Adaptive Cache Coherency for Detecting Migratory Shared Data

Alan L. Cox
Department of Computer Science
Rice University

Robert J. Fowler
Department of Computer Science
University of Rochester

Abstract

Parallel programs exhibit a small number of distinct data-sharing patterns. A common data-sharing pattern, *migratory access*, is characterized by exclusive read and write access by one processor at a time to a shared datum. We describe a family of adaptive cache coherency protocols that dynamically identify migratory shared data in order to reduce the cost of moving them. The protocols use a standard memory model and processor-cache interface. They do not require any compile-time or run-time software support. We describe implementations for bus-based multiprocessors and for shared-memory multiprocessors that use directory-based caches. These implementations are simple and would not significantly increase hardware cost. We use trace- and execution-driven simulation to compare the performance of the adaptive protocols to standard write-invalidate protocols. These simulations indicate that, compared to conventional protocols, the use of the adaptive protocol can almost halve the number of inter-node messages on some applications. Since cache coherency traffic represents a larger part of the total communication as cache size increases, the relative benefit of using the adaptive protocol also increases.

1 Introduction

Parallel programs exhibit a small number of distinct data-sharing patterns [1, 23]. In one of the most common patterns, *migratory access*, the life of a shared datum consists of a sequence of time intervals. During each interval a single processor both reads and writes the datum. In successive intervals the migratory datum is accessed by different processors. Several common programming techniques give rise to the migratory access pattern, including the use of read/write data structures protected by locks or monitors and the use of shared task queues to distribute work among the nodes of a multiprocessor.

We present and evaluate both snooping and directory-based cache coherence protocols that automatically classify cached memory blocks as *migratory* or *other* and that adaptively switch between a sub-protocol optimized for migratory access and one appropriate for other data-sharing patterns. Since the optimized sub-protocol can halve the number of coherency transactions needed to manage migratory data and since migratory data are very common, the use of these adaptive protocols yields a substantial performance improvement for many programs. The magnitude of the improvement depends, of course, on the fraction of total execution time spent waiting for memory coherency operations.

The adaptive caching protocols presented in this paper are distinguished from previous work in three ways. The migratory access pattern is identified using simple on-line algorithms that react quickly to changes in data-sharing pattern. The mechanisms and policies are simple enough to build into hardware cache controllers without a large cost or complexity increase. The adaptive protocols do not change the memory model seen by the programmer and compiler, nor do they add any special requirements to the processor-cache interface. Thus, our adaptive protocols are transparently compatible with existing language, compiler, and processor designs.

In general, multiprocessor cache coherence protocols use either a "write-update" strategy or a "write-invalidate" strategy. The write-update strategy entails interprocessor communication on every write operation to shared data. In contrast, the write-invalidate strategy entails communication only on the first write operation. In this case, the purpose of the communication is to request all other caches to invalidate their copies of the data. Until the data is accessed by another processor, subsequent write operations can be completed locally, without communication. Since migratory data is characterized by a temporal clustering of operations on a single processor, write-invalidate outperforms write-update for this data-sharing pattern. Furthermore, it is difficult to build an efficient implementation of write-update for non-broadcast-based interconnects. Therefore, write-update is generally regarded as unsuitable for use in a scaleable multiprocessor. For these reasons, our work starts from a write-invalidate protocol.

On a read miss, most write-invalidate protocols create a new copy of the block in the cache of the the processor that initiated the miss. Known as "replicate-on-read-miss", this policy is used because it allows read-shared blocks to be replicated at all of the processors that share them. Unfortunately, if a migratory block is read before it is written, the replicate-on-read-miss policy uses two separate inter-cache operations to move the block from one cache to another. For example, assume the block is initially dirty in processor P_i's cache. A read miss by P_j copies the block to P_j's cache and changes its state in P_i's cache to read-only. Later, when P_j writes to the block it is necessary to invalidate the copy in P_i's cache. In contrast, if the cache used a "migrate-on-read-miss" policy, the copying of the block to P_j and the

This work is supported in part by NSF Grant CCR-9211004.

invalidation of P_i's copy could be done in one transaction, thereby halving the number of inter-cache operations used to move the migratory block. If the first access from P_j is a write, both policies incur the same amount of overhead since they handle write-misses the same way.

On the other hand, a pure migrate-on-read-miss policy performs poorly for other data-sharing patterns. For example, if the data are read-shared, the cache block will migrate on every read request rather than becoming replicated in the caches of the processors accessing it. In this case, the replicate-on-read-miss policy of standard coherency protocols is appropriate.

Since neither "pure" policy is ideal for both data-sharing patterns, we have developed a scheme in which the two policies co-exist. In Section 2, we present a family of simple cache coherency protocols that (1) efficiently and dynamically differentiate migratory access from other data-sharing patterns and (2) use this information to adaptively switch between the policies. In Section 3, we describe our methodology for evaluating and comparing these adaptive protocols to conventional protocols. Our results appear in Section 4. Briefly, we have found that the adaptive protocols can reduce the number of inter-node messages by almost half for some applications. In Section 5, we compare the adaptive protocols and our results to related work. Finally, in Section 6, we present our conclusions.

2 Identifying and Handling Migratory Data

We first address the issue of distinguishing migratory blocks from read-shared blocks. Under the replicate-on-read-miss policy, a cache block containing migratory data is transported from processor P_i to P_j in two stages.

1. Since the block is migratory by assumption, processor P_i has written it. The block is dirty in P_i's cache and is invalid in P_j's cache. Thus, P_j's first access to the block is a read that misses. To service the read miss, P_j's cache controller sends a read request for the the block. In response, P_i's cache controller provides the modified block, and changes the state of the block to "Shared", a non-exclusive state that permits the block to be read but not written. At the end of this stage, there are valid Shared copies of the block in both caches.

