Efficient Integration of Compiler-directed Cache Coherence and Data Prefetching *

Hock-Beng Lim †
Center for Supercomputing R & D
University of Illinois
Urbana, IL 61801
hblim@csrd.uiuc.edu

Pen-Chung Yew
Dept. of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
yew@cs.umn.edu

Abstract

Cache coherence enforcement and memory latency reduction and hiding are very important and challenging problems in the design of large-scale distributed shared-memory (DSM) multiprocessors. We propose an integrated framework to solve these problems through a compiler-directed cache coherence scheme called the Cache Coherence with Data Prefetching (CCDP) scheme. The CCDP scheme enforces cache coherence by prefetching the potentially stale references in a parallel program. It also prefetches the nonstale references to hide their memory latencies. To optimize the performance of the CCDP scheme, some prefetch hardware support is provided to efficiently handle these two forms of data prefetching operations.

We also developed the compiler techniques utilized by the CCDP scheme for stale reference detection, prefetch target analysis and prefetch scheduling. We evaluated the performance of the CCDP scheme via execution-driven simulations of several applications from the SPEC CFP95 and the Perfect benchmark suites. The simulation results show that the CCDP scheme provides significant performance improvements for the applications studied.

Keywords : Compiler-directed Cache Coherence, Data Prefetching, Memory System Design, Shared-memory Multiprocessors

1. Introduction

In the recent years, large-scale distributed shared-memory (DSM) multiprocessors have emerged as a promising architecture to deliver high performance computing power. However, to fully realize the potential performance of these systems, their cache coherence schemes should be efficient, inexpensive and scalable. Also, it is necessary to develop efficient techniques to hide the large remote memory access latencies in such systems.

The cache coherence techniques used in existing multiprocessors available commercially are mainly hardware-based, such as a *snoopy cache protocol* or a hardware *directory-based* scheme. In large-scale DSM systems, the scalability of such schemes might be affected by coherence-related network traffic and the complexity of the coherence hardware. *Compiler-directed cache coherence* schemes [5] offer an alternative approach to solve the cache coherence problem. Their key attractive feature is that the cache coherence actions are performed locally by each processor as specified by the compiler, without the need for interprocessor communications. They also do not require complicated and expensive hardware directories.

Several techniques have been proposed in the literature to address the memory latency problem in multiprocessors. In particular, *data prefetching* is an effective technique to hide memory latency and to improve memory system performance. Hardware and software support for data prefetching are already provided in several experimental and commercially-available multiprocessors.

In fact, compiler-directed cache coherence and data prefetching schemes can be combined in a natural and complementary manner to provide better overall performance. We have developed a compiler-directed cache coherence scheme which effectively integrates data prefetching. The *Cache Coherence with Data Prefetching (CCDP)* scheme uses compiler analyses to identify potentially stale data references in a parallel program, and then enforces cache coherence by prefetching up-to-date data corresponding to these references from the main memory. Prefetching the potentially stale references implicitly hides the memory latencies of fetching remote up-to-date shared data. The CCDP

*This work is supported in part by the National Science Foundation under Grants MIP 93-07910, MIP 94-96320, CDA 95-02979 and MIP 96-10379. Additional support is provided by a gift from Cray Research, Inc and by a gift from Intel Corporation.

†Currently with the Hewlett-Packard Company, Cupertino, California.

scheme further optimizes performance by prefetching the nonstale data references and hiding their memory latencies.

To optimize the performance of the CCDP scheme, we designed some inexpensive prefetch hardware support to handle its two forms of data prefetching operations. We also developed compiler techniques to perform stale reference analysis, prefetch target analysis and prefetch scheduling for the CCDP scheme. Finally, we evaluated the performance of the CCDP scheme through execution-driven simulations of several benchmark programs from the SPEC CFP95 and the Perfect suites. Our experimental results indicate that the CCDP scheme provides significant performance improvements for the programs studied.

The rest of the paper is organized as follows. Section 2 discusses our framework for integrating compiler-directed cache coherence and data prefetching. The prefetch hardware support to optimize the CCDP scheme is discussed in Section 3. In Section 4, we focus on the compiler support for the CCDP scheme. Section 5 presents the experimental study to evaluate the performance of the CCDP scheme. Finally, we conclude the paper in Section 6.

2. The CCDP Framework

2.1. Background and Motivation

When shared data are cached during execution of a parallel program, multiple copies of a shared memory location might exist in the caches of the processors. Cache coherence schemes must ensure that the processors always access the most up-to-date copies of shared data. The remaining invalid copies are known as *stale data*, and the references to these data are called *stale references*.

Most hardware-based cache coherence schemes prevent stale references at run time by invalidating or updating stale cache entries right before the stale references occur. To do so, they require interprocessor communications to keep track of the cache states and to perform these cache coherence actions. Such coherence-related network traffic can use up excessive amount of interconnection network bandwidth of the system. Also, the complexity and hardware cost of a directory-based cache coherence scheme for large-scale multiprocessors could be substantial.

In contrast, compiler-directed cache coherence schemes [5] can detect the potentially stale references using compiler analyses at compile time. The compiler then directs the processors to perform cache coherence actions such as cache invalidations locally, without the need for interprocessor communications. This reduces the traffic load on the interconnection network, and eliminates the need to use hardware directories to keep track of run-time cache states. As a result, compiler-directed cache coherence schemes have better scalability and lower hardware cost than hardware directory schemes. The drawback of these schemes is that there might be situations when the compiler has to adopt conservative decisions. Therefore, the cache coherence actions generated by the compiler-directed schemes might not be as accurate as those of the hardware directory schemes.

