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

*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.

Proposed Running Head :

Compiler-directed Cache Coherence and Data Prefetching

Corresponding Author :

Hock-Beng Lim

Dept. of Computer Science, University of Minnesota

4-192 EE/CSci Bldg, 200 Union St. SE,

Minneapolis, MN 55455.

Email : hblim@csrd.uiuc.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 approach 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 numerical 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, comparable to that obtained with a full-map hardware cache coherence scheme.

Key Words : Compiler-directed Cache Coherence, Data Prefetching, Memory Latency Hiding, Shared-memory Multiprocessors

List of Symbols

No special symbols are used in this paper.

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, designers have to solve two important and challenging problems. First, it is imperative that the cache coherence scheme for such systems be efficient, inexpensive and scalable. Second, it is necessary to develop efficient techniques to hide the large remote memory access latencies in such system.

The cache coherence techniques used in existing commercially-available multiprocessors are mainly hardware-based, such as a *snoopy cache protocol* [26] or a hardware *directory-based* scheme [17, 19]. These schemes rely on interprocessor communications to determine cache coherence actions. In large-scale DSM systems, the scalability of such schemes might be affected by excessive coherence-related network traffic. Furthermore, their hardware complexity and cost can become quite substantial as the number of processors increases. *Compiler-directed cache coherence* schemes [7] 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. Thus, from the scalability and performance standpoint, such schemes are especially promising for large-scale DSM multiprocessors.

Several techniques have been proposed in the literature to address the memory latency problem in multiprocessors. In particular, previous research has shown that *data prefetching* is an effective technique to hide memory latency and to improve memory system performance. Data prefetching schemes, which can be hardware-controlled or software-initiated, predict the data required during program execution and overlap the fetching of these data with computation. Hardware and software support for data prefetching are already provided in several experimental and commercially-available multiprocessors.

So far, the cache coherence enforcement and data prefetching problems are often addressed separately. When system designers implement cache coherence schemes and data prefetching schemes independently on a system, the cache coherence actions and data prefetch operations are generated without considering the interaction between them. In fact, compiler-directed cache coherence and data prefetching schemes can be combined in a complementary manner to provide better overall performance. Therefore, we integrate compiler-directed cache coherence with data prefetching under a unified framework [22] to exploit the information on the interaction of cache coherence actions and data prefetch operations. We have developed a compiler-directed cache coherence scheme called the *Cache Coherence with Data Prefetching (CCDP)* scheme as a key part of this framework. The 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 scheme further optimizes performance by prefetching the nonstale data references and hiding their memory latencies.

The main contributions of this paper are as follows. First, we proposed a framework to integrate compiler-directed cache coherence and data prefetching. Under this framework, the CCDP scheme prefetches data to enforce cache coherence and to hide memory latency. The performance of the CCDP scheme is greatly influenced by the hardware and compiler support provided. 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. These techniques can be implemented using existing compiler technology. 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. The results of our experimental study 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. We review the basic concepts of compiler-directed cache coherence and data prefetching, and also describe the key concepts and steps of the CCDP scheme. 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. We discuss the stale reference detection, prefetch target analysis and prefetch scheduling techniques used by the scheme. Section 5 presents the experimental study to evaluate the performance of the CCDP scheme. Finally, we conclude in Section 6 and outline the future directions of our research.

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*. A variety of techniques for the detection, prevention and avoidance of stale references are used by cache coherence schemes.

Most hardware-based cache coherence schemes [17, 19, 26] prevent stale references at run time by invalidating or updating stale cache entries 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 consume an excessive amount of interconnection network bandwidth of the system. In directory-based cache coherence schemes, the cache status information has to be stored in some form of hardware directory. The complexity and hardware costs of a directory-based cache coherence scheme for large-scale multiprocessors could be substantial. At present, large-scale shared-memory multiprocessors with thousands of processors, such as the Cray T3D and Cray T3E, do not provide hardware cache coherence.

In contrast, compiler-directed cache coherence schemes [7] 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. They can also be used to improve memory system performance when hardware cache coherence is not provided. The drawback of these schemes is that there might be situations when the compiler has to adopt conservative decisions. Certain data references might be classified as potentially stale if the compiler cannot determine that they are nonstale. Therefore, the cache coherence actions generated by the compiler-directed schemes might not be as accurate as those of the hardware directory schemes.

Several researchers have developed *Hardware-supported Compiler-directed (HSCD)* schemes [7] which use some hardware to maintain local cache states at run time. With such hardware support, the stale reference detection can be refined. However, the system still incurs full memory latencies to access remote up-to-date copies of shared data. Furthermore, the hardware cost of some HSCD schemes can be significant, which means that they might be impractical for implementation on real systems using off-the-shelf components. In fact, none of the existing HSCD schemes have been implemented on real systems yet.

Although cache coherence schemes can improve multiprocessor cache performance, they cannot completely eliminate remote memory accesses. Thus, memory latency hiding techniques are necessary. Data prefetching is a well established technique to hide memory access latencies. However, cache coherence and data prefetching are often treated as separate problems to be addressed. In hardware cache-coherent systems, data prefetching is usually applied as a separate optimization which is not tightly integrated with the underlying cache coherence scheme. Since cache coherence is transparently handled by the hardware, the data prefetching schemes do not need to worry about the cache coherence issues. 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.

In this work, we developed a framework for integrating compiler-directed cache coherence and data prefetching to improve the memory system performance of large-scale DSM systems which do not have hardware cache coherence support. The framework consists of the *Cache Coherence with Data Prefetching (CCDP)* scheme, and the prefetch hardware and compiler support to optimize the scheme. By using compiler analyses to identify potentially stale references, the CCDP scheme can predict the need to access up-to-date shared data. Therefore, the compiler can direct the processors to prefetch these data into the caches before they are actually needed. Prefetching the potentially stale references implicitly hides the memory latencies of fetching remote up-to-date shared data from the main memory. Further performance improvements can be obtained by prefetching the nonstale references and hiding their memory latencies.