2. P_j's first attempt to write to the block finds it in the Shared state. The operation cannot complete until all other copies of the block have been invalidated, so P_j's cache controller sends an invalidation request. On receiving the request, processor P_i's cache invalidates its copy of the block. To ensure that write operations are serialized correctly, the write operation at P_j is not allowed to proceed until there is a guarantee that there are no other write requests that can precede this one. This guarantee requires either explicit acknowledgement messages or some mechanism, such as a shared bus, that automatically serializes operations.

If a block is currently managed using the replicate-on-read-miss policy, the adaptive protocols use the above pattern of cache coherency operations as evidence that the block is migratory. Specifically, a block appears to be migratory if, at the time of a write-hit to a block in a Shared state, (1) there are exactly two cached copies of the block, and (2) the processor currently requesting the invalidation operation is not the processor that requested the previous invalidation operation. An alternative statement of the second condition is that originator of the invalidation request has the more recently created copy of the block. A write-miss on a block for which there is a single cached copy can also be used as evidence that the block is migratory.

If a block is currently classified as migratory, the adaptive protocol expects that the block will be modified at every processor it visits. Thus, if the block is not modified before it moves to another processor, this is used as evidence that the block is not currently migratory.

All members of our family of adaptive protocols use these tests as evidence that a given block is migratory or not. In addition to the fundamental implementation differences between snooping and directory-based coherence protocols, family members differ from each other in three ways:

1. How quickly does the protocol adapt to changes in access pattern? Given the set of events used as evidence that a block is migratory or not, a protocol can reclassify a block immediately or it can introduce hysteresis by requiring several successive occurrences of these events.

2. How accurately does the protocol keep track of the classification of and the number of copies of each block? Bus-based snooping protocols and some directory-based protocols [15] do not retain any state for uncached blocks, while other directory protocols can track a block over the long term. Classification information can therefore be lost for blocks that become uncached. To eliminate extraneous communication, a protocol might not keep an accurate count of the number of copies of each block. Typically a cache is allowed to silently evict a clean block without communicating with other caches or with a directory manager.

3. Is the block initially classified as migratory, or non-migratory? This affects cold-start costs. Since many coherency protocols do not keep any state for uncached blocks, the initial classification decision can also affect the cost of managing any block that becomes uncached.

Capacity (and conflict) misses occur because real caches are finite. The choices expressed by the last two items above affect how well an adaptive protocol performs in the presence of such misses.

2.1 A Snooping, Bus-based Implementation

To implement an adaptive protocol on a bus-based multiprocessor, one could extend any of several well-known snoopy cache protocols based on write-invalidate [18]. In this example, we extend the common MESI protocol. In the base protocol, a cache entry can be in one of four states. Invalid (I) means that the block is not currently valid in this cache. Exclusive (E) means that this entry is the only cached copy of the block and that main memory is up to date. Dirty[1] (D) means that this is the only cached copy of the block and that the copy in main memory not current. Shared (S) means that this is one of multiple cached copies and that

[1]This state is usually called "Modified". We renamed it to allow us to use M to denote "Migratory" in the extended protocol.

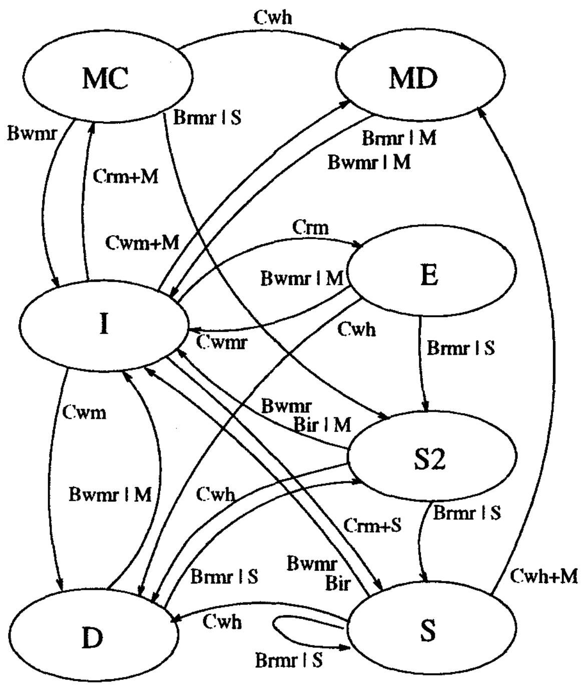

Figure 1: Transition diagram for an adaptive snooping protocol. Each transition is annotated by the event that triggers the transition and by any protocol-specific signals that are asserted on the bus as part of the transaction. The cache itself generates read miss (Crm), write miss (Cwm), and write hit (Cwh) events. Read miss requests (Brmr), write miss requests (Bwmr), and invalidation requests (Bir) are requests coming from other caches on the bus. For example, "Crm+M" labels a transition that occurs when the local cache has a read miss and "Migratory" has been asserted in the response. "Brmr | S" labels a transition triggered by a read miss request. In response, this cache asserts "Shared" on the bus.

memory is up to date. We assume that main memory also snoops, so it is updated on any coherency operations that contain data as well as on any explicit writeback operations. In both the base protocol and in the adaptive extension, any data transferred in a transaction comes from main memory unless the data was dirty in some cache at the start of a transaction.

As part of the bus transaction used to respond to a read miss, the base protocol has a mechanism for a cache to signal that a data block is Shared. The adaptive extension also needs to be able to signal that a block is Migratory in the bus transactions used to respond to read misses, write misses, and invalidation requests. We describe the protocol as though an additional *Migratory* line is added to the bus. It may be possible, however, to encode the new bus transaction type without adding any additional wires.

Figure 1 shows the transition diagram of the adaptive protocol. Figure 2 presents it in tabular form. The lower half of the diagram uses a replicate-on-read-miss policy. It corresponds to a slightly modified version of the base protocol. The main new feature is that the Shared state has been split. A new Shared-2 (S2) state means that there are no more than two cached copies of the block. This condition is guaranteed because the only transitions into S2 are from states in which there is exactly one cached copy. Thus, transitions from the Exclusive and Dirty states in response to read miss requests from the bus (Brmr) go to S2. This implies that when there are two cached copies of a block that only the older cache entry can be in the S2 state. On subsequent read miss requests, the block enters and stays in the Shared state.