Although cache coherence schemes can improve multiprocessor cache performance, they cannot completely eliminate remote memory accesses. Data prefetching is a well established technique to hide memory latencies. In hardware cache-coherent systems, data prefetching is usually a separate optimization which is not tightly integrated with the underlying cache coherence scheme. The data prefetching operations are determined based on data locality considerations alone. Also, the stale and nonstale data references are treated uniformly by the prefetching schemes.

The CCDP scheme efficiently integrates compiler-directed cache coherence and software-initiated data prefetching. It does not make use of special cache coherence hardware. However, it requires system-level prefetch hardware which is less costly than cache coherence hardware. The CCDP scheme is well-suited for loop-parallel codes with mainly regular array data accesses. Typical examples of such codes include the large-scale scientific programs which form the dominant workload in large-scale multiprocessors. These codes can be analyzed with decent accuracy using the existing compiler technology. However, with future advances in compiler technology such as pointer analysis, the CCDP scheme can be applied to programs which have irregular data accesses.

2.2. Overview of the CCDP Scheme

The CCDP scheme consists of three major steps. First, during the *stale reference analysis* step, the compiler identifies the potentially stale and nonstale data references in a parallel program by using several program analysis techniques. However, it might not be necessary or worthwhile to prefetch all of the references in these two categories. Thus, the *prefetch target analysis* step determines which of those references in the two categories should be prefetched. Finally, the *prefetch scheduling* step schedules the prefetch operations and inserts them at appropriate locations in the program. The compiler also inserts cache and memory management operations such as cache invalidations and *bypass-cache fetch* operations, which bypass the cache and access the main memory directly, when necessary.

Our scheme makes use of *vector prefetches* and *cache-line prefetches*. In vector prefetches, a block of data with a fixed stride is fetched from the main memory. The amount of data being fetched by cache-line prefetches is equal to the size of a cache line. In theory, vector prefetches should reduce the prefetch overhead by amortizing the fixed initiation costs of several cache-line prefetches.

2.3. Data Prefetching Optimizations

The CCDP scheme uses two types of data prefetching optimizations which we call *coherence-enforcing prefetch (ce-prefetch)* and *latency-hiding prefetch (lh-prefetch)*. For the *ce-prefetch* operations, a key task is to ensure that the correctness of the memory references of a program is not violated by prefetching the potentially stale references. First, the *ce-prefetch* operations should respect all control and data dependence constraints. Second, after the *ce-prefetch* operations are issued, the processors will access the prefetched up-to-date data instead of the potentially stale data in the caches. This can be achieved with the help of special prefetch hardware support which enables the *ce-prefetch* operations to invalidate the cache entries corresponding to the potentially stale references. Without such prefetch hardware, the compiler has to insert explicit cache invalidation instructions to invalidate the cache entries before the prefetches are issued. On the other hand, since the *lh-prefetch* operations are used for prefetching the nonstale references, they will not violate cache coherence.

It is necessary to prioritize the two types of prefetches used by the CCDP scheme in order to optimize its performance. Since it is more important to prefetch the potentially stale references and maintain cache coherence as efficiently as possible, the scheme assigns higher priority to the *ce-prefetch* operations. Again, specific hardware support can be designed to issue *ce-prefetch* operations at a higher priority over those of the *lh-prefetch* operations at run time.

3. Hardware Support

A major factor which affects the performance of the CCDP scheme is the effectiveness of the hardware support for prefetching on the system. The CCDP scheme can make use of the prefetch hardware on existing systems. In fact, we have previously implemented the scheme on the Cray T3D [8]. However, the conventional prefetch hardware of the Cray T3D is not optimized to handle the two types of data prefetching operations used by the CCDP scheme.

Various researchers have developed sophisticated hardware prefetching schemes which make use of hardware features to dynamically predict the data references to prefetch at run-time. However, these prefetch hardware designs are not suitable for the CCDP scheme because they cannot distinguish between potentially stale and nonstale references and take proper actions to enforce cache coherence. Thus, we propose some inexpensive and efficient prefetch hardware support for the CCDP scheme.

3.1. Architectural Model

In this paper, we assume a large-scale non-cache-coherent DSM multiprocessor. The processing nodes are interconnected by a high-bandwidth network. This architecture is similar to that of the Cray T3D. The organization of a node is shown in Figure 1.

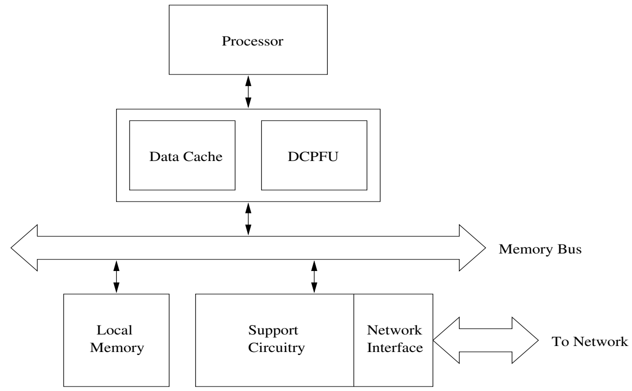

Figure 1. Organization of a Processing Node

Each node consists of a processor, a data cache and its associated *Data Coherence Prefetch Unit (DCPFU)*, a local memory module, and various support circuitry. The collection of the local memory modules in all of the nodes form a logically shared global memory. The support circuitry on each node contains address translation logic which transforms a virtual address into its physical address. It also supports messaging and synchronization activities on the system such as remote memory access, barrier synchronization, etc. With the support circuitry, a processor can directly access the memory of another processor without involving the remote processor.