Unlike the HSCD schemes, the CCDP scheme does not make use of special hardware to keep track of cache states and refine stale reference detection. However, it requires system-level prefetch hardware which is less costly than cache coherence hardware. In fact, several existing multiprocessors already provide simple system-level prefetch hardware which can be used by the CCDP scheme.

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, floating-point intensive scientific and numerical-oriented programs written in Fortran which form an important workload for large-scale multiprocessors. These codes are suitable for compiler-directed cache coherence schemes since they can be analyzed with decent accuracy using the existing compiler technology. We expect that, with future advances in compiler technology such as pointer analysis, the CCDP scheme will also be effective for programs that have irregular data accesses and are pointer-intensive. However, such programs are beyond the scope of the present work.

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. The potentially stale data references are the ones to be prefetched for cache coherence enforcement, while the remaining nonstale references are normal read references which can be prefetched for memory latency hiding. 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 two types of prefetch operations: *vector prefetches* and *cache-*

line prefetches. In vector prefetches, a block of data with a fixed stride is fetched from the main memory. On the other hand, 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

Traditionally, data prefetching has been used solely for hiding memory latency. In this context, the data prefetching operations are just performance hints for the system. In fact, the system can ignore these prefetch hints if they will cause exceptions or if there is insufficient hardware resources to issue and process them. However, when data prefetching is used for cache coherence enforcement, it is necessary to address the correctness issue. Such prefetch operations may violate program correctness if they are ignored. Thus, 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 CCDP scheme has to guarantee that 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. No special considerations are needed to ensure program correctness when prefetching the nonstale references.

As the amount of hardware resources provided by the system to handle prefetch requests is often fixed and limited, 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], which is a large-scale DSM system with a shared address space and simple system-level prefetch hardware support. 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. Although our Cray T3D implementation provided substantial performance improvements for the system [20, 21], the performance of the scheme can be further improved by optimizing the prefetch hardware support.

Various researchers have developed sophisticated hardware prefetching schemes [4, 10, 11, 15, 18]. These schemes 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. We have designed a *Data Coherence Prefetch Unit (DCPFU)* for each processor of a large-scale non-cache-coherent DSM multiprocessor.

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 [8]. The organization of a node is shown in Figure 1.

Each node consists of a processor, a data cache and its associated 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, which consists of the node number and the memory address within the node. Thus, the address translation logic helps to determine if a memory reference should be directed at a remote node. The support circuitry also contains communications and synchronization logic to support 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, which corresponds to the maximum number of outstanding prefetch requests to be handled by the DCPFU. 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, nonstale 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. This is because if a data reference matches an entry on both the PHC and the data cache, the version from the PHC will be read. In this manner, the correctness of *ce-prefetch* operations is enforced. 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 similar to the data cache tag and 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 prefetch 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. There are hardware mechanisms to record when a reserved PHC cache line becomes full, which can be used to implement a PHC cache line replacement policy.

There are two possible ways to handle the situation when all the PHC cache lines have been reserved. First, the PCU can stop issuing new prefetch requests until the processor accesses the prefetched data in the PHC. Alternatively, the DCPFU can free up those PHC cache lines which has remained for the longest amount of time in the PHC. This is also done by copying the contents of these lines to the data cache and resetting their reserved bits.

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. This simple design is adequate for the functionality of the PPQ.

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 handled by the DCPFU. 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.

The PCU helps to reduce the amount of unnecessary prefetches dynamically. 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. The CCDP scheme relies on the compiler to identify potentially stale and nonstale data references, and to generate and schedule the appropriate prefetch operations. Several compiler techniques have been developed for software-initiated data prefetching [2, 12, 13, 23, 24]. 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 data-flow analysis*, and *interprocedural analysis*. Extensive algorithms for these techniques were previously developed [6, 7], and implemented using the Polaris parallelizing compiler [25]. We make use of these algorithms in the CCDP scheme. Since the detailed algorithms are described in [6], 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 [6] 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. However, not all of the potentially stale reference patterns actually lead to a stale reference at run time. 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 [6], which treats different regions of arrays referenced in the program as distinct symbolic variables, to refine the stale reference detection. Finally, procedure calls in a program introduce side effects which complicate stale reference detection. We use interprocedural analysis [6] to further improve the accuracy of stale reference detection. In this way, the CCDP scheme can 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. The objective of our prefetch target analysis algorithm is to identify the potentially stale and the nonstale references which should be prefetched. Our prefetch target analysis algorithm [21] 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. On the other hand, the nonstale references which are not in the inner loops are treated as normal read references.

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 [29], 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, and thereby eliminating unnecessary prefetch operations. In addition, a group of references which are likely to refer to the same cache line during the same iteration of the inner loop are said to exhibit *group-spatial reuse* [29]. We only need to prefetch the *leading reference* [29] of such a group of references. 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

4.3.1 Scheduling Techniques

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 [21].

In vector prefetch generation, the array references in each loop are examined to see if they could be pulled out of the loop and still satisfy the control and data dependences. 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. In order to maximize the effectiveness of the vector prefetches, we impose a restriction on the number of loop levels that array references should be pulled out from. 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. 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. Software pipelining hides memory latency by overlapping the prefetches for a future iteration with the computation of the current iteration of a loop. However, we should ensure that the benefits of software pipelining would exceed the instruction execution overhead introduced. In our algorithm, software pipelining will only be used 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 value of this parameter can be empirically determined and tuned to suit a particular system. 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.

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. In this way, some overlap between the prefetch operations and computation can be achieved. 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 goal is to lower the likelihood that a prefetch operation is moved so far back that the prefetched data gets replaced from the cache by the time it is needed, or the distance which a prefetch operation may be moved back is so small that the prefetched data does not arrive in time to be used. This parameter can also be tuned by using an experimental study on the desired system.

