Maintaining Cache Coherence through Compiler-Directed 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
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, and CDA 95-02979. Additional support is provided by a gift from Cray Research, Inc. and by a gift from Intel Corporation. The computing resources are provided in part by a grant from the Pittsburgh Supercomputing Center through the National Science Foundation and by Cray Research, Inc.

Proposed Running Head :

Cache Coherence through Compiler-directed 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.

Phone : (612) 626-9742, Email : hblim@csrd.uiuc.edu

Abstract :

In this paper, we propose a compiler-directed cache coherence scheme which makes use of data prefetching to enforce cache coherence in large-scale distributed shared-memory (DSM) systems. The *Cache Coherence with Data Prefetching (CCDP)* scheme uses compiler analyses to identify potentially-stale and non-stale data references in a parallel program and enforces cache coherence by prefetching the potentially-stale references. In this manner, the CCDP scheme brings up-to-date data into the caches to avoid stale references and also hides the latency of these memory accesses. Furthermore, the scheme also prefetches the non-stale references to hide their memory latencies. To evaluate the performance impact of the CCDP scheme on a real system, we applied the scheme on five applications from the SPEC CFP95 and CFP92 benchmark suites, and executed the resulting codes on the Cray T3D. The experimental results indicate that for all of the applications studied, our scheme provides significant performance improvements by caching shared data and using data prefetching to enforce cache coherence and to hide memory latency.

Key Words and Phrases: Compiler-directed cache coherence; data prefetching; memory system; compiler; shared-memory multiprocessors.

List of Symbols

No special symbols are used in this paper.

1 Introduction

A major performance limitation in large-scale distributed shared-memory (DSM) multiprocessors is the large remote memory access latencies encountered by the processors. To alleviate this problem, caches are used to reduce the number of main memory accesses by exploiting the locality of memory references in programs. However, this introduces the classic *cache coherence problem.* In existing small-scale bus-based SMP multiprocessors available commercially, the standard cache coherence technique used is a *snoopy cache protocol* [28]. Hardware *directory-based* cache coherence schemes, which have better performance and are more scalable than snoopy cache schemes, are used in research machines such as the Stanford DASH [20] and commercial systems such as the SGI Origin. These pure hardware-based cache coherence 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, the complexity and the hardware cost of hardware-based schemes can be quite substantial as the number of processors increases.

Compiler-directed cache coherence schemes [6] offer an alternative approach to solve the cache coherence problem. The key attractive feature of compiler-directed cache coherence schemes 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. Although compiler-directed cache coherence schemes can improve multiprocessor cache performance, they cannot totally eliminate remote memory accesses. Therefore, it is necessary to hide the latencies of remote memory accesses in multiprocessors too. In particular, much research has shown that *data prefetching* is an effective technique to hide memory latency and improve memory system performance. Data prefetching schemes work by predicting the data required later during program execution and overlapping the fetching of such data with current computation.

In fact, data prefetching and compiler-directed cache coherence schemes can be combined in a complementary manner. In this paper, we show how to use data prefetching to implement and optimize a compiler-directed cache coherence scheme for large-scale non-cache-coherent multiprocessors. Our *Cache Coherence with Data Prefetching (CCDP)* scheme makes use of compiler analysis techniques to identify potentially-stale data references in a program and then enforces cache coherence by prefetching up-to-date data corresponding to those potentially-stale references from the main memory. Prefetching the potentially-stale references also implicitly hides the memory latencies of fetching remote up-to-date shared data. In addition, the CCDP scheme further optimizes performance by prefetching the non-stale data and hiding the memory latencies of the non-stale references.

The CCDP scheme is well-suited for large-scale DSM multiprocessors which do not have hardware cache coherence but have hardware support for data prefetching. In the recent years, such systems have emerged as an important platform for high-performance computing. Due to the complexity and cost of implementing a hardware directory scheme that scales to thousands of processors, such MPP systems do not have hardware cache coherence support. Thus, they usually do not cache shared data to avoid the cache coherence problem. To obtain good performance from these systems, it is necessary to use a parallelizing compiler which automatically handles the complex tasks of parallelization, scheduling, and communication optimization. Using the information already gathered by the compiler, the CCDP scheme performs additional compiler analyses and transformations to enforce cache coherence and to hide memory latency. The compiler support required by our scheme is not excessive and can be implemented with existing compiler technology. Furthermore, the scheme does not require complicated cache coherence hardware support. Among the large-scale parallel systems available today, the Cray T3D is a suitable target system to implement this scheme since it has a shared address space and simple system-level prefetch hardware support.

The main contributions of this paper are as follows. First, we developed the CCDP scheme and designed its compiler support. The compiler algorithms used in the CCDP scheme include *stale reference analysis*, *prefetch target analysis*, and *prefetch scheduling*. Second, we show that data prefetching can be applied in a novel framework for both cache coherence enforcement and memory latency hiding. Traditionally, data prefetching has been used solely to hide memory latency. The CCDP scheme goes beyond the conventional application of data prefetching by exploiting two types of data prefetching optimizations for cache coherence enforcement and memory latency hiding. Finally, we implemented the CCDP scheme on the Cray T3D using its existing prefetch hardware and software support. We applied the scheme on five benchmark programs from the SPEC CFP95 and CFP92 suites. Our experimental results show that despite the conservative prefetch hardware and software support on the Cray T3D, the CCDP scheme provided significant performance improvements for all of the applications studied. On a 64-PE Cray T3D partition, the CCDP scheme substantially reduces the overall parallel execution time of the applications studied by 22.3 to 89.8%.

The rest of the paper is organized as follows. Section 2 reviews previous work related to our research. We describe the key concepts and steps of the CCDP scheme in Section 3. In Section 4, we discuss the compiler support required by our scheme. Section 5 presents the experimental study to measure the performance impact of the scheme on the Cray T3D. We describe the target platform, the methodology of the study, the application codes used, and the experimental results obtained. We also discuss the significance of our experimental results. Finally, we conclude in Section 6 and outline the future directions of our research.

2 Comparison with Related Work

2.1 Compiler-Directed Cache Coherence Schemes

Several *hardware-supported compiler-directed* (HSCD) cache coherence schemes [6], which are more scalable and require less hardware support than directory-based cache coherence schemes, have been developed. These schemes use hardware support to maintain local cache states at run time, which helps to improve the efficiency of cache coherence enforcement. 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 are quite 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.

The CCDP scheme addresses these drawbacks in the following ways. First, by prefetching up-to-date data corresponding to the potentially-stale references, the scheme hides the memory latencies of remote up-to-date shared data accesses. Second, the scheme does not require special hardware support to keep track of cache states. It can also make use of the prefetch hardware support available in existing multiprocessors. In this paper, we demonstrate the CCDP scheme on the Cray T3D, thus providing the first implementation of a sophisticated compiler-directed cache coherence scheme on a commercial MPP system.

