Using Dataflow Analysis Techniques to Reduce Ownership Overhead in Cache Coherence Protocols

JONAS SKEPPSTEDT and PER STENSTRM
Chalmers University of Technology

In this article we explore the potential of classical dataflow analysis techniques in removing overhead in write-invalidate cache coherence protocols for shared-memory multiprocessors. We construct three compiler algorithms with varying degree of sophistication that detect loads followed by stores to the same address. Such loads are marked and constitute a hint to the cache to obtain an exclusive copy of the block so that the subsequent store does not introduce access penalties. The simplest of the three compiler algorithms analyzes the existence of load-store sequences within each basic block of code whereas the other two analyze load-store sequences across basic blocks at the intraprocedural level. The algorithms have been incorporated into an optimizing C compiler, and we have evaluated their efficiencies by compiling and executing seven parallel programs on a simulated multiprocessor. Our results show that the detection efficiency of the most aggressive algorithm is 96% or higher for four of the seven programs studied. We also compare the efficiency of these static algorithms with that of dynamic hardware-based algorithms that reduce ownership overhead. We find that the static analysis using classical dataflow analysis results in similar performance improvements as dynamic hardware-based approaches.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; C.1.2 [**Processor Architectures**]: Multiprocessors—*multiple-instruction-stream, multiple-data-stream processors* (*MIMD*); D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Design, Evaluation

Additional Key Words and Phrases: Cache coherence, dataflow analysis, performance evaluation

1. INTRODUCTION

Dataflow analysis techniques are key to many standard optimizations done in state-of-the-art optimizing compilers. The strength of this algorithmic framework is that it is simple and well understood. More appealing is that it attacks a quite general problem that can be stated as follows: given a flow graph of a program, then establish dependency chains between instructions. What "dependency" means

This research has been supported by the Swedish Research Council for Engineering Sciences (TFR) under contract 94-315. A preliminary version of this article was presented at the *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*.
Authors' address: Department of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden; email: {jonass; pers}@ce.chalmers.se.

is problem specific and is formulated for each dataflow problem at hand. Our interest in dataflow analysis techniques is in exploring their usefulness in optimizing cache performance in shared-memory multiprocessors. This article deals specifically with the effectiveness with which dataflow analysis techniques can remove certain overheads in maintaining coherence among caches.

A write-invalidate cache protocol [Stenström 1990] is a widely used mechanism to ensure consistency among private caches in a shared-memory multiprocessor. It maintains consistency by removing all other cache copies of a memory block, when a processor modifies its local copy. An intrinsic shortcoming of such protocols, however, is that ownership of the block has to be obtained before it can be modified. This ownership acquisition involves sending invalidations to other cached copies and results in extra network messages. Moreover, it can also lead to substantial penalties under sequential consistency [Lamport 1979] because the processor must usually be stalled until ownership has been acquired, and this stall time can encompass hundreds of cycles in machines such as Stanford DASH [Lenoski et al. 1992]. Techniques that can reduce the overhead for ownership acquisition are therefore important.

A prevalent source of ownership overhead is caused by *migratory sharing* [Gupta and Weber 1992] that shows up when data are read and then modified by one processor at a time, for instance, when data is accessed in a critical section. Assuming that different processors access the critical section in turn, each invocation causes a cache miss followed by an ownership acquisition. Because both of these transactions are serviced by the cache attached to the processor that entered the critical section most recently, it is possible to eliminate the ownership overhead by requesting an exclusive copy at the first (load) access to the block. We will consider this in more detail in Section 2.

Our approach to invoke this optimization is to use dataflow analysis techniques to statically detect *loads* followed by *stores* to the same address at compile-time. Loads in such sequences are marked and will be replaced by *load-exclusive* instructions. A load-exclusive instruction not only loads a value into a register. It also gives a hint to the cache to obtain an exclusive copy of the block. In this article we present the design of three variations of this simple compiler approach. The simplest algorithm detects load-store sequences within basic blocks of code only, whereas the other two algorithms analyze load-store sequences across basic blocks at the intraprocedural level. The latter two variations deal with uncertainties as to which execution path is taken.

We have incorporated the algorithms into an optimizing C compiler. By compiling seven parallel applications and then running them on a simulated multiprocessor, we have studied the efficiency with which they can detect load-store sequences as well as the impact on the execution time and traffic. While preliminary results of this evaluation were presented in Skeppstedt and Stenström [1994], we extend the results in that study by analyzing more application programs as well as analyzing how much traffic can be reduced. We find that the detection efficiency of the most aggressive algorithm is 96% or higher for four of the seven applications we have considered.

Another approach to reduce ownership overhead is to extend the cache coherence protocol to dynamically detect migratory sharing as proposed in Cox and

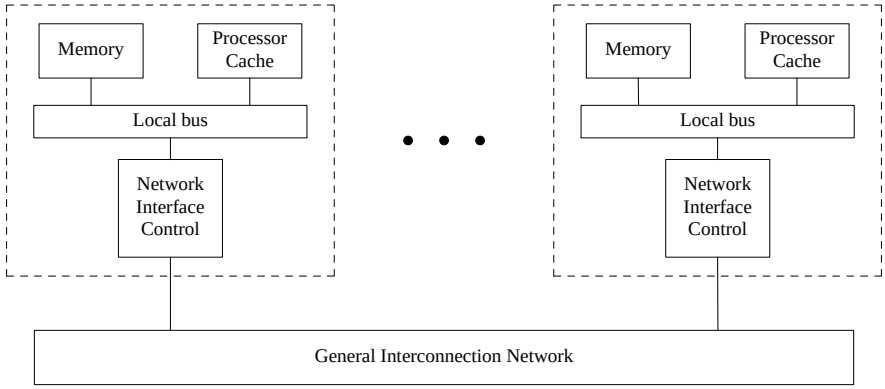

Fig. 1. Generic multiprocessor organization assumed in this study.

Fowler [1993] and Stenström et al. [1993]. To gain further understanding, we compare the compiler algorithms with cache coherence protocol extensions which make dynamic decisions on when a block should be loaded in exclusive or shared state. We find that the static analysis using classical dataflow analysis results in similar performance improvements as the dynamic hardware-based approaches.

The rest of the article is organized as follows. For a background, we review in Section 2 how ownership overhead shows up in cache-coherent shared-memory multiprocessors. Section 3 presents the dynamic hardware-based algorithms, and Section 4 presents the design of the compiler algorithms. We move on to the experimental methodology in Section 5, and in Section 6 we present our simulation results. Finally, we relate our findings to work done by others in Section 7 before we conclude in Section 8.

2. OWNERSHIP OVERHEAD IN SHARED-MEMORY MULTIPROCESSORS

Most shared-memory multiprocessors built today employ caches attached to each processing element. In systems with a small number of processors, processing elements are typically connected by a fast common bus as demonstrated by the SGI Challenge [Galles and Williams 1994] multiprocessor, and the caches primarily help such systems to reduce the bus traffic. By contrast, larger systems typically use general interconnection networks, such as wormhole-routed meshes, to meet a higher demand for bandwidth, exemplified by the Stanford DASH [Lenoski et al. 1992] and the M.I.T. Alewife [Agarwal et al. 1995]. The goal of using caches in such systems is primarily to shield the processors from the large access latencies that stem from the multihop nature of memory requests.

This study is concerned with both kinds of systems by assuming a generic organization according to Figure 1. In this system organization, each processing element contains a processor with its private cache and a portion of the globally shared memory. While the nodes are connected to a number of memory modules by a generic interconnection network, it is irrelevant for the rest of the discussion whether the memory modules are distributed equally among nodes. Consistency among the private caches in the processing nodes is maintained by a write-invalidate protocol, a prevalent choice of coherence policy. Before a block replicated among

```
P1                P2
lock      L       ...             ; P1 locks the critical section,
load      A       ...             ; loads and then
store     A       ...             ; modifies A before it
unlock    L       ...             ; exits the critical section.
...               lock      L     ; P2 does the same and experiences
...               load      A     ; a load miss followed by a
...               store     A     ; an ownership acquisition before it
...               unlock    L     ; exits the critical section.
```

Fig. 2. Two processors *P1* and *P2* enter a critical section where they read and modify variable A.

caches in such a protocol is modified, ownership must be acquired. This ownership acquisition can stall the processor under sequential consistency, resulting in significant write latencies. Moreover, ownership acquisition increases network traffic which, as a secondary effect, can increase the read and synchronization access latencies because of memory system contention also under relaxed memory consistency models.