4.3.2 Prefetch Scheduling Algorithm

Our prefetch scheduling algorithm [21] considers each inner loop or serial code segment of the program. It determines if the prefetch hardware resources can handle all of the potentially stale and nonstale prefetch target references in the loop. If there is insufficient hardware resources, then it selects the actual prefetch targets to be issued as prefetch operations based on their priority levels. Those potentially stale prefetch targets which will not be issued 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. It is unnecessary to perform loop unrolling in conjunction with vector prefetch generation because the latter already exploits spatial locality of prefetch target references.

4.4 Compiler Implementation

To study the actual performance and design tradeoffs of the compiler algorithms for the CCDP scheme, we have developed a prototype implementation of the algorithms based on the Polaris parallelizing compiler [25]. Polaris is a research parallelizing compiler developed at the University of Illinois. It is a source-to-source compiler which takes a standard Fortran

77 program as input. The output of Polaris is a parallel program in the form of a parallel Fortran dialect with additional directives and assertions. The model of parallelism which Polaris supports is similar to that of the parallel execution model which we assume for the CCDP framework.

In Polaris, the program transformations are performed in several compilation passes. Polaris provides several front-end passes for optimizing and parallelizing a program. The results of the front-end analyses are captured in the intermediate representation of the compiler. Then, back-end passes are applied to generate machine-specific target programs as output. The output code contains directives and assertions to indicate the parallel loops and the data sharing characteristics such as shared or private data.

Figure 3 shows the compilation steps which are used in our CCDP compiler prototype. It takes a sequential Fortran program as input. First, the compiler 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. It also performs advanced passes for symbolic program analysis and range manipulation. These default optimizations and transformations produce information on the parallel loops and the shared and private data in the program which are 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 [6]. The results of this step are used for the subsequent prefetching steps. The prefetch target analysis step produces the sets of potentially stale and nonstale references which should be prefetched. Those references which will not be prefetched are marked as either bypass-cache fetch operations or normal read operations.

The prefetch scheduling step then schedules the prefetch target references. The compiler selects an appropriate prefetch scheduling technique for each loop or serial code segment in the program. 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. These operations are mapped into the relevant prefetch operations on the target system. Finally, the compiler generates the code for our simulator.

5 Performance Evaluation

5.1 Simulation Environment

In our experiments, we use the EPGsim [27] execution-driven simulator to evaluate the performance of several cache coherence schemes in a uniform manner. EPGsim uses source-level instrumentation to generate events for execution-driven simulation. The events simulated include local and global memory references, parallel loop startup and scheduling operations, and synchronization operations. The simulator uses a simple static cyclic scheduling policy to distribute parallel loop iterations to the processors.

Under the EPGsim environment, we can customize the cache simulator to model various compiler-directed cache coherence schemes or hardware 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.

5.2 System Model

We model a 32-processor, non-cache-coherent DSM multiprocessor which is similar to the Cray T3D [8]. The nodes of the system are interconnected via a multistage network. The network delays and remote memory access latencies are simulated using an analytical delay model [16]. Note that the Cray T3D actually uses a 3-D torus mesh network, but simulating the actual network is very time-consuming and beyond the scope of this work. 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. For this study, we choose a simple processor model because our main focus is to study the impact of the data prefetching operations of the CCDP scheme on cache coherence enforcement. As a first step, we study this approach without considering aggressive ILP or processor-level memory optimization techniques since there are few, if any, previous studies on using data prefetching for cache coherence enforcement on systems even with single-issue processors. However, the CCDP scheme can work on systems with advanced ILP processors, as we have previously implemented it on the Cray T3D which uses the DEC Alpha microprocessors. In the future, we plan to study the impact of the CCDP scheme on systems with aggressive ILP processors since it has been shown that data prefetching can have different memory latency hiding capabilities on these processors compared to single-issue processors [28].

Each processor has a lock-up free data cache. In our simulations, we first 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 in this study, 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 [5]. Actually, compiler-directed schemes can also be implemented on systems with write-back caches. When a write-back policy is used, the modified cache words should be written back to the main memory at the epoch boundaries in order to keep the main memory consistent. A simple way to accomplish this on most systems is to flush the caches at the epoch boundaries.

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 for comparison 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 this study, we evaluate four main cache coherence schemes including the CCDP scheme. These schemes require different amounts of hardware and compiler support. We use the same simulation environment and system model but vary the cache coherence strategy so that the performance of these schemes can be compared in a uniform manner. The cache coherence schemes which we simulated are as follows:

- **BASE**: This is the baseline scheme which does not cache shared data in order to avoid the cache coherence problem. All the shared memory references will be directed at the main memory, and incur full remote memory latencies.

- **CBP**: This is the Cache Bypass scheme which avoids stale references by using *bypass-cache fetch* operations.

- **CCDP**: This is the CCDP scheme whose design framework, prefetch hardware support, and compiler techniques have been discussed earlier in this paper.

- **HWD**: This is a hardware directory-based cache coherence scheme which is included in this evaluation to provide a performance comparison against the CCDP scheme. We use software-controlled data prefetching in conjunction with the HWD scheme.

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) [9] 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. The BASE scheme can be easily implemented on real systems. However, its performance is limited by the large latencies for remote shared data accesses.

The 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. From the compiler optimization perspective, the performance of the CBP scheme will give an indication of the impact of applying stale reference analysis alone.

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 [14] 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

The experimental steps are shown in Figure 4. First, we automatically parallelize the application codes using the Polaris compiler [25]. We only invoke the standard parallelization techniques supported by Polaris. Apart from producing the parallelized code, Polaris also provides information on whether the variables used in each loop are shared or private. 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. We use a Polaris implementation of the stale reference analysis algorithm to automatically detect the potentially stale and the nonstale references in these parallelized programs. The resulting programs after stale reference analysis are further transformed so as to add the relevant cache coherence operations to implement the CBP and CCDP schemes. 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 of the scheme.

For the HWD scheme, our compiler inserts prefetch operations using the software popelining technique.

After obtaining the parallel programs with the required cache coherence operations, we instrument these programs [27] 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.

5.5 Application Codes

For our experimental study, we selected two programs from the SPEC CFP95 benchmark suite and three programs from the Perfect benchmark suite [3] 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.