2.2 Data Prefetching Schemes

Data prefetching is also an active research area. Various researchers have proposed hardware-controlled data prefetching schemes [3, 10, 11, 15, 19] and software-initiated data prefetching schemes [14, 25, 29] which can be implemented in multiprocessors. Recent efforts have focused on the design of prefetching schemes which can handle irregular data reference patterns [22] and also the implementation and performance evaluation of software prefetching algorithms on real systems [2, 30]. Since data prefetching is a well-established technique, hardware support for prefetching has been provided in several experimental and commercial multiprocessors, such as the Illinois Cedar [18], the KSR1 [17], and the Cray T3D [7]. However, most of these efforts focused only on the traditional application of data prefetching for memory latency hiding, particularly in the context of hardware cache-coherent systems. When used in this manner, data prefetching will not violate program correctness. The method which these schemes used to determine the data references to be prefetched is solely based on the estimation of data locality. These conventional prefetching schemes will not work correctly if applied to non-cache-coherent systems.

In contrast, as the CCDP scheme uses data prefetching for cache coherence enforcement in non-cache-coherent systems, it has to address the correctness issue. When determining the data references to be prefetched, the CCDP scheme considers the required cache coherence actions in addition to data locality. To our knowledge, the concept of using data prefetching for compiler-directed cache coherence enforcement as well as memory latency hiding is new. We can leverage the existing and future techniques in compiler-directed cache coherence and data prefetching within the framework of the CCDP scheme.

3 The CCDP Scheme

3.1 Background

Cache coherence schemes ensure that the processors always access the most up-to-date copies of shared data in a parallel program. The remaining invalid copies are known as *stale data*, and the references to these data are called *stale references*. Let us begin with the key concepts of the CCDP scheme.

3.1.1 Parallel Execution Model

A parallel program can be partitioned explicitly (i.e., by the programmer) or implicitly (i.e., by the compiler) into a sequence of *epochs*. Each epoch contains one or more *tasks*. A task is a unit of computation which is scheduled for execution on a processor at run time. A *parallel epoch* contains concurrent tasks, each of which might be comprised of several iterations of a parallel DOALL loop. As there are no data dependencies between the tasks in a parallel epoch, they can be executed in parallel without synchronization. A *serial epoch* contains only one task, which is a sequential code section in the program. Synchronizations are required at each epoch boundary, and the main memory should also be updated in order to maintain consistency. We assume that the system provides architecture and run-time system support to perform this update automatically.

3.1.2 Stale Reference Sequence

In this parallel execution model, stale references can arise as a result of the following sequence of memory operations : (1) a read or write reference to memory location x by processor i, (2) a write reference to x by processor j, where $j \neq i$, and (3) a read reference to x by processor i. Step 1 creates an initial copy of x in processor i's cache. In step 2, a new copy of x is created in processor j's cache, which makes the copy in processor i's cache stale. Finally, the read reference by processor i in step 3 becomes a stale reference. For programs which use DOALL-style parallelism, these three steps will take place in different epochs. This sequence of memory operations is known as a *stale reference sequence*. In modern cache organization, each cache line stores multiple words, which might lead to implicit RAW (read-after-write) and WAW (write-after-write) data dependencies. As a result, a write reference followed by a read reference during a later epoch to data which are mapped on the same cache line is also a possible stale reference condition [5]. These stale reference sequences can be detected by the compiler.

3.1.3 Cache Coherence Enforcement by Data Prefetching

3.1.3.1 Cache coherence schemes

Cache coherence schemes use a combination of techniques for the detection, prevention, and avoidance of stale references [6]. 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 cache states and to perform these cache coherence actions. The complexity and hardware costs of a hardware directory-based cache coherence scheme for large-scale multiprocessors with thousands of processors can be substantial.

In contrast, compiler-directed cache coherence schemes detect the potentially-stale references using compiler analyses at compile time. The compiler can then direct the processors to perform cache coherence actions locally, without the need for interprocessor communications. There is a wide spectrum of design alternatives for compiler-directed cache coherence schemes, which features varying degrees of hardware support. In the simplest compiler-directed cache coherence schemes which are commonly used in existing large-scale multiprocessors, shared data are either not cached at all or they are cached only when it is safe to do so [6]. These schemes require very little hardware support. They also require minimal compiler support as they do not analyze the program to detect stale data references. Although these schemes can be easily implemented, they do not deliver good performance since the amount of shared data that can be cached in this manner is limited.

We can get better performance by using the compiler to detect stale data references, and then enforce cache coherence by ensuring that the potentially-stale references access up-to-date data from main memory. For instance, this can be achieved by using a *bypass-cache fetch* operation, which bypasses the cache and accesses the main memory directly for the up-to-date data. Such an operation can be performed in contemporary high-performance microprocessors such as the DEC Alpha, MIPS R10000, and IBM PowerPC. The use of bypass-cache fetches correctly maintains cache coherence and it does not require additional hardware support. However, its performance depends on the accuracy of the stale reference detection. In practice, the compiler has to be conservative when analyzing stale references. Thus, certain data references will be classified as potentially-stale if the compiler cannot confirm that they are non-stale.

As mentioned earlier, the HSCD schemes use extra hardware to maintain local cache states at run time. With such hardware support, the system can determine whether potentially-stale references across epoch boundaries are actually stale or non-stale. This improves the exploitation of intertask temporal locality, which in turn reduces the amount of unnecessary cache invalidations or remote memory accesses.

3.1.3.2 Interaction of cache coherence with data prefetching

Although cache coherence schemes can improve multiprocessor cache performance, they cannot completely eliminate remote memory accesses. Thus, memory latency hiding techniques such as data prefetching are necessary. In fact, since cache coherence schemes generate cache coherence actions and memory accesses, data prefetching can be used to hide the latency of these operations and thereby optimizing the performance of cache coherence schemes.

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 non-stale data references are treated uniformly by the prefetching schemes. As for the existing compiler-directed cache coherence schemes, they do not efficiently incorporate data prefetching. They incur full memory latencies to access up-to-date copies of shared data from remote memory modules whenever stale or potentially-stale references are encountered.

Our CCDP scheme exploits the interaction of cache coherence and data prefetching. By using compiler analyses to identify potentially-stale references, we are predicting the need to access up-to-date shared data in advance. Therefore, the compiler can direct the processors to prefetch these data into the caches before they are actually needed. At the same time, data prefetching implicitly hides the memory latencies of fetching remote up-to-date shared data from the main memory. The basic version of the CCDP scheme [21] enforces cache coherence by prefetching only the potentially-stale references of a program. In addition, further performance improvements can be obtained by prefetching the non-stale data and hiding the memory latencies of the non-stale references.

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 and is already available on existing systems. The CCDP scheme also requires compiler support for prefetching in addition to stale reference detection. Note that the prefetching operations for potentially-stale references are *compulsory* since they are used to maintain cache coherence. On the other hand, the prefetching operations for non-stale references are *optional*, and they can be used to get better performance with more compiler analyses.