Ownership overhead is especially pronounced under migratory sharing where a memory block migrates between different processing nodes and where each processor issues read-write sequences to the block. A typical example of when such a behavior occurs is when a data structure is read and modified in a critical section. In Figure 2 we show the coherence actions taking place when two processors *P1* and *P2* enter a critical section and read and modify a variable A one after the other. When *P1* has exited from the critical section, the block is modified, and the only copy of it is located in *P1's* cache. When *P2* enters the critical section, it will first experience a cache miss that is serviced by *P1's* cache and subsequently an ownership acquisition that again is serviced by *P1's* cache. We note that not only the load, but also the store instruction from *P2*, will cause cache coherence overhead. This ownership overhead will of course show up for subsequent invocations of the critical section as well.

An obvious optimization would be to request an exclusive copy at the time of the load miss if it were known that no other processor will access the block between the load causing the cache miss and the subsequent store instruction. Then the ownership acquisition caused by the store instruction is eliminated which removes the ownership overhead in terms of memory traffic and/or latency. Next we will study how dynamic hardware-based algorithms can reduce ownership overhead by detecting which blocks exhibit migratory sharing.

3. DYNAMIC DETECTION OF MIGRATORY SHARING

In Cox and Fowler [1993] and Stenström et al. [1993] extensions to basic write-invalidate protocols are proposed which dynamically detect migratory sharing and, for blocks deemed migratory, merge ownership requests with cache-miss requests.

The detection mechanism relies on the fact that the controller of the memory module that administrates the coherence actions for a block, called the *home*, receives cache miss as well as ownership requests. Consequently, home will see a cache-miss request followed by an ownership request from the same processor, with no intervening access from another processor, for migratory blocks. To detect this

sharing pattern for a block, home records the identity of the processor that most recently wrote to the block. When home receives an ownership request from a processor other than the last one that wrote to the block, and the number of copies is two, the block is deemed migratory. Cache-miss requests will then obtain an exclusive copy of the block which eliminates the separate ownership request associated with subsequent stores. These protocols are also capable of switching a migratory memory block back to normal treatment when its sharing pattern has changed. When a cache receives a request to give up ownership of a migratory block, and the cache has not yet written to the block, then the read-write chain is broken, and the block is classified by the cache as not being migratory. The terms adaptive protocols and hardware-based approaches refer to the same protocol extensions and will be used interchangeably throughout this article.

The additional protocol mechanisms to extend a full-map write-invalidate protocol with the migratory detection consists of one pointer per memory block which is needed for identifying the processor that last wrote to the block, two extra memory states to tag a block as migratory, and an extra cache block state to detect when migratory sharing ceases.

The reduction of the execution time by removing explicit ownership requests has been evaluated in Stenström et al. [1993], assuming an architecture similar to the one in Figure 1. They found that the write stall-time could be reduced by more than 80% under sequential consistency for applications exhibiting migratory sharing and that the traffic was reduced by more than 20%.

The adaptive protocols have knowledge about recent memory accesses because a memory controller sees the requests from all processors, and from this information it tries to predict future behavior. In contrast, our compiler algorithms predict sharing behavior based on static analysis, and that is the focus of the next section.

4. THE COMPILER ALGORITHMS

In this section we consider three simple compiler algorithms to detect load-store sequences aiming at removing ownership overhead. We start in Section 4.1 to present the dataflow problem our algorithms are concerned with and the constraints encountered by the algorithms. Then in Sections 4.2 and 4.3, we present in detail how the algorithms are implemented, and finally in Section 4.4 we discuss their expected limitations which we examine experimentally in Section 6.

4.1 Dataflow Analysis Problem

The purpose of the compiler algorithms is to detect sequences of loads and stores to the same address. Such load instructions are *marked*. Marked load instructions can then be used as hints to the memory system with the purpose of acquiring ownership of the block at the point of the load. If a marked load misses in the cache, ownership overhead for the subsequent store instruction can be eliminated. How the ownership acquisition is implemented is orthogonal to the marking algorithm itself. However, in Section 6 we will specifically consider the effectiveness of reducing ownership overhead by replacing marked loads by special *load-exclusive* instructions. Unlike an ordinary load instruction, a load-exclusive instruction not only loads data, it also instructs the cache to acquire an exclusive copy of the corresponding block.

The dataflow algorithm analyzes the intermediate code representation for each

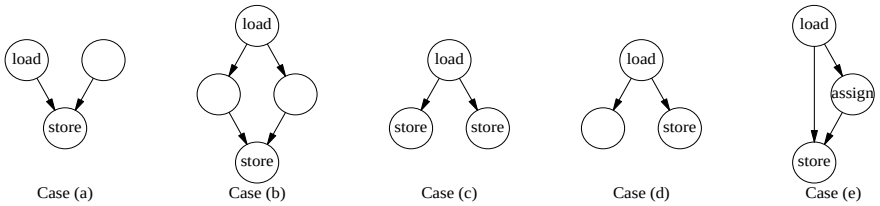

Fig. 3. Example flow graphs with load and store instructions to the same address. Conservative marks loads in cases (a), (b), and (c), while Speculative marks all loads.

process at the intraprocedural level to find load-store sequences that satisfy the following conditions during run-time:

(1) the load and the store refer to the same address and

(2) if the load is executed then the store is executed as well.

For load-store sequences which satisfy these conditions, the load instruction is marked. We have evaluated three algorithms that differ in aggressiveness when selecting which loads should be marked. The simplest one—denoted *Local*—analyzes the existence of load-store sequences within each single basic block only. Since a load-store sequence within a basic block by definition satisfies the second condition, Local is trivially implemented by making sure that the address of the load and the store is the same.

A limitation of Local is that it will miss opportunities of marking loads for load-store sequences that span several basic blocks. Therefore, we also consider two marking algorithms that rely on dataflow analysis across all basic blocks in a procedure. To illustrate the issues that then arise, let us consider the example flow graphs in Figure 3, where the vertices represent basic blocks and the edges represent the control flow between basic blocks. We assume that the effective addresses of loads and stores are the contents of their base pointers plus the offset. For simplicity, we also assume in Figure 3 that all loads and stores use the same base pointer and the same offset. Moreover, **assign** refers to the operation of changing the content of a base pointer.

To start with, we see that the loads in (a), (b), and (c) can be safely marked because conditions (1) and (2) are satisfied. As for cases (d) and (e), however, there are uncertainties as to whether a store to the same address will be executed. In both these cases, the uncertainty is due to conditional execution paths; in (d) the store may not be executed, and in (e) the base pointer might be changed.

In the first of the two algorithms that consider load-store sequences across basic blocks, loads are only marked if conditions (1) and (2) are satisfied, resulting in that only loads corresponding to flow graphs (a) through (c) in Figure 3 are marked. This algorithm is denoted *Conservative*.

To get an upper bound on the effectiveness of the marking capability of the compiler, we also consider an aggressive algorithm denoted *Speculative* that ignores uncertainties related to which execution path is taken, as in cases (d) and (e). This algorithm has a potential to cut more ownership overhead than Conservative, but it may also introduce invalidations and misses. Note that marking of loads is strictly a performance issue—excessive marking does not compromise correctness.

4.2 Algorithm Overview

Conceptually, our algorithms detect load-store sequences by using dataflow analysis similar to *live-variable analysis* [Aho et al. 1986]. Generally, our algorithms decide whether a store reaches a certain load by propagating stores backward in the flow graph taking conditions (1) and (2) into account.