TOMCATV is a highly vectorizable mesh generation program. The main data structures which it uses are seven matrices of size 513×513. SWIM solves a system of shallow water equations using finite difference approximations. The main data structures used in the program are 14 matrices of size 513×513. It is a highly parallel code whose major loops are doubly-nested, with the outer loops being parallel.

TRFD simulates the computational aspects of a two-electron integral transformation according to a fourth-order tensor transformation equation. The evaluation of the integral transformation is formulated as a series of matrix multiplications. FLO52 is a computational fluid dynamics code for analyzing the transonic inviscid flow past an airfoil by solving the unsteady Euler equations. A 2-D domain is discretized into quadrilateral cells. The grid size simulates 192 intervals around the profile and 32 intervals radially. Application of Euler equations to these cells produces a set of coupled ordinary differential equations which is solved by a multigrid method. QCD is a quantum chromodynamics code which performs lattice gauge theory simulation by using a pseudo heat-bath algorithm and a Monte Carlo technique. A major component of the QCD code is the updating of $SU(3)$ matrices, which are a type of 3×3 complex matrix. The code also uses a pseudo-random number generator.

5.6 Performance Results

We use three performance metrics in this study to evaluate the cache coherence schemes, namely, cache miss ratio, network traffic, and simulated execution time of the programs. The cache miss ratio indicates the effectiveness of the cache coherence schemes to cache shared data, and to reduce unnecessary cache invalidations and cache misses. The network traffic measures the network loads generated by the schemes, which affects the scalability of the schemes. Finally, the simulated execution time gives the overall performance of the programs under the schemes studied.

5.6.1 Performance of cache coherence schemes

Let us now present the performance of the four main cache coherence schemes which we evaluated in terms of the three performance metrics.

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. The CCDP scheme attempts to reduce the false-sharing effect by invalidating only the required cache words instead of the entire cache line.

Figure 5 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. In absolute terms, the reduction in cache miss ratio ranges from 5.6% in SWIM to 27.2% in TRFD. Thus, with compile-time stale reference analysis alone, the CBP scheme can improve the caching of shared data. However, since the compiler analyses performed by the CBP scheme are conservative, there is still a 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 can be attributed 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 through prefetch target analysis.

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. 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, we normalize the network traffic under the four cache coherence schemes to that of the BASE scheme. Figure 6 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. This accounts for the substantial reduction in the overall network traffic under these schemes. The write traffic depends on the cache write policy used. As the compiler-directed cache coherence schemes evaluated in this study (i.e., the BASE, CBP, and CCDP 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 than a write-through policy.

As expected, the coherence traffic in the compiler-directed cache coherence schemes studied is negligible. This is because the schemes use compile time analyses to specify cache coherence operations for each processor. There is no need for interprocessor communications. 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 TRFD and FLO52 due to the higher amount of false-sharing misses as a result of conservative compiler analysis.

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. This includes the instructions required to perform data prefetching. 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 measure its execution time under the four cache coherence schemes. Then, we normalize the execution times of the schemes to that of the HWD scheme, which is used as the basis of comparison. We choose not to use the execution time of the BASE scheme as the basis for comparison because the CBP, CCDP and HWD schemes all significantly outperform the BASE scheme. Figure 7 shows the normalized execution times of the applications under the HWD, CBP and CCDP schemes.

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 data 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, which hides the latency of accessing these data. 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 algorithm used 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 experimental results show 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.

5.6.2 Impact of cache organization

The performance of cache coherence schemes is affected by the cache organization. In our baseline system model, we assume that each processor has a 64-Kbyte direct-mapped cache with a line size of 32 bytes. We have studied the performance of four cache coherence schemes with this cache organization. In this section, we examine the effects of varying the cache organization on the performance of the CCDP and HWD schemes.

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. Then, we compare the cache miss ratios of the CCDP and HWD scheme to the corresponding results obtained previously using a 64-Kbyte cache. Figure 8 shows the impact of cache size on the cache miss ratios. In the figure, the results for the CCDP and HWD schemes using the 64-Kbyte cache are labelled "ccdp" and "hwd," respectively. On the other hand, the results for the CCDP and HWD schemes using the 256-Kbyte cache are labelled "ccd2" and "hwd2," respectively.

The results indicate that the larger cache reduces the nonsharing misses incurred by all of the applications studied. This is due to the fact that a larger cache reduces 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, the overall miss ratios do not vary much when the cache size is changed. In these two applications, 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. Then, we compare the cache miss ratios of the CCDP and HWD schemes to the corresponding results obtained previously using a direct-mapped cache. Figure 9 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 labelled "ccdp" and "hwd," respectively. On the other hand, the results for the CCDP and HWD schemes using the four-way set associative cache are labelled as "ccd3" and "hwd3," respectively.

Our results show that 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.

Comparing the performance results shown in Figures 8 and 9, we find that for the applications studied, changing the set associativity of the cache does not lead to as much performance impact on the CCDP and HWD schemes as changing the cache size.

5.6.3 Impact of the prefetch hardware

As we have already shown, the CCDP scheme can be implemented on existing systems such as the Cray T3D which have system-level prefetch hardware support. However, the prefetch hardware available on such systems is not designed to handle the two forms of data prefetching operations of the CCDP scheme. Thus, we designed a DCPFU to optimize the performance of the CCDP scheme. 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 a variation of the CCDP scheme which we call the CCDS scheme. The CCDS scheme is similar to the CCDP scheme except that its prefetch hardware support is a simplified one modeled after that of the Cray T3D, instead of the DCPFU used in the CCDP scheme. However, we simulate a 32-entry prefetch queue for each processor instead of the 16-entry prefetch queues used in the Cray T3D. Also, the prefetch hardware does not issue separate *ce-prefetch* and *lh-prefetch* operations. In all the schemes, the cache configuration used is a 64-Kbyte direct-mapped cache.

Figure 10 shows the normalized execution times for the CCDP, CCDS and HWD schemes. The execution times for the CCDP and CCDS schemes are normalized to that of the HWD scheme. 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. However, the CCDS scheme also provides competitive performance compared to the HWD scheme. In fact, it outperforms the HWD scheme in the SWIM application.