The upper part of Figure 1 corresponds to the subprotocol for handling migratory data. A migratory block can be in either the Migratory-Clean (MC) or Migratory-Dirty (MD) states, depending on whether or not it has been modified at its current location. The transitions among the Invalid, Migratory-Clean, and Migratory-Dirty states implement the migrate-on-read-miss policy.

There are two ways that the protocol can shift from the replicate-on-read-miss policy to the migrate-on-read-miss policy. If processor P_j has a read miss on a block that is currently cached only at P_i in either the Exclusive or Dirty state, then P_i's cache controller changes the state of the local cache entry to Shared-2 and asserts on the bus that the block is Shared. Since Shared is asserted in the the response, the cache entry at P_j will be created in the Shared state. A subsequent write hit at P_j issues an invalidation request on the bus (Bir). In response, P_i's controller invalidates its local entry and asserts Migratory in the acknowledgement. At P_j the protocol takes the transition labelled "Cwh+M" in Figure 1 from Shared to Modified-Dirty.

This aggressive protocol also switches from the replicate-on-read-miss policy to migrate-on-read-miss if a processor has a write miss to a block with a single cached copy in either the Exclusive or Dirty states.

The switch from migrate-on-read-miss to replicate-on-read-miss occurs when a cache with a Migratory-Clean entry for the block receives any miss request from the bus.

The snooping protocol as we have described it uses replicate-on-read-miss as the initial policy for each block. It also switches between policies with no hysteresis. A possible variation is to use migrate-on-read-miss as the initial policy. If this change is made, then the Exclusive state would no longer have any in-transitions and could thus be eliminated as a "dead" state.

Adding hysteresis to the adaptive protocol would multiplicatively increase the number of states, complicating its representation as either a state machine or a transition table. In practice, however, the additional states would be efficiently encoded as a small (one or two bits) counter field.

2.2 A Directory-based Implementation

In this description of a directory-based protocol, we assume that the underlying system is a distributed shared-memory multiprocessor with coherent caches, also known as a *cache-coherent non-uniform memory access* (CC-NUMA) multiprocessor. This type of architecture is based on a collection of *nodes* joined by a logically complete point-to-point network. Each node is a processing element consisting of a processor with its local cache, a memory module, and a memory controller that handles all inter-node memory references going into or out of the node. Each memory controller also maintains directory entries and implements the coherency protocol for all of the blocks in main memory at that node. This node is called the *home node* for those blocks.

Adding an adaptive protocol to an existing directory-based protocol increases the size of each directory entry.

Transitions on Local Cache Events				
State	Event	Request	Reply	New State
I	Crm	Brmr	$\neg M \wedge \neg S$	E
	Crm	Brmr	M	MC
	Crm	Brmr	S	S
	Cwm	Bwmr	$\neg M$	D
	Cwm	Bwmr	M	MD
E	Cwh			D
S2	Cwh	Bir		D
S	Cwh	Bir	$\neg M$	D
	Cwh	Bir	M	MD
MC	Cwh			MD

Transitions on Bus Requests				
State	Request	New State	Assert	Data
E	Brmr	S2	S	
	Bwmr	I	M	
Dirty	Brmr	S2	S	Provide
	Bwmr	I	M	Provide
S2	Brmr	S	S	
	Bwmr	I		
	Bir	I	M	
S	Brmr	S	S	
	Bwmr	I		
	Bir	I		
MC	Brmr	S2	S	
	Bwmr	I		
MD	Brmr	I	M	Provide
	Bwmr	I	M	Provide

Figure 2: State transition tables for the adaptive snooping protocol.

The amount of extra storage depends on both the design of the original protocol and the properties of the particular adaptive policy chosen. For example, if the *copy set*, the data structure that lists the nodes at which a block is cached, allows the memory controller to determine the order in which the cached copies of a block were created, then extra storage to hold the last invalidator is not required. Furthermore, although it is only necessary to add one new state to the protocol to indicate that a block is migratory, it may be convenient to add additional states. In particular, while one could test whether the cardinality (*copy set*) is equal to two as an approximation to the decision rule, this will classify a block as migratory even though there used to be three copies, but one copy was dropped from some cache. To more accurately capture the notion of migratory access, the state encodes the number of copies that have been created rather than the number of copies that currently exist.

Adding hysteresis to the protocol will further expand the size of a directory entry.

Figure 3 presents a pseudo-code implementation of one possible implementation. (Only those parts of the protocol directly related to the adaptive protocol are illustrated.) This implementation initially uses replicate-on-read-miss for each block and requires two successive events to classify a block as migratory, while the switch in the other direction occurs immediately. In this example, the identity of the last invalidator is represented explicitly. Furthermore, this version retains the classification of a block as migratory or other even when the block is not cached. The state of a block is

```
read miss:
  switch ( state )
      case UNCACHED:
          state ← ONE COPY ;
      case UNCACHED/MIGRATORY:
          state ← ONE COPY/MIGRATORY ;
      case ONE COPY:
          state ← TWO COPIES ;
      case ONE COPY/MIGRATORY:
          if dirty = FALSE then
              state ← TWO COPIES ;
              one migration ← FALSE ;
      case TWO COPIES:
          state ← THREE OR MORE COPIES ;
      case THREE OR MORE COPIES:
          null statement ;
  if state = ONE COPY/MIGRATORY then
      migrate the block ;
  else one migration ← FALSE ;
      replicate the block ;

write miss invalidating one or more copies of a block:
  if state = ONE COPY/MIGRATORY then
      if dirty = FALSE then
          state ← ONE COPY ;
          one migration ← FALSE ;
  elsif last invalidator ≠ current processor
          and state = ONE COPY then
      if one migration then
          state ← ONE COPY/MIGRATORY ;
          else one migration ← TRUE ;
  else state ← ONE COPY ;
  last invalidator ← current processor ;
  migrate the block ;

write hit invalidating one or more copies of a block:
  if last invalidator ≠ current processor
          and state = TWO COPIES then
      if one migration then
          state ← ONE COPY/MIGRATORY ;
          else one migration ← TRUE ;
  else state ← ONE COPY ;
      one migration ← FALSE ;
  last invalidator ← current processor;
  perform the invalidations;

write hit on a clean, exclusively-held block:
  if last invalidator ≠ current processor
          and state = ONE COPY then
      if one migration then
          state ← ONE COPY/MIGRATORY ;
          else one migration ← TRUE ;
  last invalidator ← current processor ;
```