3.2 Overview of the CCDP Scheme

The CCDP scheme consists of three major steps :

1. **Stale reference analysis**. The compiler identifies the potentially-stale and non-stale 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 non-stale references are normal read references which can be prefetched for memory latency hiding.

2. **Prefetch target analysis**. It might not be necessary or worthwhile to prefetch all of the potentially-stale and non-stale references. Thus, the prefetch target analysis step determines which of the references in these two categories should be prefetched. This minimizes the number of unnecessary prefetches.

3. **Prefetch scheduling**. Having identified the suitable target references for prefetching, we need to schedule the prefetch operations. The prefetch scheduling algorithm inserts the prefetch operations at appropriate locations in the program.

In addition to these steps, the compiler also inserts cache and memory management operations such as cache invalidations and bypass-cache fetches into the program 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.

Note that the compiler analysis techniques which we apply to generate the prefetch operations are similar in nature to those used for message vectorization or pipelining in message-passing systems. However, the motivations behind these two approaches are actually different. A key issue which differentiates message-passing optimizations and shared-memory optimizations is the treatment of cache coherence. In message-passing systems, the cache coherence problem does not arise. Under this programming model, the programmer has to explicitly specify the necessary data partitioning and access patterns to ensure that each processor always access the correct data. In other words, the programmer is responsible for determining the potentially-stale references in this context. The compiler applies message aggregation or pipelining to optimize the underlying *send-receive* operations of the parallel program which have been specified by the programmer.

In contrast, the CCDP scheme focuses on compiler-directed cache coherence under shared-memory programming model. Here, the compiler uses stale reference analysis to automatically identify the potentially-stale references. The programmer does not need to explicitly specify the stale references in the program. Then, the compiler applies the prefetch target analysis and prefetch scheduling techniques to generate prefetch operations to move the correct data to the processors.

An advantage of our approach over the message-passing approach is that it allows the programmer to use the simpler shared-memory programming model, and relieves the programmer from some of the tasks required in message-passing programming such as specifying data distribution and access patterns. In data-parallel languages such as HPF, the data distribution patterns are restricted to only those supported by the data distribution primitives, and they are often regular in nature. Irregular data distributions are usually not so well supported. In our work, we focus on the shared-memory model where there is no limitation on data distribution. This gives the compiler more flexibility to analyze and optimize a parallel program.

3.3 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 actually makes use of two types of data prefetching optimizations which we call *coherence-enforcing prefetch (ce-prefetch)* and *latency-hiding prefetch (lh-prefetch).* For the CCDP scheme to be efficient, the *ce-prefetch* and *lh-prefetch* operations should be tailored specifically to suit their requirements.

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. There are two aspects to this problem. First, the *ce-prefetch* operations should respect all control and data dependence constraints. Thus, our prefetch scheduling algorithm does not schedule the *ce-prefetch* for a variable before the point where it was potentially last modified by another processor. Also, to be conservative, we do not schedule *ce-prefetch* operations speculatively. 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 specialized hardware support, the compiler has to insert explicit cache invalidation instructions to invalidate the cache entries before the prefetches are issued.

Compared to the *ce-prefetch* operations, the *lh-prefetch* operations are straight-forward. Since they are used for prefetching the non-stale references, they will not violate cache coherence. No special considerations are needed to ensure program correctness when prefetching the non-stale references.

The performance issue of the prefetch operations is also important. Since 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. Obviously, it is more important to prefetch the potentially-stale references and maintain cache coherence as efficiently as possible. Thus, in the CCDP scheme, we assign 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.

3.4 Example

We present a simple example to illustrate how the CCDP scheme works. Consider the matrix multiplication code segment shown in Figure 1.

The outer loop of the code is parallel, and we assume that each loop iteration is performed by a different processor. There are three potentially-stale references in the inner loop of the code segment (marked as *PS-ref*) to the elements in arrays A, B, and C because the initial values of these arrays might have been assigned on a different processor before entering the matrix multiplication code segment.

The CCDP scheme would prefetch the data for these potentially-stale references. Figure 2 shows how the *ce-prefetch* operations might be generated by the CCDP scheme. There are two *ce-prefetch* operations : *ce-pref* and *ce-vecpref*. The *ce-pref* operation is a cache-line prefetch operation, while *ce-vecpref* is a vector prefetch operation. As mentioned earlier, if the system does not have specialized prefetch hardware support to handle *ce-prefetch* operations, then the compiler will insert cache invalidation operations before the prefetch operations to ensure correctness.

4 Compiler Support

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 [5, 6] and implemented using the Polaris parallelizing compiler [27]. We make use of these algorithms in the CCDP scheme. Since the detailed algorithms are described in [5], we will only discuss the main ideas and functions of the stale reference analysis techniques used.

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 [5] 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 stale reference detection can be refined by exploiting some temporal and spatial reuses of memory locations in the program. To do so, two *locality preserving analysis* techniques [5] are used. These techniques help to reduce the number of unnecessary coherence operations.

We use array data-flow analysis [5] to perform stale reference detection more accurately compared to early stale reference detection algorithms [4]. In previous algorithms, an array is treated as a single variable to simplify the compiler analysis. However, this is likely to overestimate the amount of potentially-stale references. The array data-flow analysis algorithm solves this problem by treating different regions of arrays referenced in the program as distinct symbolic variables.

Finally, procedure calls in a program introduce side effects which complicate stale reference detection. Previous stale reference detection algorithms [4] avoided this problem by invalidating the caches at procedure boundaries or by inlining the procedures. However, cache invalidations at procedure boundaries can increase the number of unnecessary cache misses at run time. Inlining might lead to excessive growth in code size, which in turn increases the compilation time and memory requirement for the program. We use interprocedural analysis [5] to further improve the accuracy of stale reference detection, so that 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 role of the prefetch target analysis algorithm is to identify the potentially-stale and the non-stale references which should be prefetched. Our prefetch target analysis algorithm makes use of simple heuristics which are easy to implement and are likely to be effective. The algorithm initially includes all of the potentially-stale and non-stale references in two separate sets of possible prefetch candidates. As the algorithm proceeds, those references which should not be prefetched are removed from the sets. The algorithm is shown in Figure 3.

Our prefetch target analysis algorithm focuses on the potentially-stale and the non-stale references in 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 so as to preserve program correctness. On the other hand, the non-stale 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 [31], 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. The prefetch target analysis algorithm marks the prefetch target references which exhibit self-spatial locality using a similar approach as in [2, 23], so that the prefetch scheduling step can carry out the appropriate loop unrolling.

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* [31]. We only need to prefetch the *leading reference* [31] of such a group of references. The other references in the group are issued as normal reads. If a set of references are *uniformly generated* (i.e., they have similar array index functions which differ only in the constant term) [12], then they have group-spatial reuse. To detect this reuse, the compiler needs to construct expressions for the address of each reference in terms of the loop induction variables and constants. Next, given the cache line size and the size of the data object referenced (e.g., a byte, a word, or a double word), the compiler can perform mapping calculations to determine whether these addresses are mapped onto the same cache line. However, the arrays should be stored starting at the beginning of a cache line for this analysis to be correct. This can be enforced by specifying a compiler option. If the addresses cannot be converted into linear expressions, our algorithm will conservatively treat them as references to be prefetched.