If we examine carefully the components of the execution times shown in Figure 10, we find that the CCDS scheme has more processor busy time than the CCDP scheme in all of the applications. 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. On the other hand, the DCPFU in the CCDP scheme automatically invalidates the potentially stale cache entries while the *ce-prefetch* operations are issued without incurring additional software overhead. 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. On the other hand, since the DCPFU can access the PHC and the data cache in the same clock cycle, there is no extra software overhead involved for data transfer.

The CCDP scheme also incurs lower memory access time than the CCDS scheme for the following reasons. First, 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. Second, 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.

5.6.4 Impact of prefetch scheduling techniques

Another important factor which influences the performance of the CCDP scheme is the effectiveness of the compiler techniques. We have shown the impact of stale reference analysis by comparing the performances of the CBP and BASE schemes. In this section, we examine the performance impact of the prefetch scheduling techniques of the CCDP scheme on the applications studied. The CCDP scheme uses three prefetch scheduling techniques, namely, vector prefetch generation (VPG), software pipelining (SP), and moving back prefetches (MBP).

We apply the following prefetch scheduling algorithms which use the three scheduling techniques incrementally:

- ALG0: In this baseline algorithm, we only apply stale reference analysis and prefetch target analysis. We do not apply any prefetch scheduling techniques. The prefetching operations are simply replaced by bypass-cache fetch operations to get up-to-date data from the main memory.

- ALG1: After applying stale reference analysis and prefetch target analysis, we only apply the VPG technique to schedule the prefetch operations.

- ALG2: This algorithm enhances ALG1 by applying the VPG and then the SP scheduling techniques.

- ALG3: This algorithm applies all three scheduling techniques, in the order of VPG, SP and MBP.

Note that applying ALG0 leads to an improvement over the CBP scheme. In the CBP scheme, all of the potentially stale references are issued as bypass-cache fetch operations. On the other hand, ALG0 reduces some unnecessary bypass-cache fetch operations. By applying the prefetch scheduling techniques incrementally in ALG1, ALG2 and ALG3, we can study the effectiveness of these techniques. Actually, applying ALG3 results in the full CCDP scheme.

For this experiment, we simulate the applications after applying each of the prefetch scheduling algorithms. Then we measure the incremental speedups obtained by each technique, with the results for ALG0 as the basis for comparison. Finally, we compute the percentage of contribution by each scheduling technique towards the overall speedups of the applications. Figure 11 shows the percentage contributions toward the overall speedup given by the prefetching techniques for each application.

From Figure 11, the bulk of the speedups for the applications studied which are achieved by the CCDP scheme can be attributed to the VPG technique. Since the vector prefetch operation has been optimized to exploit the prefetch hardware of the CCDP scheme, VPG is naturally the most effective prefetch scheduling technique. The SP technique also provides significant performance improvements, especially in TOMCATV and TRFD. Finally, the MBP technique provides the least contribution to the speedups of the applications. This is because the amount of memory latency hiding by the MBP technique is less than that of the VPG and SP techniques. However, the MBP technique can provide significant improvement in some applications such as TRFD.

We have measured the incremental performance benefit of each prefetch scheduling technique when applied in the natural order which has been implemented in our compiler. It is possible that if we apply these techniques in a standalone manner or in a different order, the relative performance improvements might be different. However, this is beyond the scope of the present work.

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 with the proposed compiler techniques, 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.

In future, we plan to perform further research into the architecture and compiler issues of the CCDP framework. For the architecture aspect, it is worthwhile to characterize the performance advantages provided by various features of the DCPFU. We would also like to study the impact of the CCDP scheme on large-scale systems which utilize aggressive ILP microprocessors. For the compiler aspect, we plan to study the interaction of the compiler support of the CCDP scheme with the architectural parameters of the system in detail, and to investigate the integration of locality enhancement techniques into the CCDP framework. With the development of more advanced compiler techniques, we plan to validate the effectiveness of the CCDP scheme on programs which are pointer-intensive and have irregular data accesses.

References

[1] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th International Symposium 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 *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, June 1995.

[3] M. Berry, et al. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

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

[5] 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.

[6] 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, March 1996.

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

[8] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.

[9] Cray Research, Inc. *Cray MPP Fortran Reference Manual, Version 6.1*, June 1994.

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

[11] J. W. C. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 54–63, May 1991.

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

[13] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 354–368, June 1990.

[14] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume I, pages 312–321, August 1990.

[15] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] C. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, 32(12):1091–1098, September 1987.

[17] J. Laudon and D. Lenoski. The SGI Origin : A CCNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[18] R. L. Lee, P.-C. Yew, and D. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, August 1987. Also available as CSRD Tech. Report 639, January 1987.

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

[20] H.-B. Lim and P.-C. Yew. A compiler-directed cache coherence scheme using data prefetching. *Proceedings of the 1997 International Parallel Processing Symposium*, pages 643–649, April 1997.

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

[22] H.-B. Lim and P.-C. Yew. Efficient integration of compiler-directed cache coherence and data prefetching. *Best Paper Award, Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, pages 331–340, May 2000.

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

[24] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[25] 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.