We have implemented the algorithms as an optimization pass on the intermediate code level. Loads and stores are three-address code statements with a symbolic base pointer and an immediate-value offset for address calculation, denoted b_i and o_i in Figure 4, respectively.

All stores with the same base pointer and offset in a procedure form a class denoted the *store class* which forms the unit of dataflow. For example, the stores in basic blocks *B2* and *B3* in Figure 4 belong to the same store class. A store using a certain base pointer and offset is said to *generate* the corresponding store class. A generated store class propagates backward in the flow graph until either (1) the base pointer is changed which is said to *kill* all store classes that use this base pointer or (2) it is inhibited by Conservative to propagate further. Conservative requires that, for a store class to propagate to a basic block B from a successor of B, it must propagate from *all* successors of B. Thus the difference between Conservative and Speculative, as we will see in the next section, is that Conservative uses intersection as a confluence operator, while Speculative uses union. When a store class has propagated to a load, that load is marked. A store class which has propagated to a certain point in the flow graph is said to be *live* at that point.

In Figure 4, the store class generated in *B4* can propagate to *B1* without being killed, while the assignment of b_1 in *B3* kills the store class using b_1 which is not propagated to *B1* by Conservative. By contrast, Speculative propagates it from *B2*. Consequently, Conservative will mark only the load with b_2 as a base pointer while Speculative will mark both loads.

4.3 Implementation

The algorithm that implements the dataflow analysis and load-instruction marking consists of the following four passes:

(1) *Collect store classes.* This pass identifies all store classes that exist in the flow graph.

(2) *Local analysis.* This pass operates on each basic block. It creates sets of locally generated and killed store classes and marks loads for store classes generated and not killed in the basic block.

(3) *Dataflow analysis of store classes.* This pass is only applicable to Conservative and Speculative. It creates sets of store classes that are live at the beginning and at the end of each basic block through backward analysis.

(4) *Global analysis.* This pass is only applicable to Conservative and Speculative and marks loads corresponding to store classes that are live at the end of a basic block and that are not killed locally.

For Local, it suffices to perform the first two passes, and these passes can then be merged to a single pass. By contrast, Conservative and Speculative require all four passes. We next present the detailed implementation of each of the four passes.

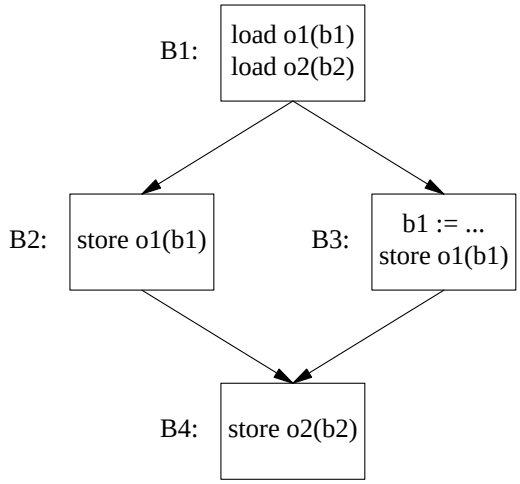

Fig. 4. Example flow graph.

Collect Store Classes. The purpose of this pass is to identify all store classes that exist in the flow graph. Recall that each store class is characterized by a base pointer and an offset. A new store class is inserted by assigning it an index to simplify dataflow operations.

In this pass each base pointer is associated with a number of indices corresponding to store classes that use this base pointer. When the content of the base pointer is changed, all corresponding store classes can then be killed by simply performing bit operations on the bit vectors used in the dataflow analysis described next.

Local Analysis. The purpose of this pass is to mark loads corresponding to store classes that are generated and not killed in each basic block. This is done by establishing two sets of store classes called *GEN* and *KILL*.

Assume B is a basic block with two sets *GEN* and *KILL* representing the generated and killed store classes, respectively. Moreover, there is a function—denoted *store_classes*(b)— which maps a base pointer b to the set of store classes that use b as a base pointer. The statements of a basic block are scanned in the backward direction. When a statement is a *store* with base pointer b and offset o, the store class with that base pointer and offset, denoted (b, o), is generated and added to the *GEN* set and removed from the *KILL* set:

$$B.GEN := B.GEN \bigcup \{ (b, o) \}$$
$$B.KILL := B.KILL - \{ (b, o) \}$$

When the statement is an *assignment of a base pointer*, all store classes for that base are killed. This is done by removing them from the *GEN* set and adding them to the *KILL* set:

$$B.GEN := B.GEN - store\_classes(b)$$
$$B.KILL := B.KILL \bigcup store\_classes(b)$$

Finally, when the statement is a *load*, and there is a store class in *GEN* with the same base pointer and offset, the load is marked. For other statements no action

is taken. Note that all dataflow operations described above are simply carried out as bit-vector operations.

Dataflow Analysis of Store Classes. The purpose of the dataflow analysis is to keep track of which store classes are live at the beginning and at the end of each basic block by letting store classes generated in one basic block propagate backward to the preceding basic blocks.

Given a basic block B, the set of store classes that are live at the beginning and at the end of a basic block are denoted $B.IN$ and $B.OUT$, respectively. Then for all immediate successor basic blocks S of B in the flow graph, we can formulate how store classes that are live at the beginning of each S ($S.IN$) propagate to B ($B.OUT$). This is done in the following dataflow equation:

$$B.OUT := op\ S.IN \quad \forall \text{ immediate successors } S \text{ of } B$$

For Conservative, it is required that a store class is live at the beginning of all immediate successor basic blocks, i.e., the operation *op* is intersection. By contrast, Speculative only requires that a store class is live in at least one successor basic block, i.e., *op* is union.

Finally, the dataflow equation to establish $B.IN$ uses the two sets $B.GEN$ and $B.KILL$ as follows:

$$B.IN := (B.OUT \bigcup B.GEN) - B.KILL$$

i.e., store classes live at the beginning of B are defined by the store classes live at the end of B plus store classes generated in B minus store classes that are killed locally in B.

The above operations applied to each basic block are repeated until no changes occur as in other iterative dataflow analyzes [Aho et al. 1986].

Global Analysis. The last step in Conservative and Speculative marks loads whose associated store classes belong to the OUT set of the basic block and are not killed locally. Thus given a basic block B, each load in B is marked for which the following conditions are true:

(1) there is a store class in $B.OUT$ with the same base and offset and
(2) the store class is not killed locally in B at a point after the load.

4.4 Performance Issues

Although the marking algorithms are simple, their potential to remove ownership overhead can be limited due to the following issues:

—aliasing,
—word versus block addresses, and
—intervening accesses by other processors.

Aliasing is a problem for any dataflow analysis, and in our case it can occur if a load and a store reference the same address using different base pointers. This is not dealt with by our algorithms and may limit the detection efficiency. In Section 6, we will investigate to what extent aliasing limits the detection efficiency for the evaluated applications.

```
Ref. P1     P2      Comment

T0:         Load    Cold miss
T1:  Load+  INV     Miss and invalidation
T2:         Load    Useless miss
T3:  Store
```

Fig. 5. Reference trace for two processors *P1* and *P2* to the same data item. The load by *P2* at *T2* experiences a useless miss due to the marked load (Load+) at *T1*.

The second issue stems from the fact that the compiler algorithms detect load-store sequences based on word addresses rather than block addresses. A trivial extension to incorporate this would be to align data structures on cache-block boundaries and then treat the offset in a store class as a block address. However, whereas load-store sequences to the same word has an intuitive explanation (e.g., read-modify-write operations on the same data), load-store sequences to different words in the same block seem less natural from a program behavioral point of view. Therefore, one may expect small additional gains by incorporating such block-level analysis in the algorithms. In Section 6, however, we will study how much this fact limits the gains of the compiler algorithms.