Figure 3: Pseudo-code for part of an adaptive directory-based coherency protocol. This conservative protocol initially uses replicate-on-read-miss and requires two successive "migratory events" to classify a block as migratory.

represented in two parts. First, the variable *state* indicates how many copies of the block have been made since the last time it was held exclusively by one node. *State* also specifies whether the block is migratory or not. Second, the *dirty* flag indicates whether the block has been modified if its state is "ONE COPY" or "ONE COPY/MIGRATORY".

For the purposes of this example, we have assumed that the directory entry for a memory block, particularly the *last invalidator* and *one migration* fields, is preserved even when the block is not present in any cache. Thus, on a write-hit on a clean, exclusively-held block the protocol checks to see whether the block exhibits migratory behavior spanning the most recent interval in which the block was uncached. This is particularly useful in systems with small caches. It means that when a migratory block is reloaded with a read miss that the cache does not have to send a message back to the memory controller to obtain write permission. This is still a big savings even if there are relatively few coherency messages.

3 Methodology

3.1 Benchmark Suite

We used five of the SPLASH programs for our benchmark suite: `Cholesky`, `Locus Route`, `MP3D`, `Pthor`, and `Water`. We refer the reader to Singh *et al.* [19] for a detailed description of each program. We ran the programs using the standard inputs provided with the suite: `Cholesky` using `bcstk14`, `Locus Route` using `Primary2.grin`, `MP3D` for 10 iterations with 10,000 particles using a 14x24x7 space array, `Pthor` using `risc`, and `Water` using `LWI12`. Given these inputs, the amount of shared memory used by each program was 1476 KBytes by `Cholesky`, 1232 KBytes by `Locus Route`, 552 KBytes by `MP3D`, 2676 KBytes by `Pthor`, and 200 KBytes by `Water`.

3.2 Simulation Techniques

Given the modest problem sizes, we simulated executions for a sixteen processor system. We used Tango [8] to perform execution-driven simulation and to generate shared-memory access traces for each program in our benchmark suite. To estimate the impact on execution time for one architecture, we performed execution-driven simulation using a version of *dixie*, a Tango memory system simulator for DASH, that implements the adaptive protocol. Lenoski *et al.* [15] provides a detailed description of this architecture.

To evaluate the impact of the adaptive protocols on message traffic for a wide variety of cache and block sizes, we used Tango to generate shared-memory access traces. These traces were used to drive a less-detailed but faster memory system simulator. The traces include accesses to ordinary shared data, but exclude accesses to synchronization variables, private data, and instructions. The traces varied in length from 3,734,816 shared-memory accesses for `MP3D` to 18,088,572 shared-memory accesses for `Water`.

3.3 Simplified Architectural Model

In the simplified model, cache coherence is enforced by a directory-based, write-invalidate protocol using a delayed write-back policy. Each processor has a 4-way set-associative cache that uses an LRU replacement strategy. A modified cache block is written back to main memory when it is replaced in the cache by another block, or when the data is accessed by another processor.

The intended purpose of using an adaptive protocol is to reduce the amount of interprocessor communication required by programs with migratory data. Our machine simulator is therefore designed to compare the amount of interprocessor communication required by the adaptive protocol to the amount required by a replicate-on-read-miss protocol. The model has two types of message. Short messages do not contain the contents of data blocks. They are used for requests and acknowledgements. Long messages, such as responses to miss requests, contain the contents of a data block. The number of messages necessary to perform an operation varies depending on the placement of the directory entry and of any existing cached copies of the block. Table 1 summarizes the number of inter-node messages that we charge to perform a cache operation. For each kind of operation, the number of messages depends on whether or not the home node of the block is the same as the node that initiated the operation, on the current state of the block, and on the cardinality of the set *DistantCopies*, the set of nodes other than the initiator and home nodes at which copies of the block are cached. For example, if a block is dirty there will be exactly one cached copy of the block, but $\|DistantCopies\|$ will be zero if that copy is at either the initiator or the home nodes.

We assume that when a node drops a clean entry from its cache that a message is sent to the block's home node to notify the directory of this action. If such a message is not sent, then later when the block is written by another node it would be necessary to send an invalidation request message that requires an acknowledgement. One could argue that the notification message is a cheap, low-priority "maintenance" message that can be sent asynchronously and that therefore it should not be charged for on the same basis as messages whose latency are on the critical path that determines the execution time of any operation. Nonetheless, we charge the same for these messages as for the other messages.

The assignment of data pages to specific nodes affects the performance of cache coherency protocols in two ways. If page placement is poor then a higher fraction of all coherency operations will require inter-node communication and a higher fraction of the operations requiring communication will require more messages. Although these effects are minimal for migratory data, using a poor page placement algorithm would result in a overall inflated estimate of the total number of messages needed. The simulator therefore attempts to find a reasonable page placement. Rather than attempting to simulate a general dynamic NUMA page-placement protocol [7], we use a simple dynamic technique for finding a good static placement that is similar to those used by Bolosky *et al.* [3] and Stenström *et al.* [21]. We assume that private data and code could have been placed and replicated perfectly to eliminate inter-node message traffic.

