead, it is the result of IVCs not intercepting transactions: *HitSucc* is much larger than *HitFail* in *HintRS* but *HitSucc*+*HitFail* accounts for little. In these applications and for these IVCs, hints cannot help much more.

Finally, for *Mp3d*, the IVC intercepts a relatively small fraction of the transactions (Figure 7). In addition, the hints are mostly incorrect. The low accuracy of the hints in *Mp3d* is due to the migratory behavior of its data, as pointed out by Bjorkman *et al* [2]. *Mp3d*, therefore, suffers in part from poor hints.

To understand why larger IVCs do not accomplish much more, we compare *HintRS* to *HintInf* in Figure 7. From the figure, we see that infinite IVCs have a significant effect on *Mp3d*, *Ocean* and, to a lesser extent, *Flo52* and *Arc2d*. In these applications, the contribution of *HitSucc*+*HitFail* increases. Most of the increase, however, corresponds to *HitFail*. This means that, while a very large size allows the IVCs to intercept more requests, the hints often become obsolete.

To summarize, IVCs with coherence and conflict hints speed up our eight applications by an average of 11%. Some applications are sped up much more. For example, *Barnes* runs 25% faster. The cost of this optimization comes largely from the changes made to the cache coherence protocol to support the hint transactions. The cost of the IVC itself is small. In the analysis, we have found out that the impact of this optimization is not uniform across applications. Finally, we have also found out that little further gains can be achieved by simply increasing the size of IVCs. Instead, it may be necessary to find better hints, especially for large IVCs.

5.3 Improving Window Locality

To evaluate the impact of optimized window locality in page-mode DRAMs, we first evaluate the support described in Section 3.3.1, then examine an upper bound on the performance gains that can be achieved, and then consider several architectural variations.

5.3.1 Basic Evaluation

We start the evaluation by adding the support described in Section 3.3.1 to the *HintRS* architecture. The optimized scheme we call *OptWin* for optimized windows. Note that *HintRS*, like all the previous systems, already uses (and its performance benefits from) windows. However, their use is not optimized. The execution time of the applications under *OptWin* is shown in Figure 5 as the rightmost bar for each application. In the figure, if we compare the *OptWin* bars to the *HintRS* bars (two bars to the left), we see that all applications benefit from this optimization. *OptWin*, therefore, is a robust optimization. The execution time decreases by an average of 10%. With this optimization, the applications now run, on average, 20% faster than they used to in *Base*.

This optimization tries to reduce the *Read* time. To understand its impact better, we examine the breakdown of the *Read* time (Figure 6). In the figure, the *OptWin* bars are the rightmost ones in each application. Their height is consistently lower than the height of the *HintRS* bars (two bars to the left). Furthermore, all the categories that include memory accesses, namely *AttMem*, *2Hop*, and *3Hop*, decrease. Of course, *AttMem* has the largest relative reduction because the memory access itself accounts for a large fraction of the transaction time. For example, in *Ocean* and *Dyfesm*, *AttMem* decreases by nearly 30%, which is close to the upper bound (Section 2.3). However, except in *Dyfesm*, the contribution of *AttMem* tends to be small. The other relevant categories, namely *2Hop* and *3Hop*, account for a larger fraction of time but exhibit relatively smaller reductions.

To gain more insight, we measure the window hit rate of attraction memory reads. We consider read requests only because the other requests do not stall the processor. Figure 8 shows the window hit rate for the different applications for the unoptimized system (*HintRS*) and the optimized one (*OptWin*). The third curve (*UppBnd*) will be considered later. The figure also includes three horizontal lines showing the average of each curve over all the applications. The figure shows that the window hit rate for *HintRS* ranges from 10% to 50%. It has a wide variation across applications. With our window optimization, the hit rate increases for all applications. The resulting hit rate (*OptWin*) ranges from 30% to 90%. Interestingly, the *OptWin* curve has a shape similar to the *HintRS* curve. Overall, the optimization has raised the average window hit rate 24 points, from 30% to 54%. Such a change is large enough to deliver the 10% execution speedup discussed above. We also note that the average values of the unoptimized and optimized window hit rates are above 16.7%, the break-even point for the window-based memory to be competitive (Section 2.3).