The last issue stems from the fact that the algorithms analyze the code of a single processor and lack information of how other processors' loads and stores interfere. Therefore, marked loads can actually increase the number of invalidations and misses. To illustrate the problem, consider the trace of loads and stores to the same address by two processors *P1* and *P2* in Figure 5. In that trace, *P2* first issues a load that results in a cold miss. Because of the load-store sequence of *P1*, the load at *T1* is marked (Load+) and results in an invalidation of the block in *P2*'s cache. As a result, *P2* will experience a useless miss [Dubois et al. 1995] at *T2*. An interesting observation is that such intervening accesses come from two sources: data races (interference at the word level) and false sharing (interference at the block level). Data races show up if two processors issue accesses to the same address and if at least one of them is a store. For example, in the trace of Figure 5, the Load at *T2* and the Store at *T3* result in a data race. However, *properly labeled programs* [Gharachorloo et al. 1990] do not have any data races because competing accesses are separated by a synchronization. Thus, if a load-store sequence is not separated by a synchronization in between, and the program is properly labeled, extra invalidations and misses are not introduced given that there is no false sharing. We will study these effects later in Section 6.

5. EXPERIMENTAL METHODOLOGY

First we present the compiler and benchmarks we have used. Next we define metrics of detection efficiency for the different compiler and hardware algorithms. Finally we present the multiprocessor architectures we have simulated to evaluate effects on execution time and traffic.

5.1 Compiler and Benchmark Programs

We have incorporated the compiler algorithms in an optimizing C compiler [Skeppstedt 1990] which compiles parallel applications using the ANL macros [Boyle et al. 1987]

Table I. Benchmark Programs and Data Set Sizes Used

Benchmark	Description and Data Sets
Water	N-body water molecular dynamics simulation 288 molecules, 4 time steps
Cholesky	Cholesky factorization of a sparse matrix matrix bcsstk14
MP3D	3D particle-based wind-tunnel simulator 10,000 particles, 10 time steps
LU	LU-decomposition of a dense matrix 200 x 200 matrix
Ocean	Ocean basin simulator 128 x 128 grid, tolerance 10^{-7}
Barnes-Hut	128 bodies
PTHOR	RISC circuit

and generates code for shared-memory multiprocessors based on SPARC processors. Even though our compiler performs many standard optimizations [Aho et al. 1986; Chow and Hennessy 1990], it becomes important to understand how the results compare to code compiled with other compilers. We have compared some key parameters with `gcc` (version 2.1) with optimization level O2. We have found that the numbers of loads and stores to shared memory typically differ by less than 1% between our compiler and `gcc`.

We have used a set of seven applications developed at Stanford University, five of which are part of the SPLASH-1 suite [Singh et al. 1992] (MP3D, Water, Cholesky, Barnes-Hut, and PTHOR). We used the data set sizes that are shown in Table I. MP3D, Water, and Cholesky have a fair amount of migratory sharing [Cox and Fowler 1993; Gupta and Weber 1992; Stenström et al. 1993], and we expect to see significant gains by the static as well as the dynamic techniques. By contrast, the remaining four applications (LU, Ocean, Barnes-Hut, and PTHOR) also exhibit other types of sharing patterns such as producer-consumer sharing. The mixed behavior of these applications is interesting in order to find out whether the compiler algorithms we study make bad decisions that can hurt performance when there is little room for improvements.

5.2 Metrics of Detection Efficiency

To understand how close to optimum the detection efficiency of the static and dynamic algorithms are, we have developed an optimal omniscient algorithm that removes all ownership overhead associated with load-store sequences. This algorithm (1) predicts whether further improvements would have been possible and (2) detects any wrong decisions made by the algorithms that can degrade performance.

For each load miss, the optimal algorithm decides whether or not an exclusive copy should be fetched. An exclusive copy is fetched if the following conditions are satisfied:

(1) the processor will later store to any word of the block and
(2) no other processor will make an intervening access to the block between the load and the store.

The optimal algorithm uses as input the global shared-memory reference trace

of a program. This trace has been derived by first compiling the program by an optimizing C compiler that contains our algorithms. Then the program is executed on a simulated multiprocessor with an organization according to Figure 1. This multiprocessor implements a write-invalidate protocol and assumes infinite caches. Each memory reference—be it a cache hit or cache miss—encounters unit-cycle delays. This simulation model is built on top of the CacheMire Test Bench [Brorsson et al. 1993]: a simulation and programming development platform consisting of simulated SPARC processors.

To compare how many load misses that are correctly handled by each algorithm, we define the set of loads that fetch exclusive copies according to the optimal algorithm as the set of *optimal loads*. With *coverage* for a certain algorithm we mean the fraction of optimal loads which are also handled correctly by that algorithm.

In order to capture load misses acquiring exclusive copies that potentially can cause more invalidations and misses, we also count these loads, which we call *bad loads*. A bad load is a load miss that acquires an exclusive copy such that another processor will access the block before it is written to. To contrast the number of bad loads for a certain algorithm with the number of optimal loads contained in the trace, we define the *degree of bad loads* to be the number of bad loads divided by the total number of optimal loads. Note that a 100% degree of bad loads means that an algorithm generates as many bad loads as the optimal algorithm generates optimal loads.

5.3 Simulated Multiprocessor Architectures

We have developed three detailed architectural simulation models: (1) a basic write-invalidate protocol which constitutes the baseline architecture, (2) the baseline extended with functionality to handle compiler-generated, coalesced read and ownership requests, and (3) two adaptive cache coherence protocols. These models are described in detail below. The simulation platform consists of a functional simulator of SPARC processors which generate memory references to an attached memory system architectural simulator with a detailed timing model. Since the executing processors are delayed according to the latencies encountered by each memory reference, the same interleaving of memory references will be encountered as in the target architecture.

Baseline Architecture. The overall organization of the baseline architecture is shown in Figure 1. It consists of 16 processing nodes. Apart from the local portion of the shared memory, each processing node also contains a two-level cache hierarchy whose organization is shown in Figure 6. It consists of a write-through, direct-mapped first-level cache (denoted *FLC*) with an associated first-level write buffer denoted *FLWB*. In the baseline, the *FLWB* buffers store requests to a copy-back, second-level cache (*SLC*), and there is full inclusion between the *FLC* and the *SLC*.

System-level cache coherence between the second-level caches is maintained by a Censier and Feautrier write-invalidate protocol which associates a bit vector with each memory block [Censier and Feautrier 1978]. Virtual pages are 4KB and are mapped to physical memory modules using a round-robin policy that interprets the four least significant bits of the virtual page number as the node identity. The node in which a certain page is mapped is called the *home* of all blocks in that page.

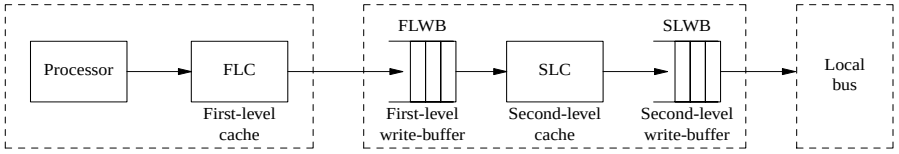

Fig. 6. The two-level cache hierarchy in each processing node.

Loads that miss in the *FLC* and the *SLC* cause a miss request to be sent to home. If the copy is present at home, and if home is the local node, the miss is serviced locally. Otherwise, two or four node-to-node traversals are required to fill the cache.

Stores are written through the *FLC* by buffering them in the *FLWB*. If the *SLC* copy is exclusive, the store can be carried out locally. Otherwise, ownership has to be acquired. The baseline protocol, as well as the other protocols we evaluate, implements *sequential consistency* by stalling the processor until ownership is granted. Depending on the location of home and whether another node has an exclusive copy, ownership acquisition may encounter zero, two, or four node-to-node traversals.

In Figure 6, a write buffer is also associated with the *SLC*, denoted *SLWB*. Since the processor is stalled on every global store, this buffer is not needed in the baseline architecture.