In contrast, our execution-driven simulations with *dixie* use the standard round-robin memory allocation. In both the trace-driven and the execution-driven simulations, the page size was 4 Kbytes.

Block size was varied from 16 to 256 bytes to evaluate the effects of larger block sizes on the amount of migratory data and on the ability of the adaptive protocols to deal with it.

The size of the local caches can be expected to affect the benefits of adaptive protocols in two ways. First, with

	home node	block status	inter-node messages without data	inter-node acknowledgements containing a data block
read miss	local	clean	0	0
read miss	local	dirty	1	1
read miss	remote	clean	1	1
read miss	remote	dirty	$1 + \|DistantCopies\|$	$1 + \|DistantCopies\|$
write miss	local	clean	$2 \times \|DistantCopies\|$	0
write miss	local	dirty	1	1
write miss	remote	clean	$1 + 2 \times \|DistantCopies\|$	1
write miss	remote	dirty	$1 + \|DistantCopies\|$	$1 + \|DistantCopies\|$
write hit	local	clean	$2 \times \|DistantCopies\|$	0
write hit	remote	clean	$2 + 2 \times \|DistantCopies\|$	0

Table 1: The number of inter-node messages generated by each type of cache operation requiring communication between the cache and memory controllers. *Home node* is the location of the directory entry for the block. It can either be the the node that initiated the operation (local) or some other node (remote). *DistantCopies* is the set of cached copies that are located at neither the local nor the home node. It is therefore a subset of *CopySet*.

smaller caches the replacement rate will be higher and therefore the time spent in coherency operations will be a smaller fraction of the total execution time. Since adapting to migratory access is a strategy that reduces the number of coherency operations, this limits potential benefits to be a fraction of an already small cost. Second, replacements interfere with the ability of the adaptive policies to effectively identify migratory data. To examine these effects, we simulated caches ranging from 4K bytes to 1M bytes.

4 Evaluation

4.1 Trace-Driven Simulation of the Directory-based Protocol

We evaluated three adaptive protocols. The *conservative* protocol corresponds to the implementation discussed in Section 2.2. This version initially manages each cache block under the copy-on-read-miss policy and it has a built in delay that requires that a cache block to migrate twice under the conventional copy-on-read-miss policy before it is classified as migratory. The *basic* protocol also starts each block under copy-on-read-miss, and classifies a block as migratory or other after a single event. The *aggressive* protocol initially classifies all blocks as migratory and will reclassify them based on a single event.

Since our simulations are based on traces of references to shared data, our evaluations of the three protocols are cast in terms of how they impact on the cost of managing shared data. Table 2 presents the number of memory system messages used. Using a fixed 16-byte block size, message counts are tabulated by application and by protocol for cache sizes ranging from four kilobytes per node up to one megabyte per node. For every combination of cache size and application, the number of non-data-carrying messages decreases substantially as each of the more aggressive adaptive protocols is used. For example, for MP3D and 1 M caches the most aggressive protocol uses 64.5 percent fewer of these messages than the conventional protocol. This indicates that, as intended, the protocols are successfully identifying migratory data and are realizing the expected elimination of most of the protocol messages. The fact that as more aggressive protocols are used, the number of data-carrying messages is constant or shows a very slight increase is an indication that the effect of mis-classifying data is small.

Table 2 also tabulates the percentage reduction in total message count. If all messages cost the same, this measures the percentage reduction in the cost of accessing shared data. For example, with large caches the most aggressive protocol yields a cost reduction of between 40 and 50 percent for `Cholesky`, `MP3D`, and `Water` and more modest cost reductions of between 10 and 20 percent for `Locus Route`, and `Pthor`. If data-carrying messages are charged twice as much as the other messages, the cost reductions will be less. For example, for one megabyte caches and the aggressive protocol the cost reductions for `MP3D` and `Locus Route` are still 38 and 10 percent, respectively, if the ratio of costs is two to one for the two kinds of message. With a four to one ratio of costs these figures decrease to 27 and 6.4 percent, respectively.

Note that the relative effectiveness of the adaptive protocols improves as the cache size increases. This is caused by two related effects. First, with larger caches there are fewer misses and writebacks due to capacity and conflict constraints, so coherency operations represent a larger proportion of the overall communication. Second, classifying a block as migratory is useful only if it allows one to use a more efficient mechanism to migrate it between caches. The higher replacement rates of small caches increase the probability that although a block exhibits the migratory pattern, its movement will be back and forth between main memory and a cache rather than between caches.

Choosing a larger block size can be expected to have an adverse effect on the effectiveness of the adaptive protocols, especially the more aggressive variants. One reason is that for larger blocks there are fewer opportunities to use the mechanism. Furthermore, as block size increases, fewer blocks will be migratory because of false sharing; even though there may be a lot of migratory variables, as block size increases more of them will be stored in blocks that will exhibit other sharing patterns. We evaluated the effect of increasing the cache block size by measuring the cache activity when there are no collisions in the cache or capacity induced misses.

Table 3 summarizes the reduction in the number of cache coherency messages for each of the applications as the cache blocks size changes. Measured in terms of the total number of messages sent, the effectiveness of the adaptive protocol decreases for `MP3D` as the block size increases. In fact, for