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

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

[28] P. Ranganathan, V. Pai, H. Abdel-Shafi, and S. Adve. The interaction of software prefetching with ILP processors in shared-memory systems. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

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

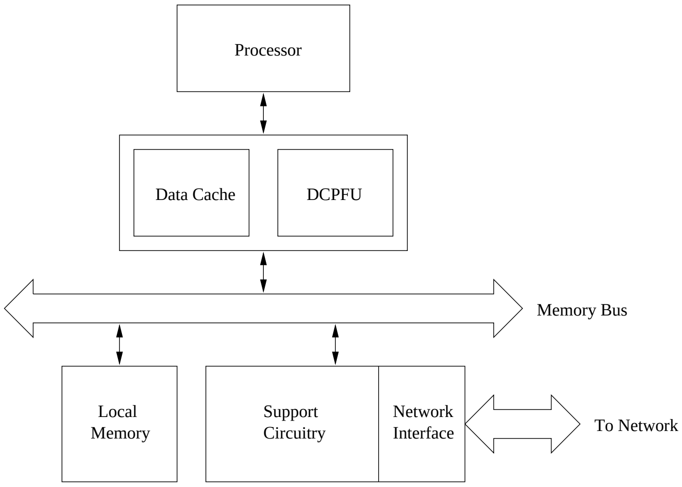

Figure 1: Organization of a Processing Node

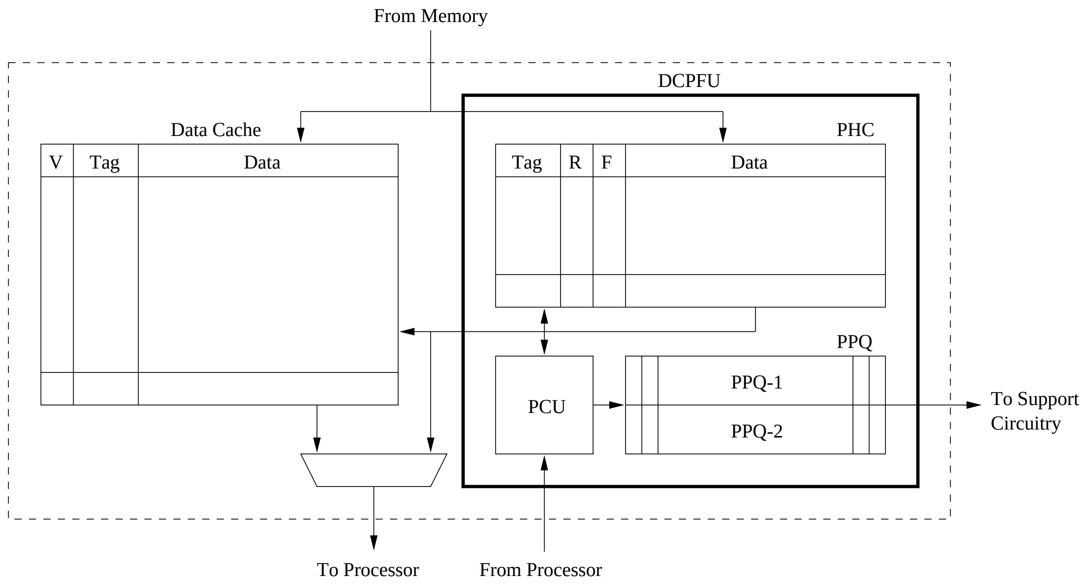

Figure 2: Organization of the DCPFU

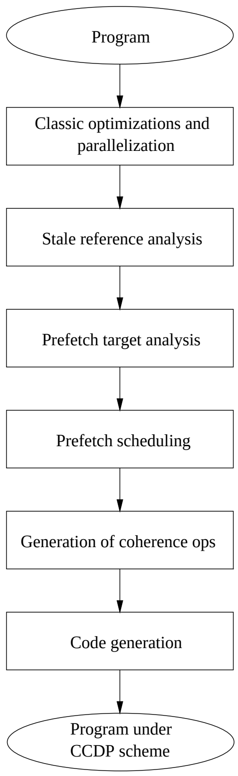

Figure 3: Compilation steps for the CCDP scheme.

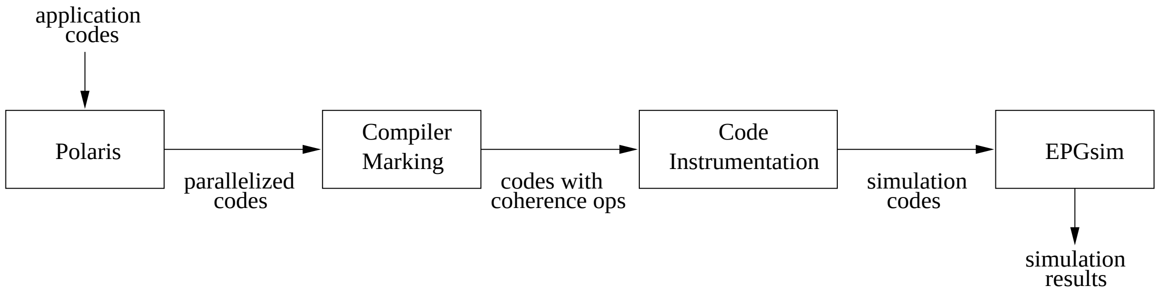

Figure 4: Experimental steps.

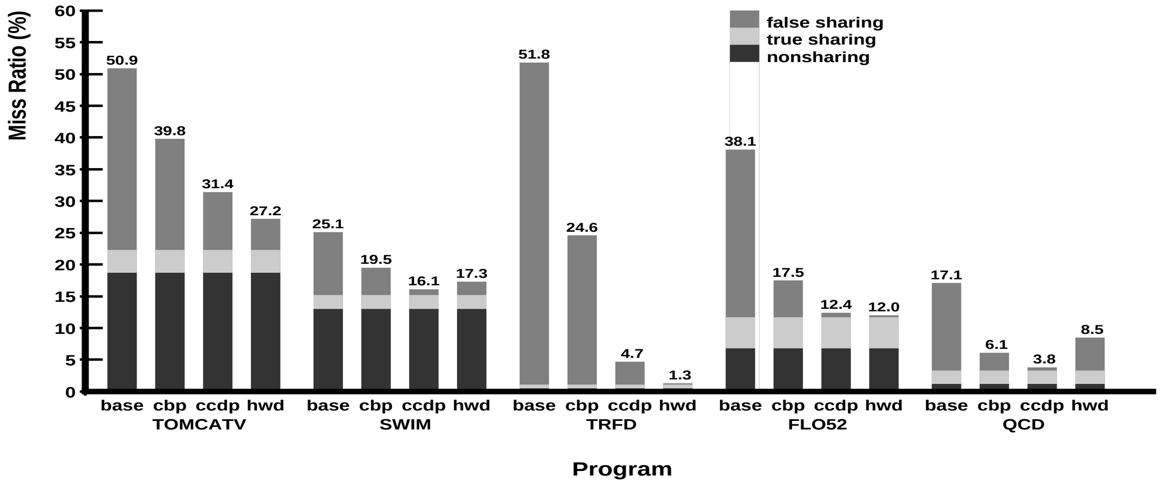

Figure 5: Cache miss ratios under various cache coherence schemes.