Support for the Compiler Algorithms. We evaluate the effectiveness of our compiler algorithms by replacing marked loads by special instructions denoted *load-exclusive*. Unlike ordinary load instructions, they also act as hints to the cache to obtain an exclusive copy of the associated block. While we could have evaluated the heuristic of disregarding ownership acquisition for load-exclusive instructions to blocks that are present but shared, we decided to let the cache acquire ownership of the block also when it is present in the cache.

The actions taken by the cache hierarchy when a load-exclusive request is issued are as follows. If the block is present in the *FLC*, the data are returned to the processor, and an ownership request is buffered in the *FLWB*. Since ownership acquisition is nonbinding [Mowry and Gupta 1991], the processor may proceed until the next store and does not have to await ownership to be granted. If the block is not present in the *FLC*, however, the processor has to stall, and a load-exclusive request is buffered in the *FLWB*. If the *SLC* has an exclusive copy, no action is taken besides filling the *FLC*. Conversely, if the block in the *SLC* is in a shared state, an ownership request is buffered in the *SLWB* and handled in the same way as a store to a shared *SLC* block in the baseline architecture. Finally, if the block is not present in the *SLC*, an exclusive copy is requested, and the processor has to stall until the block is filled in *FLC*. This includes as many network traversals as if ownership is requested in the baseline architecture.

Finally, if an ownership request is buffered in the *SLWB* when a store reaches the *SLC*, the *SLC* and the processor are stalled until ownership is granted. Note that the cache hierarchy mechanisms needed in this case include an *SLWB*. In our simulations we assume that the *FLWB* and the *SLWB* contain 8 and 16 entries, respectively.

Adaptive Cache Coherence Protocol Extensions. The adaptive cache coherence protocol we evaluate, according to Stenström et al. [1993] and Cox and Fowler [1993], extend the baseline write-invalidate protocol with a pointer of size $\log_2 16 = 4$ bits per memory block and an extra cache state. We have evaluated two variants of the adaptive cache coherence protocols. The first one, denoted *default-shared*, initially tags each memory block as not being migratory, while the second, denoted *default-migratory*, by default tags all memory blocks as being migratory.

The architectural parameters we assume for all three architecture variations are as follows. Each node contains a SPARC processor clocked at 100MHz, that is 1 pclock = 10ns. We model a 4KB *FLC* and an infinite *SLC*, both with a block size of 32 bytes. *FLC*, *SLC*, and local memory access times are 1, 6, and 30 pclocks, respectively. The nodes are interconnected with a contention-free network with a uniform node-to-node latency of 54 pclocks. Each control message is 5 bytes, and each data message is 37 bytes.

6. SIMULATION RESULTS

We first show the detection efficiency of the different algorithms against an optimal algorithm and then as a case study present simulated execution times and traffic.

6.1 Detection Efficiency

The diagrams of Figure 7 show the coverages (top) and degrees of bad loads (bottom) for the applications we have studied. For each application we show five bars that from left to right correspond to Local (L), Conservative (C), Speculative (S), default-shared (DS), and default-migratory (DM). Beneath each application, we also show the fraction of all cache misses issued by the processors that become optimal loads under the optimal algorithm. There are two distinct groups of applications; those with many optimal loads (Water, Cholesky, and MP3D) and the remaining four with few optimal loads.

We will first look at the applications with many optimal loads, namely, Water, Cholesky, and MP3D. Because of the migratory-sharing dominance in these applications, more than 80% of all cache misses are optimal loads in these applications.

The high coverages in Water and Cholesky indicate that the compiler optimizations and the *DM* protocol do very well as compared to the optimal algorithm—they eliminate 96% or more of all explicit ownership requests by merging them with the preceding load-miss requests. *L* does surprisingly well for Water and Cholesky, indicating that most load-store sequences in these applications are contained in certain basic blocks. By contrast, in the other applications there is a clear difference in coverage between *S* on one hand and *L* and *C* on the other. The coverage for *S* in MP3D is 97% whereas the coverages for *L* and *C* are only 72% and 73%. We analyzed which misses that *L* and *C* did not cover and found that they are due to a single load-store sequence in the procedure `move_single` that spans several basic blocks and which includes a conditional pointer assignment; *C* is unable to cover this case as opposed to *S* that marks the load in this situation. For Cholesky, *DS* is less effective with a coverage of 85%. We analyzed the optimal loads which were not covered by *DS* and found that the missing ones were due to data structures which were modified once and then read only for the rest of the execution. We also analyzed why the compiler algorithm was unable to reach a coverage of 100% and

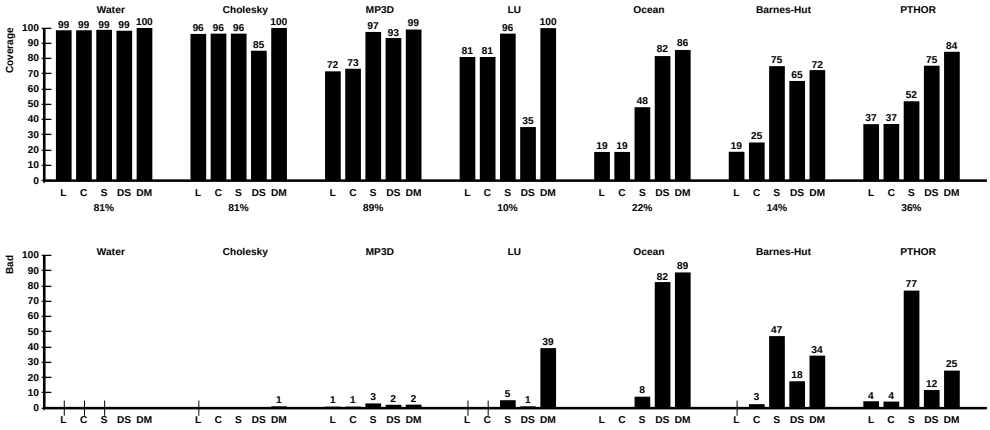

Fig. 7. Coverage (top) and degree of bad loads (bottom) in percent for the seven benchmarks.

found that they were due to a loop-carried load-store sequence which our compiler algorithms currently are unable to cover.

Let us now consider the degrees of bad loads by looking at the bottom diagram of Figure 7. We note that virtually no bad loads are generated by any optimization for Water and Cholesky. Moving on to MP3D, however, we note that all optimizations generate at least 1% bad loads, and *S* generated 3% bad loads. We have found that these bad loads are due to data races in the data structure denoted `Cells`. MP3D can execute correctly without protecting competing accesses with locks. To test whether the bad loads disappeared if data races are removed, we also ran MP3D with the locking option which eliminates all data races. The degree of bad loads now became 0.1%, and the remaining bad loads are attributable to false sharing in `Cells`. This experiment suggests that most of the bad loads are due to data races which means that if programs are properly labeled and if there is no false sharing, intervening accesses by other processors do not limit the effectiveness of the compiler algorithms.

The four applications to the right have a lower fraction of optimal loads ranging from 10% to 36% as shown in Figure 7, and there is not so much to gain for any of the algorithms. The detection accuracy for these applications is interesting, however, because bad loads can reverse the performance for these applications.

Starting with LU, *L* and *C* cover more than 80% of the optimal loads and do not generate any bad loads. By contrast, although *S* shows a higher coverage, the degree of bad loads is higher (5%) as well. We traced these bad loads to the `WAITPAUSE` synchronization primitive. These bad loads are related to a conditional execution path where the load was marked, but the related store is not always executed. In LU, most of the memory blocks are modified once and then become read only for the rest of the execution. In *DM*, all blocks are first fetched in exclusive mode which results in a high coverage. After a block has been modified, other processors will fetch it for reading. The first of these read-fetches of a block, however, will be granted an exclusive copy. This is the reason why we see a high degree of bad for *DM* in LU.