Cache Size	conventional		conservative			basic			aggressive		
4 Kbyte	w/o data	w/ data	w/o data	w/ data	%	w/o data	w/ data	%	w/o data	w/ data	%
Cholesky	4549	2429	3921	2429	**9.01**	3820	2429	**10.5**	3617	2429	**13.4**
Locus Route	2052	1126	1864	1127	**5.90**	1778	1128	**8.57**	1687	1128	**11.4**
MP3D	2092	934	896	935	**39.5**	855	936	**40.8**	784	936	**43.1**
Pthor	2041	1018	1666	1022	**12.1**	1606	1024	**14.1**	1549	1024	**15.9**
Water	3290	1644	2535	1644	**15.3**	2527	1644	**15.5**	2515	1644	**15.7**
16 Kbyte											
Cholesky	2074	941	1225	941	**28.1**	1096	942	**32.4**	872	942	**39.8**
Locus Route	1403	660	1268	662	**6.43**	1220	667	**8.54**	1131	667	**12.8**
MP3D	2236	921	946	923	**40.8**	905	923	**42.1**	836	923	**44.3**
Pthor	1942	912	1568	916	**13.0**	1508	918	**15.0**	1457	918	**16.7**
Water	2542	1121	1557	1121	**26.9**	1550	1121	**27.1**	1537	1121	**27.4**
64 Kbyte											
Cholesky	2426	908	1421	908	**30.1**	1244	908	**35.5**	974	908	**43.5**
Locus Route	1302	531	1175	535	**6.75**	1134	541	**8.66**	1058	542	**12.8**
MP3D	1779	612	703	613	**45.0**	682	614	**45.8**	634	614	**47.8**
Pthor	1848	825	1471	829	**13.9**	1405	833	**16.3**	1360	833	**18.0**
Water	1695	584	698	584	**43.8**	692	584	**44.0**	683	584	**44.4**
256 Kbyte											
Cholesky	2451	893	1416	893	**31.0**	1227	893	**36.6**	972	893	**44.2**
Locus Route	1270	472	1139	476	**7.32**	1091	485	**9.56**	1020	485	**13.6**
MP3D	1769	596	690	598	**45.6**	670	598	**46.4**	628	598	**48.1**
Pthor	1744	761	1365	766	**14.9**	1300	770	**17.3**	1262	771	**18.9**
Water	1687	575	686	575	**44.3**	681	575	**44.5**	674	575	**44.8**
1024 Kbyte											
Cholesky	2356	856	1239	856	**34.8**	1029	856	**41.3**	870	856	**46.3**
Locus Route	1268	470	1136	474	**7.35**	1076	483	**10.4**	1018	483	**13.7**
MP3D	1769	596	686	598	**45.7**	665	598	**46.6**	629	598	**48.1**
Pthor	1732	755	1359	760	**14.8**	1291	764	**17.3**	1257	764	**18.7**
Water	1687	574	685	574	**44.3**	680	574	**44.5**	673	574	**44.8**

Table 2: Message counts (in thousands) by cache size, application, and protocol. The column labelled % under each of the adaptive protocols (conservative, basic, and aggressive) is the percentage reduction in total messages used compared with the conventional protocol.

MP3D, false sharing causes the number of invalidations to increase as the block size increases from 64 bytes to 128 bytes. This indicates that the data ping-pongs between processors. The effectiveness of the adaptive protocol increases for Cholesky and Locus Route as the block size increases. The effectiveness of the adaptive protocol increases for Pthor and Water until the block size is 128 bytes. Using the most aggressive adaptive protocol is still the correct strategy for all of the applications and for all of the block sizes for which simulations were done.

The disadvantages of using large block sizes with adaptive protocols become apparent if one applies cost models that charge more for messages that transport data than those that do not. If the ratio of costs is two to one, then for all applications except Cholesky the savings decline substantially for block sizes over 64 bytes[2] Cholesky does show minor improvement. If one applies a cost model that charges one unit per message plus one unit for each sixteen bytes of data transmitted to the data in Table 3, the savings decline even faster and any advantages of the adaptive protocol are close to zero for 256-byte blocks. Cholesky shows a savings of 7.5% for the conservative protocol and 8% for the aggressive protocol, while Locus Route shows a savings of 0.9% for the conservative protocol but a *penalty* of 0.4% for using the aggressive protocol.

[2]Due to space limitations this table is omitted from the paper, but it will appear in a TR. Although it is inconvenient, the costs can be constructed from the data in the tables presented here.

4.2 Execution-Driven Simulation of the Directory-based Protocol

In order to determine the impact that the reduced communication by the adaptive protocol has on execution time, we performed execution-driven simulations of the three programs exhibiting the largest reductions: Cholesky, MP3D, and Water. The basic adaptive protocol reduced the execution times of the parallel sections of these programs by 19.3%, 10.4%, and 3.5%, respectively. As expected, most of the savings are from reduced write-hit latencies. MP3D showed the greatest improvement because it generates the most intense inter-cache traffic of the programs simulated. Even then, there was almost negligible added latency observed due to contention for either the interconnection network or for the local bus of each node. Surprisingly, eliminating the extra invalidation operations decreases the average latency of primary cache read misses by 20%. It ac-

Block Size	conventional		conservative			basic			aggressive		
16-byte	w/o data	w/ data	w/o data	w/ data	%	w/o data	w/ data	%	w/o data	w/ data	%
Cholesky	2337	846	1183	846	**36.2**	961	846	**43.2**	849	846	**46.7**
Locus Route	1268	470	1136	474	**7.35**	1076	483	**10.4**	1018	483	**13.7**
MP3D	1769	596	686	598	**45.7**	665	598	**46.6**	629	598	**48.1**
Pthor	1731	754	1357	760	**14.8**	1287	764	**17.5**	1258	764	**18.6**
Water	1687	574	685	574	**44.3**	680	574	**44.5**	673	574	**44.8**
32-byte											
Cholesky	1294	465	637	466	**37.3**	525	466	**43.7**	469	466	**46.9**
Locus Route	812	300	720	304	**7.90**	674	310	**11.5**	638	310	**14.7**
MP3D	1219	413	484	416	**44.8**	464	417	**46.0**	443	417	**47.3**
Pthor	1835	789	1382	795	**17.0**	1311	799	**19.6**	1291	799	**20.4**
Water	1007	342	406	342	**44.6**	403	342	**44.8**	399	342	**45.1**
64-byte											
Cholesky	773	275	365	275	**38.9**	307	275	**44.4**	280	275	**47.0**
Locus Route	609	225	525	229	**9.51**	488	233	**13.5**	468	233	**15.8**
MP3D	1011	364	461	372	**39.5**	446	376	**40.2**	435	376	**41.0**
Pthor	1777	736	1330	742	**17.5**	1261	748	**20.1**	1246	748	**20.7**
Water	667	226	266	226	**44.9**	264	227	**45.1**	262	227	**45.3**
128-byte											
Cholesky	506	178	228	178	**40.6**	198	179	**44.9**	184	179	**46.9**
Locus Route	479	180	401	183	**11.3**	376	187	**14.6**	366	187	**16.1**
MP3D	959	363	494	375	**34.3**	472	379	**35.7**	466	379	**36.1**
Pthor	1774	726	1337	733	**17.2**	1266	738	**19.8**	1255	738	**20.3**
Water	517	175	248	175	**38.9**	232	176	**41.1**	231	176	**41.3**
256-byte											
Cholesky	373	130	165	131	**41.2**	148	132	**44.4**	142	132	**45.7**
Locus Route	451	171	373	174	**12.2**	356	177	**14.2**	352	177	**14.9**
MP3D	1024	401	594	419	**28.9**	562	426	**31.0**	559	426	**30.9**
Pthor	1803	738	1396	745	**15.7**	1320	751	**18.5**	1314	751	**18.7**
Water	481	162	311	163	**26.3**	275	165	**31.5**	275	165	**31.7**