The data cache and the DCPFU are tightly coupled. For simplicity, we assume that there is only one level of cache hierarchy in the node. The cache is lock-up free, i.e., it can handle multiple outstanding cache misses. The processor can issue two types of prefetch instructions : a cache-line prefetch and a vector prefetch.

3.2. Organization of the DCPFU

The DCPFU consists of a *Prefetch Control Unit (PCU)*, a *Priority Prefetch Queue (PPQ)* and a *Prefetch Holding Cache (PHC)*. The DCPFU can handle multiple outstanding prefetch requests. In this paper, we assume that the DCPFU can handle up to 32 outstanding prefetch requests. Figure 2 shows the organization of the DCPFU.

3.2.1. The Prefetch Holding Cache (PHC)

The PHC is a small, fully-associative cache with 32 cache lines. The PHC holds the up-to-date data corresponding to potentially stale references which are fetched from the memory by *ce-prefetch* operations. On the other hand, non-

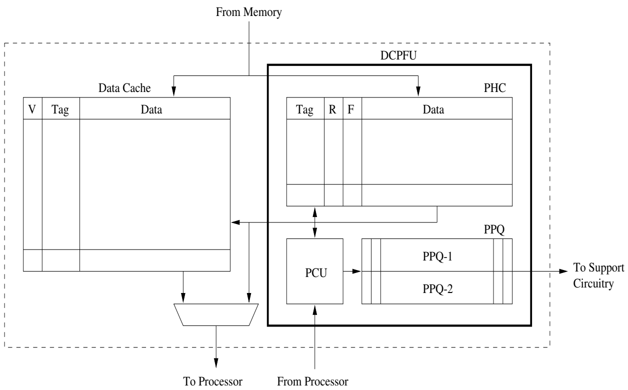

Figure 2. Organization of the DCPFU

stale data fetched by *lh-prefetch* operations are stored directly into the data cache.

The processor checks both the PHC and the data cache simultaneously. If the prefetched data has arrived in the PHC, the processor loads the data from it. At the same time, the PHC cache lines containing these prefetched data will be moved into the data cache. By reading the data for *ce-prefetch* operations from the PHC, the processor will not access potentially stale data in the data cache. Furthermore, since the prefetched up-to-date data are temporarily buffered in the PHC, the likelihood that the data cache will be polluted by these prefetched data is reduced.

Each PHC cache line contains the following fields : a tag, a *reserved bit*, a *full bit*, and the data. The tag is used to determine if there is a PHC hit. The reserved bit indicates whether the PHC cache line has already been reserved by an earlier prefetch request. The full bit indicates whether the prefetched data for the PHC cache line has arrived from the memory. Before a prefetch request is issued, the PCU has to reserve an available PHC cache line by filling its tag field and setting the reserved bit. Thus, a PHC cache line with the reserved and full bits set is one which holds prefetched data which have arrived but have not yet been accessed by the processor. After the processor accesses such a PHC cache line, its reserved bit will be reset and its content will be moved to the data cache.

3.2.2. The Priority Prefetch Queue (PPQ)

The PPQ queues the prefetch operations which are to be issued. It is divided into two sections, PPQ-1 and PPQ-2, each having 16 entries. Each PPQ entry holds a tag to identify the prefetch operation, which is simply the memory address to be prefetched. PPQ-1 is used for *ce-prefetch* operations, while PPQ-2 is used for *lh-prefetch* operations. The PCU inserts the prefetch requests into the PPQ in the order specified by their priority levels. The prefetch requests in the PPQ-1 will be issued first in FIFO order, followed by the prefetch requests in PPQ-2 in FIFO order.

3.2.3. The Prefetch Control Unit (PCU)

The PCU is responsible for processing prefetch operations issued by the processor and keeping track of the outstanding prefetch requests. When the PCU accepts a cache-line prefetch operation, it generates a request in the relevant section of the PPQ, depending on whether the prefetch operation is a *ce-prefetch* or *lh-prefetch*. As for a vector prefetch operation, the PCU generates individual cache-line prefetch requests based on the starting address, stride, and length of the vector to be prefetched. This relieves the processor from explicitly issuing multiple cache-line prefetches and paying for their initiation overheads.

For each prefetch operation, the PCU checks if there is already an outstanding prefetch request in the PPQ, or if the cache line has already been prefetched and it is still stored in the PHC. In this way, the PCU merges multiple prefetch requests to the same cache line and reduces the number of unnecessary prefetch requests. The PCU also maintains the status and contents of the PHC cache lines. In particular, it reserves a PHC cache line for each issued prefetch request by setting the reserved bit. When the prefetched data arrives from the memory, the PCU sets the full bit of the reserved PHC cache line. It resets the reserved bit after the PHC cache line has been accessed by the processor and copied to the data caches.

4. Compiler Techniques

The performance of the CCDP scheme is also strongly influenced by the effectiveness of its compiler support. Several compiler techniques have been developed for software-initiated data prefetching [2, 6, 9]. However, as these data prefetching schemes are used solely for memory latency hiding, the data prefetching operations are determined based on data locality considerations alone. Since these techniques do not distinguish between potentially stale and nonstale references, they cannot be applied directly in the CCDP scheme. Our compiler techniques for the CCDP scheme are designed to address this limitation.

4.1. Stale Reference Analysis

Three main program analysis techniques are used in stale reference analysis : *stale reference detection*, *array dataflow analysis*, and *interprocedural analysis*. We make use of extensive algorithms [4, 5] which were previously developed for these techniques. Since the detailed algorithms are described in [4], we will only state the functions of the stale reference analysis techniques.