It might be possible to further reduce the number of unnecessary prefetch operations by also exploiting *self-temporal* and *group-temporal* [31] localities. However, this would require additional compiler analyses and transformations, such as the estimation of *loop volume* [2, 23]. Some arbitrary decisions and approximations must be made to handle complications such as unknown loop bounds. Therefore, we do not consider these forms of localities in our algorithm.

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 non-stale references which are eliminated from the set of prefetch targets are simply issued as normal read operations.

4.3 Prefetch Scheduling

After the prefetch target analysis step, we have the set of potentially-stale and the set of non-stale target references which should be prefetched. The next important task is to schedule these prefetches.

4.3.1 Design Considerations

There are four main design considerations for the prefetch scheduling algorithm. First, it should ensure that the correctness of memory references is not violated as a result of the prefetch operations. Second, like conventional data prefetching algorithms, our prefetch scheduling algorithm improves the effectiveness of the prefetch operations so that the prefetched data will often arrive in time for the actual data references. If the prefetched data arrive too early, they might be replaced before they are used. Although this will not violate cache coherence, it leads to cache misses which affects performance. On the other hand, if prefetched data arrive too late, the processors would have to wait for the data to arrive.

Third, the prefetch scheduling algorithm should take into account several important hardware constraints and architectural parameters of the system :

- The size of the data cache in each processor, which puts an upper bound on the amount of data which should be prefetched into the cache.
- The size of the prefetch queue or issue buffer for each processor, which limits the number of prefetch operations which can be issued at any particular instant of time.

- The maximum number of outstanding memory operations allowed by the processor, as current high-performance microprocessors can only keep track of a fixed number of outstanding memory operations.

- The average memory latency for a prefetch operation, which limits how much computation can be overlapped with prefetch operations.

Finally, the prefetch scheduling algorithm should try to minimize the prefetch overhead. Since vector prefetches incur lower overhead compared to several cache-line prefetches, our prefetch scheduling algorithm generates vector prefetches as much as possible.

4.3.2 Scheduling Techniques

Our prefetch scheduling algorithm makes use of three scheduling techniques : *vector prefetch generation*, *software pipelining*, and *moving back prefetches*. These techniques are similar to the scheduling strategies used in conventional prefetching algorithms for memory latency hiding. But, we have enhanced and adapted them to suit the requirements of the CCDP scheme. Our algorithm effectively combines these techniques and applies them at appropriate locations within a program.

Vector prefetch generation Gornish [13, 14] developed an algorithm for conservatively determining the earliest point in a program at which a block of data can be prefetched. It examines the array references in each loop 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.

However, a drawback of Gornish's algorithm is that it tries to pull out array references from as many levels of loop nests as possible, and does not consider important hardware constraints such as the size of the prefetch queue or issue buffer and the cache size. Even if array references can be pulled out of multiple loop levels, the prefetched data might not remain in the cache by the time the data are referenced.

We adapt Gornish's approach to generate vector prefetches. The basic algorithm for pulling out array references is described in [13]. 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 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. The compiler then inserts the prefetch operation into the code just before the appropriate loop.

Software pipelining In Mowry's approach [23, 25], *software pipelining* is used to schedule cache-line prefetch operations. A loop is transformed into 3 sections : (1) a *prolog* loop which prefetches the data for the first few (say n) iterations, (2) a *steady state* loop which executes the current iteration and prefetches data for a later iteration, and (3) an *epilog* loop which executes the code for the last n iterations. The number of iterations to prefetch ahead, n, is given by

$$\left\lceil \frac{average\ prefetch\ latency}{loop\ execution\ time} \right\rceil$$

.

Software pipelining is an effective scheduling strategy to hide 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. We adapt the software pipelining algorithm to suit the CCDP scheme. In our algorithm, software pipelining will be used only for inner loops that do not contain recursive procedure calls. After the compiler computes the expected loop execution time and the number of iterations to prefetch ahead, it decides whether it is profitable to use software pipelining. This is a design issue which is machine dependent. 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 or the cache size.

Moving back prefetches 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 as far away as possible from the point where the data will be used. There are several situations in which this technique is applicable. First, it might not be possible to pull out array references from some loops due to unknown loop scheduling information or control and data dependence constraints. Second, the prefetch hardware resources might not be sufficient if we prefetch the references in certain loops using vector prefetch generation or software pipelining. Third, we can use this technique for prefetch targets in serial code sections, where vector prefetch generation and software pipelining are not applicable.

We adapt Gornish's algorithm for pulling back references [13]. The original algorithm tries to move references as far back as control and data dependence constraints allow. However, the algorithm might move a prefetch operation so far back that the prefetched data gets replaced from the cache by the time it is needed. Another possible situation is that the distance which a prefetch operation may be moved back is not large enough for the prefetched data to arrive in time to be used. Thus, 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. This parameter can also be tuned via experiments on the desired system.

4.3.3 Prefetch Scheduling Algorithm

Our prefetch scheduling algorithm is shown in Figure 4. It considers each inner loop or serial code segment of the program. First, it determines if the prefetch hardware resources can handle all of the potentially-stale and non-stale 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 non-stale prefetch targets which are not issued are replaced by normal read operations.

Next, depending on the type of loop or code segment, the algorithm uses a suitable scheduling technique for the prefetch targets within the loop or code segment. There are six main cases which our algorithm handles. In case 1, the algorithm first tries to schedule the prefetches in a serial loop with known loop bound using vector prefetch generation. If vector prefetch generation fails, then the algorithm tries to schedule the prefetches using software pipelining, which has a higher overhead than vector prefetch generation, but allows more overlap between prefetches and computation than moving back prefetches. If software pipelining is also not suitable, the algorithm will move back the prefetches to get at least some overlap between prefetches and computation. On the other hand, when the loop bound of the serial loop is unknown, the algorithm cannot determine the length of the prefetch vector. Thus, it does not use vector prefetch generation in this situation.

In case 2, when the prefetches are in a parallel loop with static scheduling and known loop bounds, it is possible to issue vector prefetches. It is more complicated to use software pipelining since the algorithm would need to take into account the processor number and the loop scheduling policy. So, our algorithm does not make use of software pipelining in this case. If the loop bounds are unknown, then our algorithm will only consider moving back the prefetches. In case 3, the parallel loop uses dynamic scheduling. Due to unknown scheduling information, it is impossible to predict the iterations which are executed by each processor. Thus, the algorithm cannot use vector prefetch generation or software pipelining, but will try to move back the prefetches regardless of whether the loop bounds are known or not. Similar action will be taken in case 4, since vector prefetch generation and software pipelining are not applicable to serial code sections.