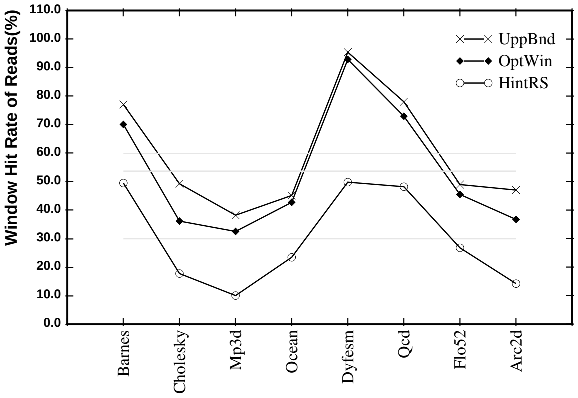


Figure 8: Window hit rates for different scenarios.

5.3.2 Upper Bound Analysis

Figure 8 shows that several applications still have a low window hit rate after the optimization. For example, *Mp3d*, *Cholesky*, and *Arc2d* have a window hit rate of about 35%. Taking an average for all the applications, about 1 out of 2 attraction memory reads still miss in active windows. To understand why, we classify the active window misses into two types. The first type of active window miss occurs when the address that the processor requests is not in the attraction memory. The second one occurs when the address is in the attraction memory but is not in any of the active windows. To a large extent, our optimization can only affect the second type of misses.

To determine the effect of this second type of misses, we simulate a hypothetical machine where, if the attraction memory contains the requested address, we record the access as an active window hit and supply the line with the speed of a window hit. Of course, the active window is moved appropriately. This scenario is close to the best possible performance of any window algorithm. We call such a scenario *UppBnd*, for upper bound. The resulting window hit rate is shown in the *UppBnd* curve of Figure 8. From the figure, we see that the *UppBnd* curve is very close to the *OptWin* curve for all the applications. On average, the window hit rate for *UppBnd* is only six points higher than the one for *OptWin*. Furthermore, simulations of the *UppBnd* machine show that the applications run, on average, only 4% faster than on the *OptWin* machine. Clearly, therefore, there is very little more that can be done by tinkering with the window algorithm.

To improve the active window hit rate significantly, we need to eliminate active window misses of the first type, which occur when the requested address is not in memory. These misses are caused by start-up effects and conflicts in the attraction memory. They are not caused by coherence activity because the reception of an invalidation message does not clear the address bits of the tags (Section 3.3.2). Conflict misses can be caused by poor locality of the references to the active objects or by too many active objects that thrash the active windows. In the latter case, it can help to increase the associativity of the attraction memory. In the next section, we consider this and other architectural variations.

5.3.3 Architectural Variations

To assess the window locality optimization better, we evaluate it under several different architectural settings. In this section, we present a brief summary of the evaluation.

Higher Attraction Memory Associativity. We evaluate the window locality optimization in the more expensive case of an 8-way set-associative attraction memory. As expected, *OptWin* is faster now than with 4-way set-associative attraction memories. However, *HintRS* is also faster than before because the attraction memory suffers fewer conflicts. The result is that the relative impact of the window optimization decreases. It reduces the execution time an average of 8% instead of 10%.

Smaller Window Sizes. All the previous experiments have used 2-Kbyte windows. While the size of realistic windows is likely to be much larger (Section 2.3), we have chosen this small size to compensate for the small data size of the applications studied. To compensate more, we also perform experiments with windows as small as 256 bytes. With such small windows, both *OptWin* and *HintRS* are slower. However, the relative gains of *OptWin* over *HintRS* are approximately the same as in the 2-Kbyte windows. The optimization, therefore, is robust.

Window Optimization Only. Finally, we apply the window locality optimization directly to the *Base* system. Therefore, we do not use IVCs. The result is a reduction in execution time that is slightly larger than if we applied the optimization to *HintRS*. However, if we measure the reduction in percentages, the two changes have practically the same magnitude. Therefore, we conclude that the effect of the IVC plus the hints and the effect of the window locality optimization are largely orthogonal.

To conclude, the data indicates that the window locality optimization presented is promising. While the 10% execution time reductions that we observe are modest, they are remarkably robust across applications and architectures. Furthermore, we expect that the impact will be higher as processor speeds increase or memories become more integrated with processors, possibly in the same chip. While this optimization requires changes to the memory controller, it does not affect the DRAM data paths or the memory read and write primitives.