To find the potentially stale data references, it is necessary to detect the memory reference sequences which might violate cache coherence. The stale reference detection algorithm [4] accomplishes this by performing data-flow analysis on the epoch flow graph of the program, which is a modified form of the control flow graph. The algorithm uses two locality preserving analysis techniques to exploit some temporal and spatial reuses of memory locations in the program. It also uses array data-flow analysis [4], which treats different regions of arrays as distinct symbolic variables, to refine the stale reference detection. Finally, we use interprocedural analysis [4] to enable the CCDP scheme to exploit locality across procedure call boundaries.

4.2. Prefetch Target Analysis

As the prefetch operations introduce instruction execution and network traffic overhead, it is important to minimize the number of unnecessary prefetches. Our prefetch target analysis algorithm [8] focuses on the inner loops, where prefetching is most likely to be beneficial. Those potentially stale references which are not in the inner loops will not be prefetched. Instead, they are issued as bypass-cache fetch operations, which fetch the up-to-date data directly from the memory to preserve program correctness.

The algorithm also exploits spatial reuses to eliminate some unnecessary prefetch operations. If cache-line prefetch operations are used for a reference that has self-spatial reuse [12], then we can unroll the inner loop such that all copies of the reference access the same cache line during an iteration. In this manner, only one cache-line prefetch operation is needed to bring in data for all copies of the reference. In addition, for a group of references which exhibit *group-spatial reuse* [12], we only need to prefetch the *leading reference* [12] of the group. The other references in the group are issued as normal reads.

Finally, note that the potentially stale references which need not be prefetched due to locality exploitation will be treated as normal read references. This is because the up-to-date data will be brought into the caches by other *ce-prefetch* operations, and thus program correctness will be preserved. Similarly, the nonstale references which are eliminated from the set of prefetch targets are simply issued as normal read operations.

4.3. Prefetch Scheduling

Our prefetch scheduling algorithm makes use of three scheduling techniques : *vector prefetch generation*, *software pipelining*, and *moving back prefetches*. We will summarize the key aspects of these techniques as the details are discussed in [8].

In vector prefetch generation, the array references in each loop are examined to see if they could be pulled out of the loop. A vector prefetch operation can then be generated for these references if the loop is serial or if the loop is parallel and the loop scheduling strategy is known at compile time. Our algorithm pulls out an array reference one loop level at a time. It then constructs a vector prefetch operation and checks if the number of words to be prefetched will exceed the available prefetch queue size or the cache size. The vector prefetch operation will be generated only if these hardware constraints are satisfied and if the array reference should not be pulled further out.

We use software pipelining to schedule cache-line prefetch operations for inner loops that do not contain recursive procedure calls. Our algorithm uses a compiler parameter to specify the range of the number of loop iterations which should be prefetched ahead of time. The algorithm also takes the hardware constraints into consideration by not issuing the prefetches when the amount of data to be prefetched exceeds the available prefetch queue size or the cache size. If neither vector prefetch generation nor software pipelining can be applied for a particular loop, then our algorithm attempts to move back the prefetch operations from the point where the data will be used, subject to control and data dependence constraints. To maximize the effectiveness of the prefetches, our algorithm uses a parameter to decide whether to move back a prefetch operation. The range of values for this parameter indicates the suitable distance to move back the prefetches.

Our prefetch scheduling algorithm [8] considers each inner loop or serial code segment of the program. Those potentially stale prefetch targets which will not be issued due to hardware constraints are replaced by bypass-cache fetches to preserve program correctness. On the other hand, the nonstale prefetch targets which are not issued are normal read operations. Depending on the type of loop or code segment, the algorithm uses a suitable scheduling technique for the prefetch targets. Loop unrolling is performed before software pipelining is applied to a loop if the loop contains prefetch targets which exhibit self-spatial locality.

4.4. Compiler Implementation

We have developed a prototype implementation of the algorithms based on the Polaris parallelizing compiler [10]. Our CCDP compiler prototype takes a sequential Fortran program as input. First, it performs the standard optimizations and parallelization passes in Polaris. Some of the classic optimizations are constant folding, constant propagation, and dead code elimination. The parallelization passes include data dependence analysis, inlining, induction variable substitution, reduction recognition, and array privatization. These optimizations and transformations produce information on the parallel loops and the shared and private data in the loops which will be used by the later passes.

After the parallelization is completed, the compiler performs the stale reference analysis algorithm to mark the potentially stale and nonstale references in the parallel program. We use a previous implementation of the stale reference analysis algorithm in Polaris [4]. The prefetch scheduling step then schedules the prefetch target references. After the prefetch scheduling step, the compiler generates the cache coherence and prefetch operations. These include the cache-line and vector *ce-prefetch* and *lh-prefetch* operations and the bypass-cache fetch operations. Finally, the compiler generates code for our simulator.

5. Performance Evaluation

5.1. Simulation Environment

In our experiments, we use the EPGsim [11] execution-driven simulator to evaluate the performance of several cache coherence schemes in a uniform manner. EPGsim uses source-level instrumentation to generate the events for simulation such as local and global memory references, parallel loop startup and scheduling operations, and synchronization operations. It uses a static cyclic scheduling policy to distribute parallel loop iterations to the processors.

We can customize the EPGsim cache simulator to model various compiler-directed or directory-based cache coherence schemes. The cache models maintain the cache states, compute cache and memory access delays, and collect execution statistics. We also model prefetch hardware support in conjunction with the cache models. EPGsim uses an analytical network model [7] to simulate the network delays and remote memory access latencies.