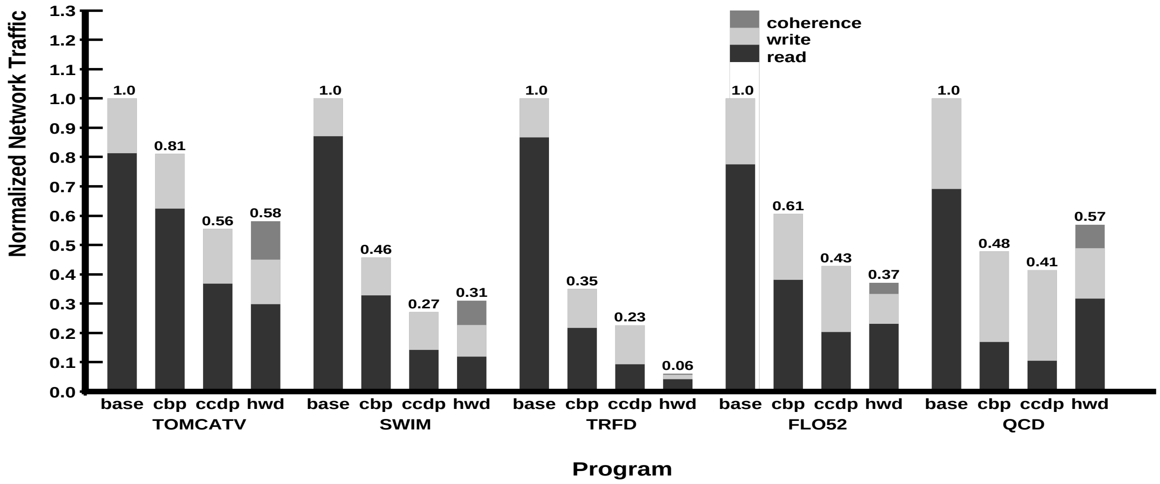

Figure 6: Normalized network traffic under various cache coherence schemes.

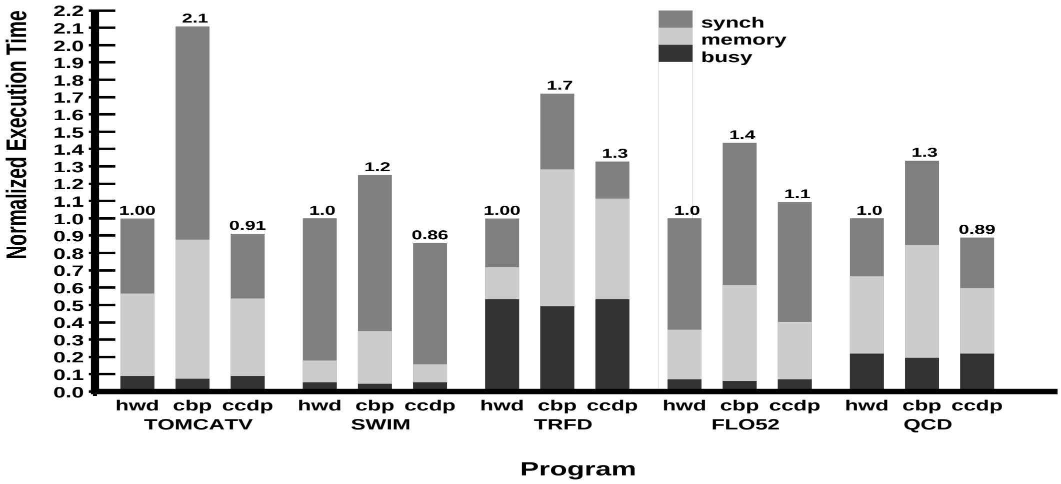

Figure 7: Normalized execution times under various cache coherence schemes.

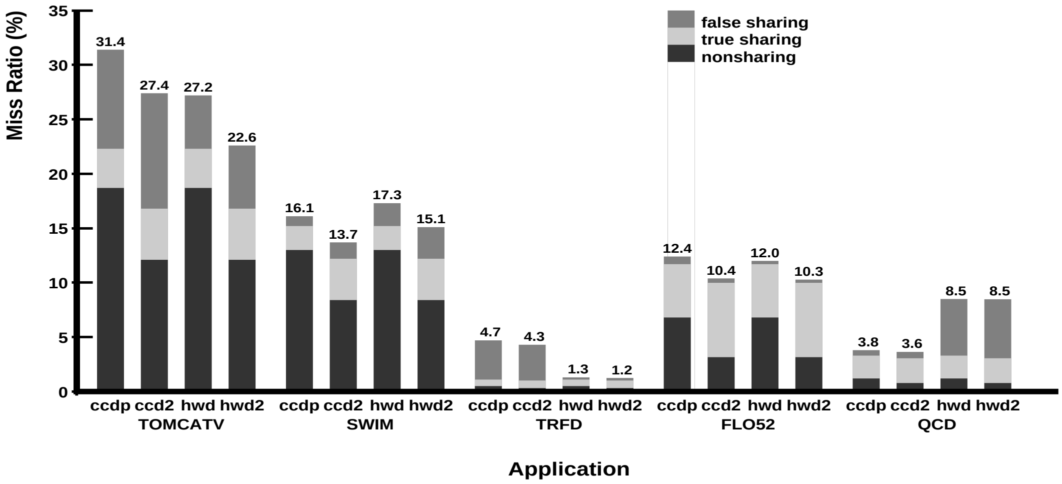

Figure 8: Cache miss ratios for 64-Kbyte and 256-Kbyte direct-mapped data caches.