Continuing with Ocean, L and C were unable to obtain a high coverage although 22% of the cache misses should load a block in the exclusive state (see Figure 7). By contrast, S manages to achieve a coverage of 48%. We have found that this difference mainly stems from the count variable in the barrier synchronizations that cannot be covered by L and C. While S covers such occurrences, we found that S was limited by the fact that it looks for load-store sequences to the same word address; in Ocean, most undetected loads were in a code which in a simplied form is

```
for (j = L; j < U; j += 2)
   v[j] = v[j-1] + v[j+1];
```

The loop-carried load-store sequence in this code is at the block level because at least one of the vector elements (`v[j-1]` or `v[j+1]`) belongs to the same cache block as `v[j]`. This is a case where it would have paid off to let the compiler explore load-store sequences at the block level instead of at the word level. In Ocean, *DS* and *DM* have both high coverages and high degrees of bad. We speculate that the bad loads in the adaptive protocols are due to false sharing which trigger frequent reclassifications of whether a block is migratory or not.

Barnes-Hut and PTHOR are two applications with irregular data structures. For Barnes-Hut, the coverages of L and C were limited by conditional execution paths and load-store sequences at the block level, and the latter also limited S. L generates virtually no bad loads. C has a degree of bad loads of 3%, while for S it is as much as 47%. Almost all of the bad loads were generated in a code where a pointer is first read and then conditionally modified, and this occurs in the procedure `loadtree`. In C syntax, the code is in the following form:

```
if (*qptr == NULL)
   *qptr = (nodeptr) p;
```

While *DS* and *DM* have coverages (65% and 72%) similar to S, their degrees of bad (18% and 34%) are less.

Finally, none of the compiler algorithms was able to achieve a high coverage for PTHOR. Due to many load-store sequences at the block level and with conditional branches, the coverage of each algorithm was limited to 37% for L and C, and 52% for S. While both L and C generated a degree of bad loads of 3%, S generated a degree of bad loads as high as 77%. The majority of the bad loads for S are generated in the procedure `xeval` where one variable (`glob->DeadCount`) is frequently read without being modified. Since there are execution paths from these loads to stores for this variable, S uses load exclusive here. *DS* and *DM* have higher coverages (75% and 84%) than the compiler algorithms, and their degrees of bad (12% and 25%) are much less than that for S.

In summary, the compiler algorithms manage to cover a vast majority of all optimal loads for the three applications that exhibit migratory sharing. The limiting factors of even higher coverage were attributable to load-store sequences at the block level. Contrary to our beliefs, however, aliasing did not appear to be a limitation for any application based on the fact that the uncovered optimal loads are due to other reasons. As for intervening accesses, they only showed up to some extent in a synchronization primitive in LU for S. Moreover, our experiments

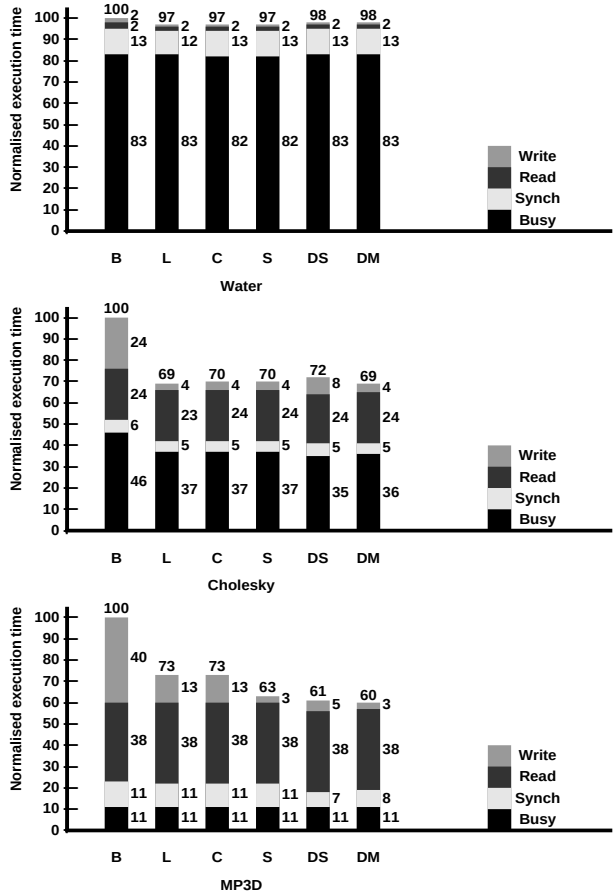

Fig. 8. Normalized execution times for the applications with migratory sharing. *B* stands for baseline. *L*, *C*, and *S* stand for the Local, Conservative, and the Speculative compiler algorithms, respectively. *DS* and *DM* stand for the default-shared and default-migratory adaptive cache coherence protocols, respectively.

with MP3D suggest that properly labeled programs with little false sharing would eliminate the impact of intervening accesses completely. Finally, the high coverages reached for applications with a fair amount of migratory sharing suggest that further improvements of our algorithms would have a minor effect on their efficiencies. One of the compiler algorithms, *S*, generated high degrees of bad loads for two applications (Barnes-Hut and PTHOR) while the adaptive protocols generated high degrees of bad loads for one application (Ocean). We will now study the effects of coverage and degree of bad loads on the execution time and traffic.

6.2 Effects on Execution Time

In this section we present the execution times for the applications for our three compiler algorithms and the two adaptive cache coherence protocols. In Figures 8 through 10, *B* denotes the normalized execution time for the baseline, and *L*, *C*, and *S* represent Local, Conservative, and Speculative, respectively. Finally, *DS* and *DM* stand for the default-shared and default-migratory adaptive cache coherence

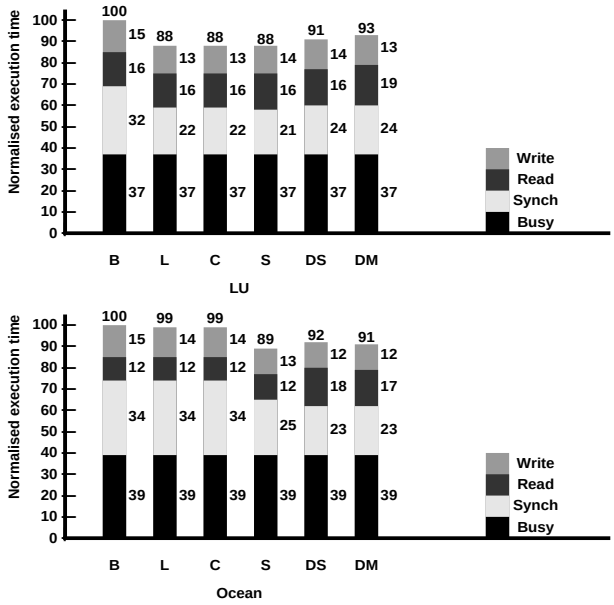

Fig. 9. Normalized execution times for LU and Ocean.

protocols, respectively. The normalized execution time for each scheme in Figures 8 through 10 is broken down into the following components from the bottom to the top: the busy time, the synchronization stall-time, the read stall-time, and the write stall-time.

We start by looking at the applications with most migratory sharing: Water, Cholesky, and MP3D. Based on the high coverages and high fractions of optimal loads in these applications, we expect to see significant reductions in the write stall-times. The write stall-times for B range from 2% in Water, 24% in Cholesky, to 40% in MP3D. When considering the compiler algorithms (L, C, and S) and the adaptive cache coherence protocols (DS and DM) we see that the write stall-times are almost completely eliminated.

Looking at the gains of the adaptive cache coherence protocols first, we would expect DM to be most effective because migratory sharing dominates in these applications. In fact, DM manages to remove all write stall-time in Water and most of the write stall-times in Cholesky and MP3D. Although DS is not as efficient as DM, it rapidly detects migratory sharing and achieves nearly identical improvements of the execution times compared to DM. The reduction of explicit ownership requests made by these schemes also helps cutting part of the synchronization time because of less contention to locks and memory. This effect is especially pronounced in MP3D, where the synchronization stall-time is reduced from 11% to 7% for DS and 8% for DM. Overall, the adaptive cache coherence protocols manage to cut at least 67% (Cholesky) of the write stall-time resulting in a reduction of the execution time by as much as 40% in MP3D. These results conform qualitatively with those presented in Stenström et al. [1993].