5.2. System Model

We model a 32-processor, non-cache-coherent DSM multiprocessor which is similar to the Cray T3D. The nodes of the system are interconnected via a multistage network. The memory system provides a cache hit latency of 1 cycle, while the base memory latency for a cache miss is 100 cycles if there is no network contention. Each processor is single-issue and all the arithmetic and logical instructions are assumed to take 1 cycle.

Each processor has a lock-up free data cache. We model a 64-Kbyte direct-mapped cache for each processor. The cache line size is 32 bytes. For the compiler-directed cache coherence schemes evaluated, a write-through with write-allocate policy is used. According to previous analysis, a write-through policy provides better performance than a write-back policy for compiler-directed cache coherence schemes [3]. On the other hand, for a hardware directory scheme, a write-back policy should deliver better performance than a write-through policy. Thus, we use a write-back policy for the hardware directory scheme which we evaluated in this study.

For a system which uses the CCDP scheme, we model a DCPFU in each of its processors. The PPQ of each DCPFU has two sections, each with 16 entries. The PCU inserts prefetch requests into the PPQ until it is full. When that occurs, it will wait until some requests have been issued, which frees up entries on the PPQ. The PHC has 32 entries, each with line size of 32 bytes. It is fully-associative, with a hit latency of 1 cycle if the prefetched up-to-date data has already arrived. If the prefetched data for a PHC entry has not arrived, i.e., the full bit of the entry has not been set, then the processor will wait for the prefetched data to arrive.

5.3. Simulated Schemes

In addition to the CCDP scheme, we also simulated three cache coherence schemes. First, the **BASE** scheme is the baseline scheme which does not cache shared data. All the shared memory references will be directed at the main memory. The BASE architecture is similar to that of the Cray T3D. Due to the lack of hardware cache coherence mechanisms, the Cray T3D does not cache shared data in the Cray MPP Fortran (CRAFT) programs. However, the BASE scheme and the other cache coherence schemes do cache private or nonshared data. We model the BASE architecture to provide a performance comparison with a large-scale non-cache-coherent DSM system.

Second, the Cache Bypass (**CBP**) scheme is a pure software-controlled cache coherence scheme which can be implemented on existing large-scale DSM systems without requiring additional cache coherence hardware. It performs stale reference analysis to detect the potentially stale references. Then, it uses bypass-cache fetch operations to bypass the cache and directly access up-to-date data in the main memory. In this manner, the processors will not access potentially stale data in their caches.

Finally, the **HWD** scheme uses a full-map hardware directory [1] with a standard three-state (*invalid, read-shared, write-exclusive*) invalidation-based coherence protocol. The directories are distributed across the nodes and are organized as pointer caches to reduce storage. We augment the HWD scheme with software-controlled data prefetching so as to provide a fair comparison with the CCDP scheme. However, the HWD scheme cannot differentiate between potentially stale and nonstale data references, and it does not use the prefetch hardware support for the CCDP scheme. Instead, it prefetches data directly into the data cache.

5.4. Experimental Methodology

First, we automatically parallelize the application codes using the Polaris compiler [10]. We only invoke the standard parallelization techniques supported by Polaris. With the parallelism and data sharing information of the parallelized codes, we can simulate their execution under the BASE scheme.

For the CBP and CCDP schemes, we use the compiler to automatically insert cache coherence operations to the application codes. For the CBP scheme, the potentially stale references marked by the compiler are treated as bypass-cache fetch operations by the simulator. For the CCDP scheme, we automatically add prefetch operations into the programs using our Polaris implementation of the prefetch target analysis and prefetch scheduling algorithms. For the HWD scheme, the compiler inserts prefetch operations using the software pipelining technique.

After obtaining the parallel programs with the required cache coherence operations, we instrument these programs [11] to generate events that model the program execution on our target architecture. Finally, we run these instrumented codes through the EPGsim simulator and measure the performance results.

For our experimental study, we selected two programs from the SPEC CFP95 benchmark suite and three programs from the Perfect benchmark suite as our workload. The two applications codes from the SPEC CFP95 suite are TOMCATV and SWIM. The three programs from the Perfect suite are TRFD, FLO52, and QCD. For all of the applications, we use their default problem sizes as specified by the benchmark suites. These applications are all floating-point intensive and they are written in Fortran.

5.5. Performance Results

5.5.1. Performance of cache coherence schemes

We use three performance metrics in this study, namely, cache miss ratio, network traffic, and simulated execution time of the programs.

Cache miss ratio The cache misses consist of *sharing* and *nonsharing* misses. A sharing miss occurs when the cached data item has been invalidated. The other types of usual cache misses such as cold, conflict and capacity misses are considered as nonsharing misses. The sharing misses can be divided into *true sharing* and *false sharing* misses. True sharing misses arise due to the need to enforce cache coherence. On the other hand, the false sharing misses are unnecessary cache misses due to over invalidations. In compiler-directed cache coherence schemes, the conservative analyses performed at compile time can lead to unnecessary cache invalidations and hence false sharing misses. In hardware directory-based schemes, false sharing occurs when different processors access different words of the same cache line, which causes the coherence protocol to treat these accesses as though they are truly shared.