6 Related Work

There are several pieces of related work. Gupta *et al* [4] and Bjorkman *et al* [2] proposed the use of hints to transform three-hop transactions into two-hop ones. Furthermore, Bjorkman *et al* [2] proposed sending the request to both the node that is the home of the line and the hint node. While we have used their ideas, our approach differs in two ways. First, we use conflict hints in addition to coherence hints; the other authors used coherence hints only. Secondly, we store the hints in the IVC, therefore skipping unnecessary attraction memory accesses and saving space; the other authors store the hints in memory. Our results are obtained using a superset of the applications used by Bjorkman *et al* except for *Water*, and show a higher speedup than theirs. Part of the reason why the speedup reported in this paper is higher is that both conflict and coherence hints contribute to it. In addition, since we simulate faster processors, the memory latencies in our system are higher than in theirs. This increases the relative impact of the hints.

Muller *et al* [12] propose an optimization for the attraction memory of the DDM COMA machine [6]. The idea is to use a skewed-associative attraction memory to decrease the chances that hot memory lines conflict with each other in several sets. Their scheme and our window scheme address different problems: their scheme increases the number of hits, while our scheme speeds up accesses that already hit.

Hagersten [5] discusses a scheme that was considered in the design of the DDM COMA machine [6]. The idea is to keep a special register for each set in a two-way set-associative attraction memory. The register points to the most recently-used partition in the set. When a set is accessed, the partition pointed to by the corresponding register is immediately checked. If that partition hits, the access is satisfied with low latency. Our window scheme is more elaborate than this. Finally, Hsu and Smith [7] studied cached DRAM organizations for vector supercomputers. Their work is different in that it is focused on machines without caches running highly vectorized codes.

7 Summary

Scalable Flat COMA multiprocessors are designed for reduced memory access latencies while minimizing programmer and operating system involvement. Unfortunately, part of the latency of memory accesses is superfluous. In particular, reads often perform unnecessary attraction memory accesses, require too many network hops, or perform necessary attraction memory accesses inefficiently. In this paper, we have presented two techniques to address these problems, namely the IVC augmented with coherence and conflict hints and the window locality optimization for page-mode DRAMs.

The effect of the window locality optimization and the effect of the IVC augmented with hints are largely independent. The window locality optimization speeds up eight 32-processor applications by 10%. This optimization is robust and has a modest cost confined to memory controller modifications. For these reasons, we recommend it. IVCs augmented with hints speed up the applications by 11%. However, the cost of this optimization is higher because it requires changes to the cache coherence protocol. In addition, its impact is less uniform across applications. For these reasons, it is less attractive, although it can be argued that it is still useful. Overall, the two optimizations combined speed up the eight applications by an average of 20%.

References

[1] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[2] M. Bjorkman, F. Dahlgren, and P. Stenstrom. Using Hints to Reduce the Read Miss Penalty for Flat COMA Protocols. In *Proceedings of the 28th Annual Hawaii International Conference of System Sciences*, pages 242–251, January 1995.

[3] S. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[4] A. Gupta, T. Joe, and P. Stenstrom. Performance Limitations of Cache-Coherent NUMA and COMA Multiprocessors and the Flat-COMA Solution. Technical Report CSL-TR-92-524, Computer Systems Laboratory, Stanford University, October 1992.

[5] E. Hagersten. Toward Scalable Cache Only Memory Architectures. Ph.D. Dissertation, The Royal Institute of Technology, Stockholm, Sweden, October 1992.

[6] E. Hagersten, A. Landin, and S. Haridi. DDM – a Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[7] W. C. Hsu and J. E. Smith. Performance of Cached DRAM Organizations in Vector Supercomputers. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 327–336, May 1993.

[8] T. Joe and J. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, April 1994.

[9] F. Jones et al. A New Era of Fast Dynamic RAMs. *IEEE Spectrum*, pages 43–49, October 1992.

[10] Kendall Square Research. *KSR1 Technical Summary*. Waltham, MA, 1992.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[12] H. Muller, P. Stallard, and D. Warren. The Application of Skewed-Associative Memories to Cache Only Memory Architectures. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages I150–I154, August 1995.

[13] D. K. Poulsen and P.-C. Yew. Execution Driven Tools for Parallel Simulation of Parallel Architectures and Applications. In *Proceedings of Supercomputing '93*, pages 860–869, November 1993.

[14] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[15] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[16] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.