Continuing with the effectiveness of the compiler algorithms, we see that L and C are as effective in cutting the write stall-times in Water and Cholesky as DM—the

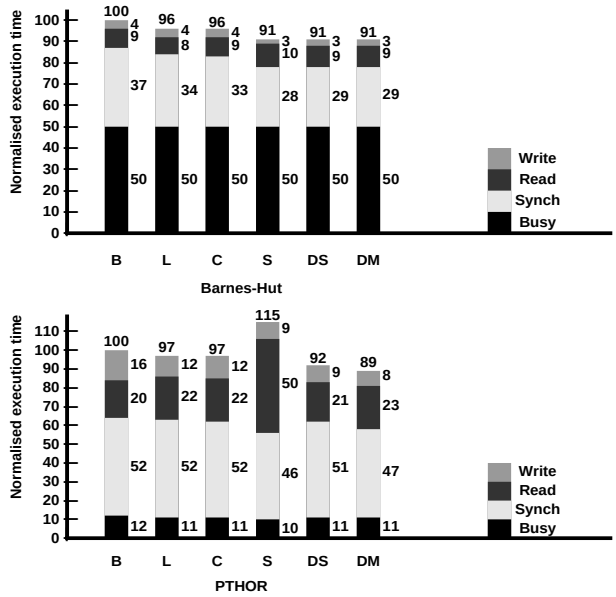

Fig. 10. Normalized execution times for Barnes-Hut and PTHOR.

adaptive cache coherence protocol that performs best. Although L and C also cut most of the write stall-time in MP3D (67%), S does even better and is virtually as effective as DM for all three applications.

We will now consider the applications with little or no migratory sharing: LU, Ocean, PTHOR, and Barnes-Hut in Figures 9 and 10. While neither our compiler algorithms nor the hardware-based mechanism are expected to improve the performance much, we are interested in seeing the effects of the degrees of bad loads that we observed in Figure 7. We make the following observations. The adaptive algorithms increase the read stall-time by more than 40% for Ocean. Neither L nor C have any significant effect on the execution times; however, S more than doubles the read stall-time in PTHOR due to useless invalidations, and this results in an increased execution time.

In summary, we note that removing the ownership overhead has a dramatic impact on the execution times for two of the seven applications. Our most aggressive compiler algorithm (S) does better or the same as both adaptive cache coherence protocols for six of the seven applications, but for one (PTHOR) it increases the execution time significantly. This suggests that L and C seem to be the best choice across the applications studied.

6.3 Effects on Traffic

We present in this section how memory traffic is affected for each algorithm. A reduction in traffic comes from reducing the number of control messages while an increase is due to additional cache misses created by useless invalidations. Recall that a control message is 5 bytes, and a data message is 37 bytes.

Figure 11 shows the traffic for each application and algorithm normalized to that of the baseline system (100%). Starting with Water, Cholesky, and MP3D, we see a

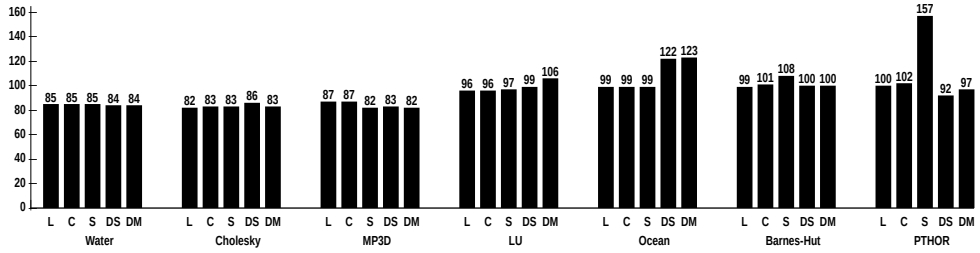

Fig. 11. Normalized traffic relative to the baseline.

significant reduction in traffic for all five algorithms. Since the degrees of bad loads in these applications are low, and there are more than 80% optimal loads, we can see traffic reductions that correspond to the coverages. For Water, the compiler algorithms reduce the traffic by 15% and the adaptive protocols by 16%. The lower coverage of *DS* for Cholesky results in a traffic reduction of 14% while *C*, *S*, and *DM* reduces the traffic by 17%, and *L* reduces it by 18%. Finally for MP3D, the lower coverages of *L* and *C* result in traffic reductions of 13% only while *S* and *DM* reduce the traffic by 18% and *DS* by 17%.

Continuing with the applications with little migratory sharing (LU, Ocean, Barnes-Hut, and PTHOR), we find that *S*, *DS*, and *DM* each increase the traffic by more than 20% for one application: *S* by 57% for PTHOR, *DS* by 22%, and *DM* by 23% for Ocean. The additional traffic is due to the high degrees of bad loads as observed in Section 6.1. The impact of the increased traffic on the execution time depends on the bandwidth provided by the interconnection network—for instance, the performance of bus-based multiprocessors could be hurt because it is often limited by the bandwidth of the bus. Although we have measured the total traffic over the application execution (and not the bandwidth requirements), our measurements show that only *L* and *C* generate traffic that is either below or only marginally above that generated by the baseline; *C* increases the traffic for Barnes-Hut by 1% and for PTHOR by 2%.

7. DISCUSSION AND RELATED WORK

The success of static analysis by the compiler is of course dependent on the programming style in an application. We do not know how representative our set of application programs are in respect to actually helping the compiler to do accurate dataflow analysis, but for these applications it is clear that standard intraprocedural dataflow analysis indeed is very useful. Since we have used parallel applications that are not annotated with primitives that reveal expected use of data, we feel that annotations such as those proposed in Cooperative shared memory [Hill et al. 1992] are not needed to reduce ownership overhead effectively using static analysis.

In Mowry et al. [1992], a compiler algorithm for nonbinding prefetching based on locality analysis is presented, and Mowry extended it for shared-memory multiprocessors by taking synchronizations into account when computing the localized iteration space of a loop [Mowry 1994]. Their algorithm does nonbinding prefetching in read-only or read-exclusive mode. These algorithms are very effective in

hiding the memory latency for array references that are linear functions of the loop indices, since for such references it is possible to calculate the address and issue prefetch instruction sufficiently ahead of time through software pipelining. In the general case, however, it can be difficult for the compiler to determine which references will miss in the cache.

Compared with Mowry's work on compiler-controlled software prefetching, this article focuses on a related but separate problem. A shared-memory multiprocessor with support for nonbinding software-controlled prefetching should, in our opinion, also support a load instruction capable of acquiring ownership. Our argument is that in codes where the compiler is unable to determine whether a reference to shared memory will create a cache miss or not, it is necessary to restrict prefetching in order to avoid instruction overhead should the reference hit without a prefetch. Our algorithms do not add any instruction overhead, and this article focuses on (1) the problem of determining which load instructions should be marked as ownership requests and (2) the effects on the execution time and traffic of this optimization. Marking of loads also should be restricted because otherwise the compiler would create additional invalidations and misses which would degrade performance. The local and conservative compiler algorithms consistently improved performance by reducing ownership overhead while only marginally creating useless cache misses.

We studied the impact of removing ownership overhead on the execution time and traffic using a detailed architectural simulator and found its effect to be dramatic in some cases. A limitation of the simulations presented in this article, however, stems from the lack of variation of key architectural choices such as memory consistency model, latency numbers, as well as cache parameters. Based on available data from other studies combined with some key observations in this study, it is possible to predict how the results would have been affected by variations in such design choices.