Table 3: Message counts (in thousands) by block size, application, and protocol. The caches are large enough to eliminate capacity misses. The column labelled % under each of the adaptive protocols is the percentage reduction in total messages used compared with the conventional protocol.

complishes this by nearly eliminating contention at the secondary cache.

We did not observe as large a reduction in the number of messages for the execution-driven simulations as for the trace-driven simulations; for example, 32% versus 46%, respectively, for MP3D. The reason for this difference is largely unrelated to the protocol: The trace-driven simulator used a better page placement policy, reducing the amount of internode communication for the other types of data. Poor page placement has a smaller effect on the number of messages used for migratory data.

4.3 Bus-based Protocol

Due to space limitations, we present only a brief summary of our evaluation of the bus-based protocols.

In terms of the power of the different kinds of protocol, the main difference is that the snooping protocol can not retain the classification of a block across time intervals in which the block is not cached. In terms of costs, the main advantage of the bus-based protocols is the ability to broadcast requests and the freedom from having to wait for individual acknowledgements to invalidation requests. Thus, on the bus-based system the cost of executing a coherency protocol will be proportional to the number of bus operations rather than the number of messages needed to implement While cache misses in a directory scheme can generate a variable number of messages (See Table 1.) that depends on where the data and directory are located, an operation in a cache-based system will generate at most one split bus transaction. Despite these differences, the two classes of protocol behave similarly.

We considered two cost models for the bus-based system. In the first model all memory or coherency operations take one bus transaction and thus have unit cost. In the second model, operations that require replies (misses and adaptive invalidations) are charged two units of cost, while operations that do not require replies (writebacks and invalidations in the conventional protocol) are charged one unit. Using these cost models we see cost reductions similar to those found in the directory-based protocols. Using the first model, programs such as Water and MP3D have savings of over 40 percent for caches of 64 K and larger. In contrast, Pthor gets a savings of 7 percent for 64K caches and 10 percent for one megabyte caches. Using the second model results predicted savings in the 25 to 30 percent range for the Water and MP3D. For Pthor the savings are 3.9 and 5 percent for the two cache sizes.

The variations among the adaptive protocols are similar to those observed for the directory protocols, but the magnitudes of the differences are less. For all cache sizes the programs that do best using adaptive protocols also do best with the more aggressive ones. The other applications show comparatively less benefit from an aggressive protocol, and for large cache sizes the aggressive protocols do slightly worse than the conservative ones.

5 Related Work

Software caching approaches to page placement on NUMA multiprocessors have used several techniques for handling migratory data. In an experimental study that compared a large number of page caching protocols, LaRowe and Ellis [13] concluded that always using a migrate-on-read-miss policy is inferior to always using replicate-on-read-miss. Recognizing the potential advantages of using an adaptive policy, LaRowe *et al.* [14] described a technique for detecting migratory data, but found it to be ineffective. PLATINUM [7, 6] uses the classification technique described in this paper. Because page movement has a cost several times that of a coherency operation, that system uses a conservative protocol with high hysteresis in classifying a page as migratory. Unfortunately, the relative benefit of an adaptive protocol for migratory data is considerably less under these software systems than it is for hardware caches. First, the adaptive protocol saves only coherency transactions, but because they are so much cheaper than page movement transactions the protocol can save only a small fraction of the cost of managing each migratory page. Second, because these systems work at a virtual page granularity, false sharing tends to hide migratory data, providing the adaptive protocol fewer opportunities to apply the migrate-on-read-miss policy.

Several software distributed shared-memory systems have used lock acquisition to indicate the intent to access migratory data. Midway requires the programmer to associate shared data with synchronization [2]. Munin permits, but does not require, the programmer to associate shared data with synchronization [4]. Our results suggest that more data is migratory at a fine grain than may be recognized by a programmer at a coarse grain.

The Sequent Symmetry multiprocessor (model B) [16] has a non-adaptive snooping protocol that uses a migrate-on-read-miss policy for all modified blocks. This policy is also used in the directory-based coherency mechanism of the MIT Alewife [5] system. While this policy is optimal for migratory data, using it on data with other sharing patterns causes additonal read misses. Thakkar [22] observes that read cycles dominate bus traffic on the Sequent and states that the extra read misses caused by this policy contribute significantly to this traffic. Our adaptive protocols avoid this problem. A quantitative comparison between these protocols and the adaptive protocols is needed.