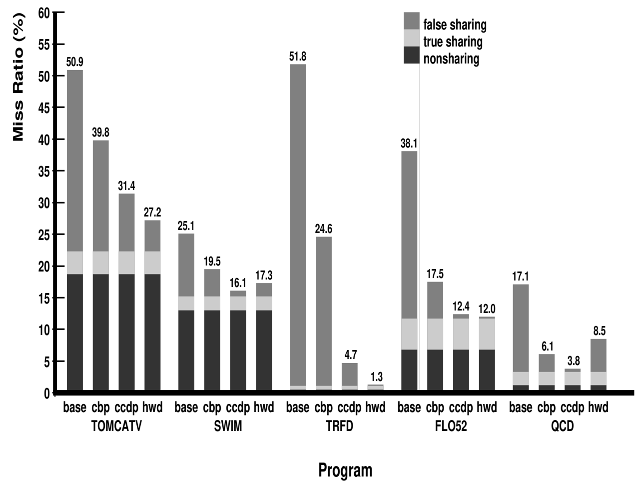

Figure 3. Cache miss ratios under various cache coherence schemes.

Figure 3 shows the cache miss ratios of the applications. By using the CBP scheme, a significant amount of false sharing misses in the BASE scheme are eliminated. The reduction in cache miss ratio ranges from 5.6% in SWIM to 27.2% in TRFD. However, since the compiler analyses performed by the CBP scheme are conservative, there is still substantial number of false sharing misses in TOMCATV and TRFD.

The CCDP scheme further reduces the cache miss ratios of the applications, ranging from 2.3% in QCD to 19.9% in TRFD. This is due to the fact that the CCDP scheme prefetches the potentially stale references, which reduces the number of cache misses. The CCDP scheme also reduces the number of false-sharing misses by eliminating some unnecessary *ce-prefetch* operations.

Compared to the HWD scheme, the CCDP scheme has lower cache miss ratios in the SWIM and QCD applications. This is because of the higher false-sharing effect in the hardware directory protocol. The CCDP scheme reduces the false-sharing effect by invalidating only the required cache words instead of the entire cache line. On the other hand, the CCDP scheme has higher cache miss ratios than the HWD scheme in the other applications. This is due to the conservative compiler analyses which lead to more false sharing misses.

Network traffic The network traffic is made up of read traffic, write traffic and coherence traffic. As their names imply, read traffic is caused by memory read operations across the network, write traffic arises from write operations, while coherence traffic is attributed to cache coherence operations such as invalidations. For each application,

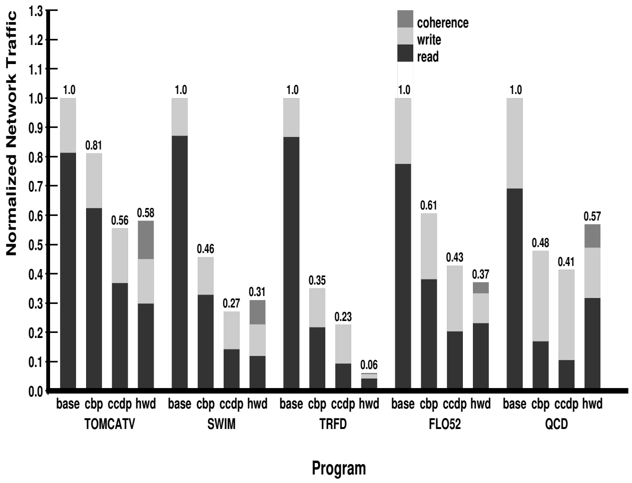

Figure 4. Normalized network traffic under various cache coherence schemes.

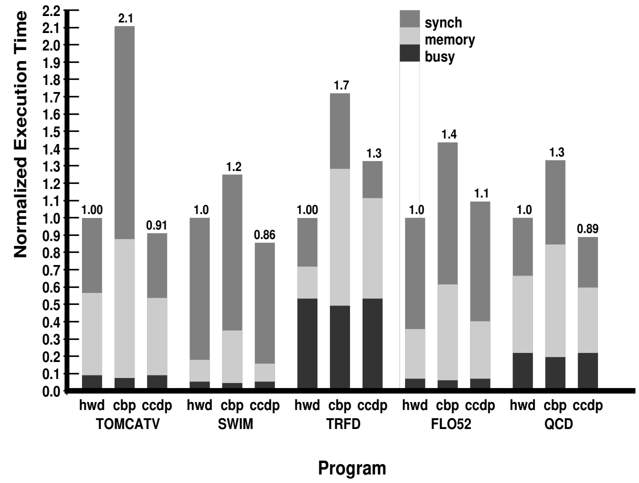

Figure 5. Normalized execution times under various cache coherence schemes.

we normalize the network traffic under the four cache coherence schemes to that of the BASE scheme. Figure 4 shows the normalized network traffic of the applications.

The read traffic is affected by the cache miss ratios. By reducing the cache miss ratios, the CBP, CCDP and HWD schemes also reduce the read traffic. The write traffic depends on the cache write policy used. As the compiler-directed cache coherence schemes use the write-through policy, they incur similar amounts of write traffic. On the other hand, the HWD scheme uses a write-back policy, which incurs a lower amount of write traffic.

As expected, the coherence traffic in the compiler-directed cache coherence schemes studied is negligible. On the other hand, the HWD scheme incurs coherence traffic due to the cache coherence transactions specified by the directory protocol. As a result, the overall network traffic of the HWD scheme is higher than that of the CCDP scheme in some applications such as TOMCATV, SWIM and QCD. The CCDP scheme incurs higher network traffic than the HWD scheme in the other applications due to the higher amount of false-sharing misses.

In absolute terms, the CBP scheme reduces the overall network traffic incurred by the BASE scheme by an amount ranging from 19% in TOMCATV to 65% in TRFD. The CCDP scheme further reduces the network traffic from that of the CBP scheme, by an amount ranging from 7% in QCD to 25% in TOMCATV.