To start with, if a relaxed memory consistency model would have been used, the gains of the algorithms seem at first glance nullified because the write latency can then be completely hidden [Gharachorloo et al. 1991]. However, the traffic reductions we have observed in this study can cut the read and synchronization penalties under relaxed memory consistency models because of less contention in the memory system.

Continuing with the effect of varying the memory system latencies, we first note that the latency assumptions in this study are similar to those found in machines such as the DASH [Lenoski et al. 1992], although DASH services cache misses in three node-to-node traversals in situations where our protocol handles them in four. However, if we were to consider more aggressive processors than we have assumed in this study such as the DEC Alpha, the impact of ownership overhead reduction on the execution time would be more dramatic than our numbers suggest.

Cox and Fowler [1993] studied the effect of cache parameters on the effectiveness of the adaptive techniques. Overall they found that if false sharing becomes dominant when the block size increases, the effectiveness of the detection goes down. We have reasons to believe that this is true for our compiler algorithms as well because they do not take into account intervening accesses from other processors. Considering limited caches, replacements can reduce the latency of each individual cache operation because blocks are more often uncached in the home node. In Cox

and Fowler [1993] it was found that the relative efficiency of the adaptive techniques was smaller for limited caches, and we expect this to hold for our compiler algorithms as well.

Finally, in Dahlgren et al. [1995] the speculative compiler algorithm was evaluated in the context of a bus-based multiprocessor. It was found that the number of busy bus cycles could be reduced by 31% for Water, 33% for Cholesky, 29% for MP3D, and 3% for Ocean. Based on the measurements in this article, we expect that the local and conservative algorithms are robust techniques to reduce the number of busy bus cycles in bus-based multiprocessors.

While this article focuses on reducing ownership overhead and write stall-time, in another work we have studied the potential of using classic dataflow techniques in a compiler algorithm to reduce read stall-time [Skeppstedt and Stenström 1995]. That work extends the concept of store-classes to the cache block level and studies compiler-inserted memory update instructions for irregular data structutes. The evaluation in that paper shows that the read stall-time could be reduced significantly for applications with migratory sharing.

8. CONCLUSIONS

In this article we have presented the design and evaluation of three compiler algorithms that aim at reducing the ownership overhead associated with write-invalidate cache coherence protocols. These algorithms use dataflow analysis techniques to detect load-store sequences using the same address. Loads belonging to such sequences are marked and provide a hint to the cache to acquire an exclusive copy of the block. Thus if a marked load misses in the cache, the ownership acquisition can be coalesced with the request for the missing block.

The simplest algorithm detects load-store sequences within each basic block only, whereas the two other algorithms analyze the existence of such sequences across basic blocks at the intraprocedural level. While the second algorithm conservatively requires that the subsequent store is executed, the third algorithm relaxes this condition by only requiring that one possible execution path contains the store. The advantages of the algorithms are that they are based on classic intraprocedural dataflow analysis similar to live-variable analysis; they add very little to the complexity of the analysis already done in optimizing compilers.

In our evaluation of the detection efficiency of the compiler algorithms, we have found that the compiler algorithms detected a clear majority of the load-store sequences that gave rise to ownership overhead in applications with migratory sharing. Our data suggest that dataflow analysis techniques can advantageously be used to reduce coherence overhead found in state-of-the-art multiprocessors systems. While the most aggressive algorithm consistently has a higher coverage than the other algorithms, it can fail and generate more cache misses in some cases. It increased the execution time for one application. We envision that branch-profiling information can be used to reduce the degree of bad for the most aggresive algorithm; however, we believe that the performance effects of this algorithm without improvement are unstable.

To gain further understanding about the usefulness of static compiler algorithms, we compared them with a dynamic hardware-based mechanism and found that they achieve similar performance improvements. We found that the dynamic mechanisms

increased the read stall-time for one application, which shows that they are not able to handle all application sharing behaviors effectively.

Of the five algorithms we evaluated, both hardware and software, it was the local and conservative compiler algorithms that showed the most consistent performance improvements. This indicates that classic dataflow analysis is a suitable technique to reduce ownership overhead in cache coherent multiprocessors.

ACKNOWLEDGEMENTS

This work was mostly done while the authors were at Lund University, and we would like to thank our former colleagues Mats Brorsson, Fredrik Dahlgren, and Håkan Grahn at the Department of Computer Engineering of Lund University, Mohd Hanafiah Abdullah of MIMOS in Kuala Lumpur, and Magnus Karlsson of Chalmers University of Technology for valuable discussions on this research and numerous suggestions on how to improve earlier drafts of this article.

REFERENCES

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., and Yeung, D. 1995. The MIT Alewife Machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ACM, New York, 2–13.

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

Boyle, J., Butler, R., Diaz, T., Glickfield, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R. 1987. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, New York.

Brorsson, M., Dahlgren, F., Nilsson, H., and Stenström, P. 1993. The CacheMire Test Bench — A flexible and effective approach for simulation of multiprocessors. In *Proceedings of the 26th IEEE Annual Simulation Symposium*. IEEE, New York, 41–49.

Censier, L. and Feautrier, P. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. 27,* 12, 1112–1118.

Chow, F. and Hennessy, J. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12,* 4, 590–536.

Cox, A. and Fowler, R. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, Calif., 98–108.

Dahlgren, F., Skeppstedt, J., and Stenström, P. 1995. Effectiveness of hardware-based and compiler-controlled snooping cache protocol extensions. In *Proceedings of the International Conference on High Performance Computing 1995*. Tata McGraw-Hill, New Delhi, 87–92.

Dubois, M., Skeppstedt, J., and Stenström, P. 1995. Essential misses and data traffic in coherence protocols. *J. of Parallel Distrib. Comput. 29,* 2, 108–125.

Galles, M. and Williams, E. 1994. Performance optimizations, implementation, and verification of the SGI Challenge Multiprocessor. In *Proceedings of the 27th Hawaii International Conference on System Sciences*. Vol. 1. IEEE, New York, 134–143.

Gharachorloo, K., Gupta, A., and Hennessy, J. 1991. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 245–257.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*. ACM, New York, 15–26.

Gupta, A. and Weber, W.-D. 1992. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Trans. Comput. 41,* 7, 794–810.

Hill, M., Larus, J., Reinhardt, S., and Wood, D. 1992. Cooperative shared memory: Software and hardware support for scalable shared memory . In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 262–273.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28,* 9, 690–691.

Lenoski, D., Laudon, J., Gharachorlo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. 1992. The Stanford DASH Multiprocessor. *IEEE Comput. 25,* 3 (Mar.), 63–79.

Mowry, T. 1994. Tolerating latency through software-controlled data prefetching. Ph.D. thesis, Computer Systems Laboratory, Stanford Univ., Stanford, Calif.

Mowry, T. and Gupta, A. 1991. Tolerating latency through software-controlled prefetching in scalable shared-memory multiprocessors. *J. Parallel Distrib. Comput. 2,* 4, 87–106.

Mowry, T., Lam, M., and Gupta, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 62–73.

Singh, J. P., Weber, W.-D., and Gupta, A. 1992. SPLASH: Stanford Parallel Applications for Shared-Memory. *Comput. Arch. News 20,* 1, 5–44.

Skeppstedt, J. 1990. The design and implementation of an optimizing ANSI C compiler for SPARC, Tech. Rep., Dept. of Computer Science, Lund Univ., Lund, Sweden.

Skeppstedt, J. and Stenström, P. 1994. Simple compiler algorithms to reduce ownership overhead in cache coherence protocols. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 286–296.

Skeppstedt, J. and Stenström, P. 1995. A compiler algorithm that reduces read latency in ownership-based cache coherence protocols. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques 1995*. IFIP, Laxenburg, Austria, 69–78.

Stenström, P. 1990. A survey of cache coherence schemes for multiprocessors. *IEEE Comput. 23,* 6 (June), 12–24.

Stenström, P., Brorsson, M., and Sandberg, L. 1993. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, Calif., 109–118.

Received September 1995; revised June 1996; accepted July 1996