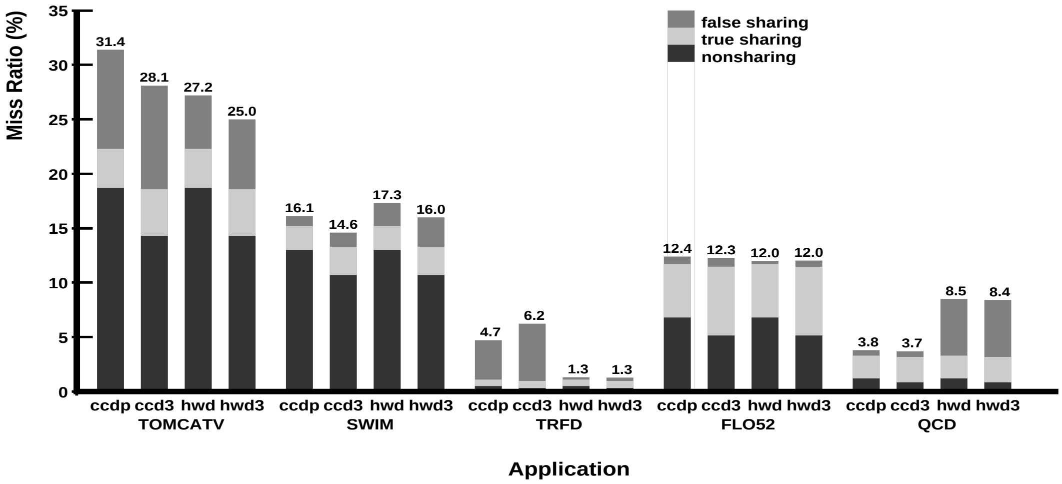

Figure 9: Cache miss ratios for direct-mapped and four-way set associative caches.

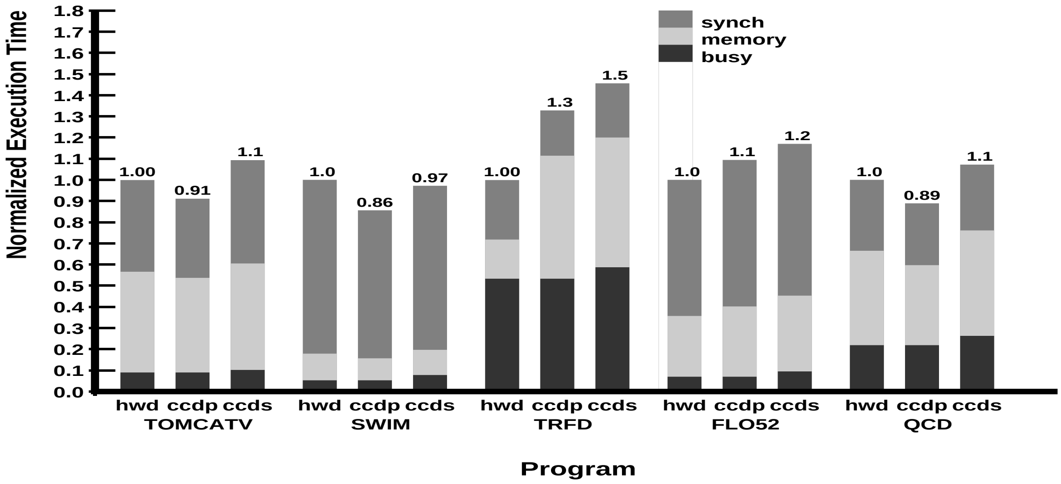

Figure 10: Normalized execution times for different prefetch hardware.

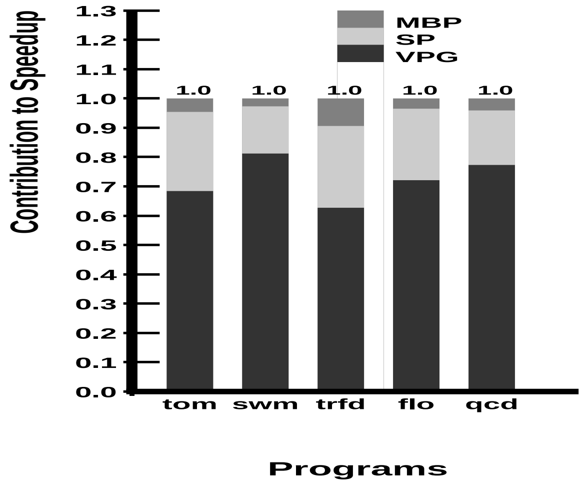

Figure 11: Performance contribution of prefetch scheduling techniques.

Author Biographies

HOCK-BENG LIM received his B.S. in computer engineering, his M.S. in electrical engineering, and his Ph.D. in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1989, 1991, and 1999, respectively. He is currently with the Hewlett-Packard Company, Cupertino, California. Previously, he was a visitor in the Department of Computer Science and Engineering at the University of Minnesota, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois. From 1993 to 1994, he was an associate member of technical staff in the Information Technology Institute in Singapore. From 1991 to 1993, he was an engineer in the Defense Science Organization in Singapore. His research interests include computer architecture, parallel and distributed processing, performance evaluation, and high-performance e-commerce and database systems.

PEN-CHUNG YEW received his Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 1981. He is currently a full professor and the head of the Dept. of Computer Science and Engineering, University of Minnesota. Previously, he was an associate director of the Center for Supercomputing Research and Development at the University of Illinois. From 1991 to 1992, he served as the program director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, D.C.

Pen-Chung Yew is an IEEE Fellow, and has served on the program committee of various conferences. He also served as a co-chairman of the 1990 International Conference on Parallel Processing, a general co-chairman of the 1994 International Symposium on Computer Architecture, and the program chair of the 1996 International Conference on Supercomputing. He served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* from 1992 to 1996, and *Journal of Parallel and Distributed Computing* from 1989 to 1995.

His research interests include high-performance multiprocessor system design, parallelizing compilers, computer architecture, and performance evaluation.