Execution time The simulated execution time is composed of several components. First, the processor busy time is the amount of time spent in executing program instructions. Second, the memory access time is the amount of time spent waiting on memory accesses. Finally, the synchronization time is the delay due to barrier synchronizations. For each application, we normalize the execution times of the schemes to that of the HWD scheme, which is used as the basis of comparison. Figure 5 shows the normalized execution times of the applications.

Our results show that the CCDP and HWD schemes outperform the CBP scheme in all of the applications. However, the CCDP and HWD schemes actually incur a higher processor busy time than the CBP scheme since they perform prefetching operations. The CCDP scheme improves upon the performance of the CBP scheme by reducing the amount of false-sharing misses and network traffic.

The HWD scheme requires cache coherence transactions imposed by the coherence protocol. This increases the network traffic, which in turn increases the latency of memory references, especially in applications with a lot of migratory sharing patterns. The CCDP scheme can outperform the HWD scheme in such applications since coherence-related transactions are carried out locally in each processor. Furthermore, it prefetches the potentially stale references. In our study, the CCDP scheme outperforms the HWD scheme in the TOMCATV, SWIM and QCD applications.

However, the conservative nature of the stale reference detection by the CCDP scheme might lead to excessive amount of potentially stale references, which increases the amount of *ce-prefetch* operations that incur higher expected memory latency. The results showed that the CCDP scheme incurs higher execution times than the HWD scheme in TRFD and FLO52. Overall, the CCDP scheme provides comparable performance as the HWD scheme for the applications studied.

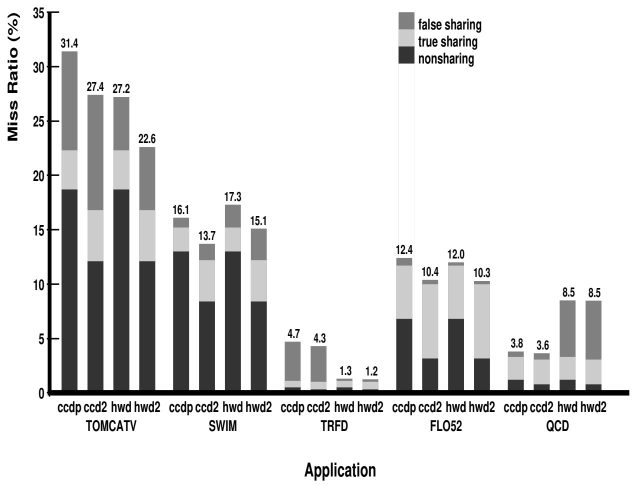

Figure 6. Cache miss ratios for 64-Kbyte and 256-Kbyte direct-mapped data caches.

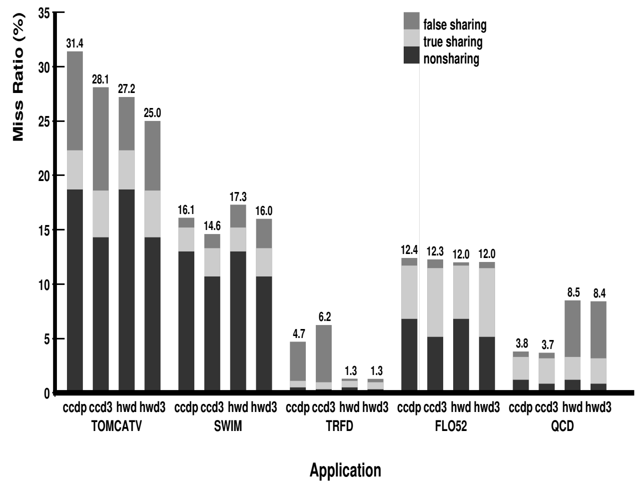

Figure 7. Cache miss ratios for direct-mapped and four-way set associative caches.

5.5.2. Impact of cache organization

In our baseline system model, each processor has a 64-Kbyte direct-mapped cache with a line size of 32 bytes. First, we evaluate the impact of changing the cache size. We simulated a system configuration in which each processor has a larger direct-mapped cache of size 256-Kbyte. Figure 6 shows the impact of cache size on the cache miss ratios. The results for the CCDP and HWD schemes using the 64-Kbyte cache are labeled "ccdp" and "hwd," respectively. The results for the 256-Kbyte cache are labeled "ccd2" and "hwd2," respectively.

The results indicate that the larger cache reduces the nonsharing misses incurred by all of the applications studied by reducing the capacity and conflict misses. In TOMCATV, SWIM and FLO52, there is a substantial reduction in the nonsharing misses for the larger cache. However, when the cache size is increased, the cached data can be retained for a longer period of time, and this results in an increase in the amount of true and false sharing misses. In the applications studied, this effect is most pronounced in TOMCATV. For TRFD and QCD, both nonsharing and sharing misses are not greatly affected by large caches. This means that a cache size of 64 Kbytes is large enough to capture most of the actively shared data for these applications.

Next, we study the impact of the set associativity of the cache. We simulated a system configuration in which each processor has a four-way set associative cache of size 64 Kbytes. Figure 7 shows the cache miss ratios for the direct-mapped versus the four-way set associative organization. In the figure, the results for the CCDP and HWD schemes using the direct-mapped cache are labeled "ccdp" and "hwd," respectively. The results for the four-way set associative cache are labeled as "ccd3" and "hwd3," respectively.

In all of the applications, the number of nonsharing misses for the four-way set associative cache is lower than the corresponding number in the direct-mapped cache. This is because the set associative cache reduces the amount of conflict misses. However, there is an increase in the amount of sharing misses when the four-way set associative cache is used. In fact, the total cache miss ratio of the TRFD application is higher for the four-way set associative configuration due to the increased sharing misses.