In case 5, when the prefetches are in a loop which contains if-statements, our algorithm uses the conservative strategy of moving back prefetches within the boundary of the if-part or else-part of the if-statements. This is to minimize unnecessary prefetches when the if-statements are executed. Similarly, if the loop or serial code segment is within the body of an if-statement (in case 6), the algorithm applies the actions for cases 1, 2, 3, or 4, but the prefetches are scheduled within the if-statement.

Note that loop unrolling is performed before software pipelining is applied to a loop if the loop contains prefetch targets which exhibit self-spatial locality. As discussed earlier, loop unrolling is applied to exploit self-spatial locality. The degree of unrolling is controlled to prevent the size of the resulting loop from growing too large. It is unnecessary to perform loop unrolling in conjunction with vector prefetch generation because it already exploits the spatial locality of prefetch target references. Finally, loop unrolling is unlikely to provide many benefits for a loop if the scheduling technique used is moving back prefetches.

5 Experimental Evaluation

We have conducted application case studies to obtain a quantitative measure of the performance improvements which can be obtained by using the CCDP scheme on the Cray T3D [7]. We adapted the CCDP scheme to suit the hardware constraints and architectural parameters of the system. The results of our application case studies on a real system give a more accurate indication of the effectiveness of the scheme, compared to the results obtained through simulations. In addition, studying the implementation of the CCDP scheme on a real system enables us to discover the hardware and software limitations in an existing system, which will help us to design new hardware and software mechanisms for implementing the scheme efficiently on future large-scale multiprocessors.

The drawback of using a real system to evaluate the CCDP scheme is that the performance metrics used are limited to those which can be easily measured on the system, such as the execution time and speedup of parallel programs. Also, the hardware and software support on a real system cannot be easily modified to study the effects of system-related parameters on the performance of the scheme. Although simulation studies do not provide as good an accuracy, they allow more flexibility for the performance metrics and also the hardware and software parameters of the simulated system.

5.1 Target Platform

The Cray T3D is a physically distributed memory MPP system which provides hardware support for a shared address space [26]. A special circuitry in each Processing Element (PE), called the *DTB Annex*, helps to translate a global logical address into its actual physical address, which is composed of the PE number and the local memory address within the PE. The Cray T3D provides simple hardware support for data prefetching. Whenever a processor issues a prefetch operation, it has to set up a DTB Annex entry corresponding to the remote address to be prefetched. Each prefetch instruction transfers one 64-bit word of data from the memory of a remote PE to the local PE's *prefetch queue*. The processor then explicitly extracts the prefetched word from the queue when it is needed. The prefetch queue can only store 16 words of data. When the prefetch queue is full, no more data can be prefetched until the prefetched words are extracted from the queue. Previous studies [1, 16] indicated that the overhead of interacting with the DTB Annex and the prefetch queue is significant.

Software support for shared address space and data prefetching is provided. The programmer can use a compiler directive in the Cray MPP Fortran (CRAFT) language [8] to declare shared data and to specify their distribution pattern amongst the PEs. A directive called *doshared* is used to specify a parallel loop. In order to avoid the cache coherence problem, the shared data are not cached under CRAFT. However, the programmer can explicitly access the local address spaces of the PEs and treat the collection of local memories as a globally shared memory. Data prefetching and data transfer can be performed in several ways. First, the programmer can use an assembly library which provides functions to set the DTB Annex entries, issue prefetch requests, and extract prefetched data from the prefetch queue [26]. Second, the programmer can explicitly perform remote memory reads and writes in a cached or noncached manner. Finally, the Cray T3D also provides a high-level shared-memory communication library called the *SHMEM library* [9]. The *shmem_get* primitive in the library provides similar functionality as a vector prefetch.

The *SHMEM library* has been optimized to utilize the DTB Annex and prefetch queue as efficiently as possible. We performed a simple test to measure the time taken to prefetch data using the *shmem_get* primitive. We used a Cray T3D partition with 64 PEs. In this test, a PE uses the *shmem_get* primitive to transfer a message of length n words from another PE. The time taken for the data transfer against the message length (from 1 to 256 words in powers of 2) is plotted in Figure 5. From the figure, there is a linear relationship between the time taken (in microseconds) and the message length (in words) of the form

$$t = t_0 + \beta n,$$

where t_0 is the startup time and β is a communication parameter which characterizes the performance of *shmem_get* on the Cray T3D. The measurements indicate that $t_0 \approx 1.12\ \mu s$ and $\beta \approx 0.26\ \mu s/words$ for the message lengths considered. Our test re-affirmed the fact that it is more efficient to use the *shmem_get* primitive to perform vector prefetches of multiple words instead of single-word prefetches.

5.2 Methodology

First, we automatically parallelize the application codes using the Polaris compiler [27]. We only invoke the standard parallelization techniques supported by Polaris. Next, by using the parallelism and data sharing information from Polaris, we manually convert the parallelized codes into three versions of CRAFT programs :

1. **BASE** : This is the baseline version which uses the default software support for shared address space in CRAFT. The BASE codes do not cache shared data. Moreover, they incur full memory access latencies for all references to remote shared data.

2. **CCDP-0** : This is an optimized version which caches shared data and maintains cache coherence by prefetching only the potentially-stale references according to the CCDP scheme.

3. **CCDP-1** : In addition to the potentially-stale references, the CCDP-1 codes also prefetch the non-stale references for additional memory latency hiding.

To create these codes, we use the following procedure. First, we distribute the shared data in the programs to the PEs. In the BASE codes, we use the shared data distribution directives available in CRAFT directly. For the CCDP-0 and CCDP-1 codes, we explicitly manage the local memories of the PEs as a shared global memory. A shared array is divided into equal portions and each portion is owned by a PE. Each PE can cache its portion of the array because it is not declared as a CRAFT shared variable.

Next, the DOALL loop iterations are distributed among the PEs. For the BASE codes, a DOALL loop is directly converted into a CRAFT *doshared* loop. In a *doshared* loop, each PE will execute the iterations which use the portion of shared data that reside in the PE's local memory. On the other hand, both the CCDP-0 and CCDP-1 codes do not use the *doshared* directive. Instead, they directly assign loop iterations to the PEs according to the same loop scheduling policy as the corresponding BASE code.

The data distribution and parallelization strategies which we employ are simple and straightforward, so that they can be easily applied on the applications by hand. With more sophisticated compiler techniques to optimize data distribution and work partitioning, such as those used by HPF compilers, the performance of the applications can be further improved. However, the use of such optimizations is outside the scope of this work. Nevertheless, we would like to emphasize that the CCDP scheme can work in conjunction with such optimizations. Using these data distribution and work partitioning optimizations alone will not solve the cache coherence constraint on the system. Therefore, the CCDP scheme can complement such optimizations nicely.