Although the adaptive protocols have the advantages of transparency to user programs and compatibility with a wide range of existing processor architectures, they are on-line and therefore have the disadvantage of being limited to reacting to past and present patterns of access. In contrast, off-line analysis can make predictions about the future behavior of a program and if those predictions are accurate, use them to outperform an on-line algorithm. For example, data identified as migratory could be moved explicitly on a read access if the architecture provides a "load with intent to modify" instruction such as those assumed by the *Read-With-Ownership* operation of the sophisticated version of the "Berkeley Ownership" protocol [12].

The benefit of using either a protocol optimized for migratory data or explicit "load with intent to modify" instructions is the elimination of separate inter-cache invalidation operations. The reduction in traffic can also improve the latency of other operations by decreasing contention. A program using these mechanisms, however, still has to wait for the migratory read operation to complete. This waiting can be decreased using either programmer- or compiler-inserted prefetch requests. Mowry and Gupta [17] studied the effects of inserting non-binding prefetch and prefetch-exclusive requests into three (MP3D, LU, and PTHOR) of the SPLASH programs run on a DASH simulator. With prefetches inserted by hand, their simulations show the same reduction in time spent waiting for invalidations as the the adaptive protocols and they also show a substantial reduction in time spent waiting for read misses. If compilers can be written that achieve the same performance, then a carefully designed prefetching mechanism may be the best approach to the problem. This will require analysis techniques that are powerful enough to insert sufficent prefetches to be effective, but not so many as to generate excessive traffic.

Although support for prefetching appears in some new architectures, not all prefetching mechanisms excel at managing migratory data. The DEC Alpha [9] architecture has prefetch instructions, but they operate on 512-byte blocks. Our results show that even with smaller block sizes a substantial amount of migratory access is masked by false sharing. A more serious problem is that all three of the multiprocessor system architectures designed for the Alpha (Alpha Demonstration Unit, DEC 4000 AXP, and DEC 7000 AXP) are based on a hybrid write-update/write-invalidate coherency protocol that manages migratory data in a very inefficient way. On these machines it can take as many as three inter-cache operations to migrate a block from P_i to P_j: (1) a read miss by P_j that replicates the block, (2) a write hit by P_j that updates the copy in P_i's secondary cache invalidating the copy in the primary cache, and (3) a write hit causing the invalidation of the copy in P_i's secondary cache. These designs do not appear to have a mechanism that would allow the prefetch-exclusive instruction (called "FETCH_M") to bypass this protocol.

A program can be decorated with annotations to enhance cache performance in the Cooperative Shared Memory scheme of Hill *et al.* [11]. The annotation check_in signals when a process is done with a block and other annotations (check_out_X and prefetch_X) mark the points in the program at which a process expects to receive exclusive read/write access. The check_out_S annotation is intended to mark the beginning of an interval in which a block is read shared. These annotations are only advisory; the system still maintains coherence even if they are not used or they are used incorrectly. The "Queue On Sync Bit" (QOSB, called "Queue On Link Bit" in [10]) mechanism implements a similar scheme in hardware.

Stenström *et al.* [20] have developed a directory-based adaptive protocol for migratory data that is very similar to the one we describe. Their rule for shifting into migratory mode is identical to the one we use. Both protocols shift out of migratory mode on read miss to a clean and migratory block. Their protocol also shifts on any write miss to a migratory block. Since there is very little dynamic reclassification in the SPLASH programs, our *dixie* simulations are consistent with their results.

6 Conclusions

The adaptive protocol for identifying and dealing with migratory data appears to be a worthwhile extension to existing directory-based cache coherency protocols. In our trace-driven simulations, it never sent more messages than a standard replicate-on-read-miss protocol. With large caches, an aggressive adaptive protocol reduces the number of memory system messages used by over thirteen percent in the worst case and for three of the five applications we examined, the number of memory system messages saved approaches the theoretical maximum of fifty percent. In our execution-driven simulations of the DASH multiprocessor, a CC-NUMA architecture, we found that the adaptive protocol could reduce execution time by almost 19% for some applications. We expect these results to improve with better page placement.

Our results indicate that for small cache block sizes there is no advantage in being conservative. The aggressive protocol that reclassifies blocks immediately, that initially classifies blocks as migratory, and that remembers classifications over intervals in which data is not cached performs better than any of the more conservative strategies.

One artifact of this study is that the traces used in the simulations contain operations only on ordinary shared data. This may have several effects on the results. If references to private data were included then private and shared data will compete with one another for space in the cache. For a particular actual cache size, the effective cache space available for shared data will be smaller.

While at first blush one might expect that the adaptive protocols would not affect the cost of operations on private data, treating private data as though it is migratory will reduce the cost of process migration.

Acknowledgements

We would like to thank Todd Mowry and Anoop Gupta for providing us with the memory system simulator for DASH, as well as John Carter, David Chaiken, Sandhya Dwarkadas, Mootaz Elnozahy, Pete Keleher, Willy Zwaenepoel and the ISCA referees for their comments.

References

[1] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[2] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[3] W.J. Bolosky, R.P. Fitzgerald, ard M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[6] A. L. Cox. *The Implementation and Evaluation of a Coherent Memory Abstraction for NUMA Multiprocessors.* PhD thesis, University of Rochester, Rochester, NY, May 1992.

[7] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[8] H. Davis, S. Goldschmidt, and J. L. Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, 1990.

[9] Alpha AXP architecture and sytems. *Digital Technical Journal*, 4(4), Special Issue 1992.

[10] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd Symposium on Architectural Support' for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[11] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware support for scaleable multiprocessors. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[12] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[13] R. P. LaRowe and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[14] R. P. LaRowe, C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[16] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[17] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *JPDC*, 12:87–106, June 1991.

[18] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, May 1984.

[19] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[20] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[21] P. Stenström, 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.

[22] Shreekant S. Thakkar. Performance of Symmetry multiprocessor system. In Michel Dubois and Shreekant S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 53–82. Kluwer Academic Publishers, Boston, 1989.

[23] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support' for Programming Languages and Operating Systems*, pages 243–256, April 1989.