5.5.3. Impact of the prefetch hardware

In this section, we study the performance impact of the DCPFU on the CCDP scheme compared to that of existing system-level prefetch hardware. We evaluated the CCDS scheme, which is similar to the CCDP scheme except that its prefetch hardware support is a simplified one modeled after that of the Cray T3D. However, we simulate a 32-entry prefetch queue for each processor instead of the 16-entry prefetch queues used in the actual system. Also, the prefetch hardware does not issue separate *ce-prefetch* and *lh-prefetch* operations. The cache configuration used is a 64-Kbyte direct-mapped cache.

Figure 8 shows the normalized execution times for the CCDP, CCDS and HWD schemes. The results show that the CCDP scheme outperforms the CCDS scheme for all of the applications studied. The overall improvement in the execution times provided by the CCDP scheme over the CCDS scheme ranges from 6.5% in FLO52 to 17.1% in QCD. This shows that the DCPFU is effective in optimizing the performance of the CCDP scheme.

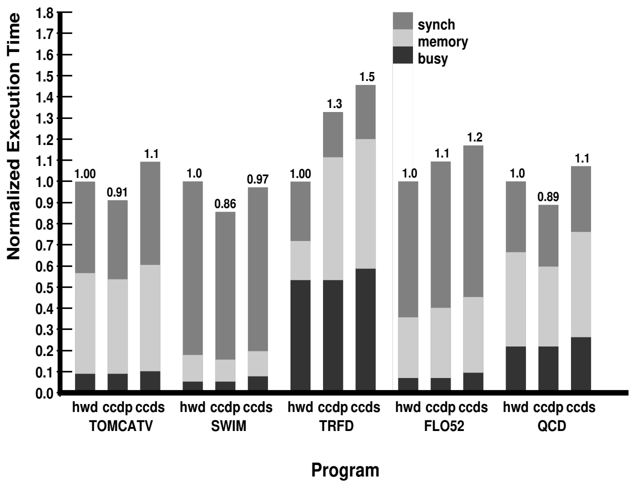

Figure 8. Normalized execution times for different prefetch hardware.

The results show that the CCDS scheme has more processor busy time than the CCDP scheme. First, the CCDS scheme does not distinguish between the *ce-prefetch* and *lh-prefetch* operations. Thus, it has to incur additional software overhead to explicitly perform cache invalidations to maintain correctness. Second, the CCDS scheme uses a prefetch queue to hold the prefetched data. Therefore, it needs to use explicit instructions to move data from the prefetch queue to the data cache.

The CCDP scheme also incurs lower memory access time than the CCDS scheme. Since the DCPFU distinguishes between the *ce-prefetch* and *lh-prefetch* operations and issues them at different priorities, the expected memory latency incurred by the CCDP scheme is lower. Also, the DCPFU provides hardware features to efficiently issue vector prefetch operations and to reduce unnecessary prefetches by merging prefetch requests to the same cache line. This reduces the memory bandwidth requirement of the CCDP scheme, which in turn reduces the expected memory access time.

6. Conclusions

In this paper, we proposed a framework for integrating compiler-directed cache coherence and data prefetching. Under this framework, the CCDP scheme prefetches the potentially stale data references in a parallel program to enforce cache coherence and the nonstale references to hide memory latency. To optimize the CCDP scheme, we designed some inexpensive prefetch hardware to handle the two forms of data prefetching operations it uses. We also discussed the compiler techniques used in the CCDP scheme for stale reference detection, prefetch target analysis and prefetch scheduling. The compiler algorithms have been implemented using the Polaris compiler.

We have performed execution-driven simulations to evaluate the performance of the CCDP scheme. Our experimental results show that the CCDP scheme provides significant performance improvements. For the applications studied, the CCDP scheme provides comparable performance to that of a full-map hardware directory-based cache coherence scheme, but at a lower hardware implementation cost. Thus, we believe that there is good potential to implement the CCDP scheme on future large-scale DSM multiprocessors which do not have hardware cache coherence support.

References

[1] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proc. of the 11th Intl. Symp. on Computer Architecture*, pages 355–362, June 1984.

[2] D. Bernstein, D. Cohen, A. Freund, and D. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proc. of the 1995 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 19–26, June 1995.

[3] Y.-C. Chen. *Cache design and performance in a large-scale shared-memory multiprocessor system*. PhD thesis, University of Illinois at Urbana-Champaign, Center for Supercomputing R & D, 1993.

[4] L. Choi. *Hardware and Compiler Support for Cache Coherence in Large-Scale Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, Center for Supercomputing R & D, Mar. 1996.

[5] L. Choi, H.-B. Lim, and P.-C. Yew. Techniques for compiler-directed cache coherence. *IEEE Parallel & Distributed Technology*, pages 23–34, Winter 1996.

[6] E. Gornish. Compile time analysis for data prefetching. Master's thesis, University of Illinois at Urbana-Champaign, Center for Supercomputing R & D, Dec. 1989.

[7] C. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Trans. Comput.*, 32(12):1091–1098, Sept. 1987.

[8] H.-B. Lim and P.-C. Yew. Maintaining cache coherence through compiler-directed data prefetching. *Journal of Parallel and Distri. Computing*, 53(2):144–173, Sept. 1998.

[9] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Dept. of Electrical Engineering, Mar. 1994.

[10] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Peterson, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. CSRD Tech. Report 1306, Univ. of Illinois at Urbana-Champaign, June 1993.

[11] D. Poulsen and P.-C. Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. In *Proc. of Supercomputing '93*, pages 860–869, Nov. 1993.

[12] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Dept. of Computer Science, Aug. 1992.