Finally, we need to insert prefetch operations into the CCDP-0 and CCDP-1 codes. By using a Polaris implementation of the stale reference analysis algorithms [5], we automatically identify the potentially-stale references and the non-stale references in these programs. Then, we manually apply the prefetch target analysis and prefetch scheduling algorithms of the CCDP scheme on the codes. Due to the complexity and the time required to implement the compiler algorithms to automatically add prefetch operations into programs, previous software-initiated data prefetching schemes [14, 24] are often initially evaluated by manually inserting prefetch instructions in the codes. Most of the previous compiler-directed cache coherence schemes are also initially evaluated by manually adding cache coherence actions. Similarly, by using hand compilation in this work, we can quickly evaluate the potential performance of the CCDP scheme empirically without sacrificing accuracy. We are in the process of automating these steps in the Polaris compiler.

Two types of prefetch operations are used in our experimental study; namely, the prefetch library function which transfers one cache line of data per request and the *shmem_get* primitive for vector prefetches. Again, we emphasize that the CCDP-1 codes prefetch the non-stale references as well as the potentially-stale references. It is important to note that the Cray

T3D prefetch hardware do not distinguish between prefetch requests for potentially-stale and non-stale references. Thus, we have to use cache invalidation operations in both the CCDP-0 and CCDP-1 codes when necessary to maintain correctness.

After the entire process described above is completed, we have produced all three versions of codes. We then execute them on the Cray T3D and measure their execution times. From the execution times, we compute the speedups for all the versions of the applications. On a real system, it is difficult to measure detailed performance metrics such as the time taken for computation, prefetching, synchronization and processor stalls, the cache hit rates and utilization, and the network traffic. However, the overall speedup measurement should give an accurate assessment of the effectiveness of the CCDP scheme.

5.3 Application Codes

For our experimental study, we selected two programs from the SPEC CFP95 and three programs from the SPEC CFP92 benchmark suites as our workload. The two applications codes from the SPEC CFP95 suite are TOMCATV and SWIM. The three programs from the SPEC CFP92 suite, namely, MXM, VPENTA, and CHOLSKY, are part of the NASA7 code. These applications are all floating-point intensive and they are written in Fortran. For all of the applications, we use their default data sets and problem sizes as specified by the benchmark suites.

TOMCATV is a highly vectorizable mesh generation program. It uses 7 matrices of size 513×513 as the main data structures. However, in the BASE version of TOMCATV, we need to expand the size of the shared matrices to 1024×1024 since the dimensions of shared arrays must be a power of 2 when we use the data sharing directive of CRAFT. This rule does not apply to the CCDP-0 and CCDP-1 versions, since they do not use the data sharing directive of CRAFT. The shared matrices and loop iterations in the BASE version are distributed using the *generalized distribution* [8] method, whereby blocks of matrix columns and loop iterations are assigned to PEs in a cyclic manner. If a *block distribution* method were to be used, half the PEs will not have useful work to do because the matrix columns 514 to 1024 do not store any useful data. The CCDP-0 and CCDP-1 versions of TOMCATV in turn follows a similar data and loop distribution method. The number of iterations computed by the program was set to 50. SWIM solves a system of shallow water equations using finite difference approximations. The main data structures used in the program are 14 513×513 matrices. It is a highly parallel code whose major loops are doubly nested, with the outer loops being parallel. We use the same strategy as in TOMCATV to partition the matrices and loop iterations in SWIM because it also uses shared matrices of size 513×513. We set the number of iterations executed in SWIM to 50.

The MXM application multiplies a 256×128 matrix by another 128×64 matrix. The middle loop of the triple-nested matrix multiply loop is parallel, while the outer loop is unrolled by a factor of 4. We distribute the columns of the shared matrices used by MXM among the PEs using a simple block distribution in all three versions of the program. To match the shared data distribution, the iterations of the middle parallel loop are also divided in a block distribution manner for all the versions of the program. The VPENTA application inverts three matrix pentadiagonals. The main data structures used are 7 matrices of size 128×128. Like in the case of MXM, we distribute the columns of the shared matrices used by VPENTA in a block distribution manner. The parallel loop iterations are block distributed accordingly. Finally, CHOLSKY performs the Cholesky decomposition on a set of input matrices. The main matrices used in the program are of size $250 \times 5 \times 41$ and $4 \times 250 \times 41$. The leading dimension of these matrices are block distributed to the PEs. Similarly, the iterations of the parallel loops are block distributed.

5.4 Performance Results

The speedups of the BASE, CCDP-0, and CCDP-1 codes over the sequential execution times of the applications are shown graphically in Figure 6. The sequential execution times were obtained by running the original Fortran codes on a single Cray T3D PE. Note that we could not execute the BASE version of SWIM using 1 PE, and thus the corresponding result is omitted. Also, the CCDP-0 and CCDP-1 versions of the MXM application are identical, and thus there is one common speedup curve for these two versions of MXM in Figure 6c.

5.4.1 Performance of the BASE version

First, let us examine the performance of the BASE version of the applications. These results demonstrate the performance delivered by the Cray T3D when we use its default software support for shared-address space, which relies on the simple but inefficient strategy of avoiding the cache coherence problem by not caching shared data.

The speedup achieved by the BASE version of TOMCATV is negligible. The TOMCATV code has a major doubly nested loop with parallel outer loop and serial inner loop. It also has another two major doubly nested loops with serial outer loop and parallel inner loop. When executing these three nested loops in the BASE version, each PE has to access shared data which reside in another PE. Therefore, the performance gains from parallelism are negated by the remote memory access costs.

SWIM is a highly parallelizable code with three major subroutines accounting for the bulk of its sequential execution time. Each of these subroutines contain a doubly nested loop of 512×512 iterations, such that the outer loop is parallel. The BASE version of the code performs quite well since the proportion of remote shared memory references in these major loops are relatively small compared to the total amount of data referenced.

For MXM, the BASE code does not exhibit much speedup even though the middle loop of the triple-nested matrix multiply loop is parallel. This is because during each iteration of the outermost loop, each PE accesses 4 columns of the shared input matrix A which are usually owned by a remote PE. The large amount of remote shared data references lead to large remote memory access costs, which diminishes the performance gains from parallelism.

VPENTA is a highly parallelizable code. Furthermore, during the execution of the program, each PE actually accesses only the portion of shared data which are stored in its local memory. Therefore, it is not surprising that the BASE version of VPENTA performs very well. However, there are some overheads associated with the shared data and computation distribution primitives in CRAFT. For larger number of PEs, the relative proportion of such overheads increases. As a result, the speedups obtained fall short of the ideal linear speedup.

Finally, the BASE version of CHOLSKY does not perform well for two reasons. The bulk of its execution time is spent in a complex nested loop, with parallel outermost loop and serial inner loops. However, the outermost parallel loop has only 4 iterations. Consequently, the amount of loop level parallelism that can be exploited in the code is small. In addition, it also incurs substantial remote shared memory access costs during the execution of the major nested loop.

5.4.2 Impact of Prefetching Potentially-Stale References

The percentage improvements in the execution times of the CCDP-0 codes over those of the BASE codes are shown in Table 1. These results are computed from the execution time measurements and they indicate the performance impact of caching shared data and using data prefetching for cache coherence enforcement on the Cray T3D.

In TOMCATV, during the execution of the three major doubly nested loops mentioned earlier, each PE accesses potentially-stale data which are owned by another PE. By prefetching the potentially-stale data references, the CCDP-0 version is able to achieve a large improvement of 44.8 to 68.5% over the execution time of the BASE version for the range of PEs used. This performance improvement can be attributed to the fact that the CCDP scheme enables the system to cache shared data and to prefetch the potentially-stale references, unlike in the case of the BASE version where the system has to access remote up-to-date shared data when they are needed.

For SWIM, the CCDP-0 version also provides performance improvement over the BASE implementation by caching shared data and reducing the shared memory access penalty through prefetching potentially-stale references. A significant improvement in execution time from 12.5 to 13.2% is achieved by the CCDP-0 version for the range of PEs used.

In MXM, when a PE executes an iteration of the outermost loop of the core triple-nested matrix multiply loop, it has to access 4 columns of the shared matrix A which might be owned by another PE. These are potentially-stale references which are prefetched in the CCDP-0 version. As a result, the CCDP-0 code outperforms the BASE version by 64.5 to 89.8% for the number of PEs used. Again, the CCDP scheme provides performance improvements by allowing shared data to be cached and also by prefetching the potentially-stale references to the matrix columns in advance.

As for VPENTA, since each PE only accesses the shared data in its local memory, the potentially-stale references in the CCDP-0 code are all satisfied by accessing data locally. However, since the CCDP scheme does not use the data sharing and loop distribution primitives in CRAFT, the overhead for the CCDP-0 version of VPENTA is lower than that of the BASE version. Consequently, the CCDP-0 code executes 4.4 to 23.9% faster than the BASE version for the number of PEs used.

In CHOLSKY, the CCDP-0 code provides a large performance improvement of 29.4 to 68.8% over that of the BASE code for the number of PEs used. However, the speedup attained by the CCDP-0 code decreases when the number of PEs used is 16 or more. When the number of PEs used exceeds 4, the bulk of the computation is actually performed by only 4 PEs since the main computationally intensive nested loop in CHOLSKY has an outermost parallel loop with 4 iterations. However, those 4 PEs will have to prefetch data which reside in other PEs. As the number of PEs increases, the costs of prefetching data will increase due to the larger distances between PEs and memory modules, which in turn causes the performance of the code to degrade.

5.4.3 Additional Impact of Prefetching Non-Stale References

Table 2 gives the percentage improvements in the execution times of the CCDP-1 codes over those of the CCDP-0 codes. These results illustrate the additional impact of prefetching non-stale references to hide memory latency, since the CCDP-1 codes also exploits data prefetching for cache coherence enforcement in the same manner as the CCDP-0 codes. We show the overall performance improvement attained by the CCDP scheme in Table 3. This is given by the improvement in execution time of the CCDP-1 codes over that of the BASE codes.

In TOMCATV, there is a significant amount of non-stale references in the major doubly nested loop. By exploiting data prefetching as an optimization to hide the memory latency of these non-stale references, the CCDP-1 version provides a substantial additional improvement of 4.1 to 29.3% in terms of execution time over that of the CCDP-0 version. Overall, the CCDP scheme reduces the execution time of the BASE version of TOMCATV by 47.1 to 77.7% for the number of PEs used.

Similarly, the three major doubly nested loops in SWIM also contain non-stale references. But most of these references made by each PE are directed at the local memory. Thus, they do not benefit from data prefetching as much as the remote non-stale memory references. However, the CCDP-1 version of SWIM still executes 2.6 to 10.6% faster than that of the

CCDP-0 version, depending on the number of PEs used. The overall performance improvement over the BASE version provided by the CCDP scheme is 15.1 to 22.3% for the number of PEs used.

In Table 2, the column for MXM contain "Not applicable (NA)" entries. In the innermost loop of the triple-nested matrix multiply loop, the data references to the A, B, and C matrices are all potentially-stale since the initial values of the elements of these shared matrices might have been assigned by another PE before the matrix multiply loop is executed. Hence, there are no non-stale references to be separately prefetched, which explains why the CCDP-1 version of MXM is actually the same as the CCDP-0 version. To summarize, the reduction in the execution time of MXM given by the CCDP scheme ranges from 64.5 to 89.8%.

As for VPENTA, its CCDP-1 version produces close to the ideal linear speedups. Compared to the CCDP-0 version, the CCDP-1 version provides a performance improvement of 0.9 to 3.0% over that of the CCDP-0 version for the number of PEs used. This additional performance improvement recorded by the CCDP-1 version is significant since the CCDP-0 version already performs very well. The CCDP scheme improves the overall performance of VPENTA by 5.6 to 26.2%.

Finally, the CCDP-1 version of CHOLSKY provides 0.5 to 25.1% improvement in execution time over that of the CCDP-0 version. As in the case of the CCDP-0 version, the performance of the CCDP-1 version of CHOLSKY degrades when the number of PEs used is 16 or more. This can also be explained by the relatively higher costs incurred by data prefetching when the number of PEs used is increased but the actual parallelism is limited. Overall, the performance improvement for CHOLSKY due to the CCDP scheme ranges from 47.1 to 68.9% for the number of PEs used.

5.5 Discussion

From the results of our experimental study, we learned several interesting and important lessons about cache coherence enforcement and memory latency hiding on large-scale non-cache-coherent DSM multiprocessors such as the Cray T3D. Our results clearly reaffirmed the fact that an important limiting factor in the performance of the Cray T3D is the remote shared-memory access latencies. The BASE version of applications such as SWIM and MXM, where the PEs access shared data which reside mainly in local memory, is scalable and performs well. On the other hand, the BASE version of applications such as TOMCATV, MXM, and CHOLSKY do not perform well since the bulk of their shared data references are directed at remote memory locations. As expected, the simple cache coherence strategy employed by the Cray T3D, in which shared data are not cached under CRAFT, is not effective in general.

We have shown that the CCDP-0 and CCDP-1 versions of the applications produce the same output as the original sequential codes on the Cray T3D. More importantly, the scheme also significantly enhances the performance of the Cray T3D by caching shared data and prefetching both the potentially-stale and the non-stale references to reduce remote shared-memory access latencies. Our results also indicate that data prefetching for cache coherence enforcement and memory latency hiding are complementary in nature, resulting in performance improvements when they are used together.

Through the experimental study, we also gained valuable experience with the existing support for software-initiated data prefetching on the Cray T3D. We find that the implementation of data prefetching on the Cray T3D is conservative in several aspects. First, its prefetch hardware cannot distinguish between the two types of prefetch operations used by the CCDP scheme. For vector prefetches of potentially-stale references, the *shmem_get* primitive invalidates the cache entries before performing the prefetches. However, for cache-line prefetches, we have to explicitly invalidate the cache entries corresponding to the prefetched data before performing the prefetches, which incurs additional software overhead. The Cray T3D prefetch hardware also does not support different priority levels for prefetching potentially-stale and non-stale references. Second, the overhead of interacting with the DTB annex and to extract data from the prefetch queue is significant. Finally, the prefetch queue can only be used to prefetch a small number of words at a time. A larger prefetch queue could increase the amount of overlap between prefetching and computation. Despite these limitations, the CCDP scheme is still able to provide significant performance improvements for the Cray T3D.

6 Conclusions

In this paper, we proposed a compiler-directed cache coherence scheme called the CCDP scheme which uses data prefetching for both cache coherence enforcement and memory latency hiding. Two types of data prefetching operations are required to prefetch the potentially-stale and the non-stale references in a parallel program. We discussed the compiler support required by the scheme; namely, stale reference analysis, prefetch target analysis, and prefetch scheduling. The compiler support for the scheme can be implemented using current parallelizing compiler technology.

We have also implemented the CCDP scheme on the Cray T3D and measured its performance impact on the system. Our experimental results show that the CCDP scheme can make use of the prefetch hardware support available on existing systems to correctly enforce cache coherence. More importantly, significant performance improvements were obtained for all of the applications studied. The results confirmed that prefetching data for cache coherence enforcement and for memory latency hiding are complementary optimizations. Finally, the CCDP scheme provides these performance improvements even though the prefetch hardware and software implementations on the Cray T3D are conservative. Thus, we believe that the CCDP scheme is a promising and practical scheme which can be implemented on future large-scale shared-memory multiprocessors. Conceptually, the CCDP scheme can also be used in systems with purely software support for shared address space and prefetching, such as networks of workstations.

A major factor which affects the performance of the CCDP scheme is the effectiveness of the prefetch hardware support. Although the prefetch hardware designs on existing systems can be used by the scheme, they are not optimized to handle the two types of data prefetching operations which are required by the scheme. Therefore, we are currently designing some prefetch hardware support for the CCDP scheme to further improve its performance. Our experience with the existing prefetch hardware support on the Cray T3D helps us to design the new prefetch hardware. We are also in the process of developing a compiler implementation of the prefetch target analysis and prefetch scheduling algorithms. This enables us to obtain a deeper understanding of the extent of performance improvements achievable by automatically transforming programs for cache coherence enforcement and memory latency hiding with the CCDP scheme.

Acknowledgments

We thank L. Choi for his contributions to the work on stale reference detection. We thank R. Numrich, K. Feind, G. Elsesser, and V. Ngo from Cray Research for providing information and help regarding the Cray T3D. Finally, we also thank the anonymous referees for their valuable comments and suggestions.

References

[1] R. Arpaci, D. Culler, A. Krishnammurthy, S. Steinberg, and K. Yelick. Empirical evaluation of the Cray T3D : A compiler perspective. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 320–331, June 1995.

[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] T.-F. Chen. *Data Prefetching for High-Performance Processors.* PhD thesis, University of Washington at Seattle, July 1993. Also available as U. Washington CS TR 93-07-01.

[4] H. Cheong and A. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[5] 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.

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

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

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

[9] Cray Research, Inc. *SHMEM User's Guide, Revision 2.0*, May 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] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

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

[14] 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.

[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] V. Karamcheti and A. Chien. A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 298–307, June 1995.

[17] Kendall Square Research Corporation. *KSR1 Principles of Operations*, 1992.

[18] D. Kuck et. al. The Cedar system and an initial performance study. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 213–223, May 1993.

[19] 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.

[20] 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.

[21] 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.

[22] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

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

[24] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[25] 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.

[26] R. Numrich. The Cray T3D address space and how to use it. Tech. report, Cray Research, Inc., August 1994.

[27] 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.

[28] 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.

[29] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989. Also available as Rice COMP TR 89-93.

[30] V. Santhanam, E. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

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

```
DOALL i = 1, n
      DO j = 1, n
            DO k = 1, n
                  C(i, j) = PS-ref (C(i, j)) + PS-ref (A(i, k)) *PS-ref (B(k, j))
            END DO
      END DO
END DOALL
```

Figure 1: Matrix multiplication example : Potentially stale references (marked as *PS-ref*).

```
DOALL i = 1, n
    DO j = 1, n
        ce-pref (C(i, j))
        ce-vecpref (A(i, 1:n))
        ce-vecpref (B(1:n, j))
        DO k = 1, n
            C(i, j) = C(i, j) + A(i, k) * B(k, j)
        END DO
    END DO
END DOALL
```

Figure 2: Matrix multiplication example : Cache coherence under CCDP scheme.

Prefetch Target Analysis Algorithm

Input: Set of potentially-stale references, PS.
Set of non-stale references, NS.
Output: Set of potentially-stale target references to be prefetched, $PSpref$.
Set of non-stale target references to be prefetched, $NSpref$.

let $PSpref = PS$,
$NSpref = NS$;
for each nested loop, **NL**, in the program **do**
eliminate potentially-stale references which are not located in the innermost loop of **NL** from $PSpref$, and these references will be issued as bypass-cache fetch operations;
eliminate non-stale references which are not located in the innermost loop of **NL** from $NSpref$, and these references will be issued as normal read operations;
endfor
for each inner loop or serial code segment, **LSC**, in the program **do**
let P_i = set of potentially-stale references enclosed by **LSC**, where $P_i \subset PS$,
N_i = set of non-stale references enclosed by **LSC**, where $N_i \subset NS$;
construct linear expressions for the addresses of references in P_i and N_i in terms of loop induction variables and constants;
mark the references in P_i and N_i which exhibit self-spatial locality;
detect presence of group-spatial locality amongst references in P_i and N_i;
if a group of references in P_i exhibit group-spatial locality **then**
determine the leading reference of the group;
eliminate the non-leading references in the group from $PSpref$, and these references will be issued as normal read operations;
endif
if a group of references in N_i exhibit group-spatial locality **then**
determine the leading reference of the group;
eliminate the non-leading references in the group from $NSpref$, and these references will be issued as normal read operations;
endif
endfor

Figure 3: The prefetch target analysis algorithm.

Prefetch Scheduling Algorithm

Scheduling techniques: Vector Prefetch Generation (VPG), Software Pipelining (SP), and Moving Back Prefetches (MBP).
Input: Set of potentially-stale target references to be prefetched, $PSpref$.
Set of non-stale target references to be prefetched, $NSpref$.

```
for each inner loop or serial code segment, LSC, in the program do
    let PST_i = set of potentially-stale prefetch target references enclosed by LSC, where PST_i ⊂ PSpref,
        NST_i = set of non-stale prefetch target references enclosed by LSC, where NST_i ⊂ NSpref,
        CET_i = set of ce-prefetch targets to be issued as prefetch operations, where CET_i ⊆ PST_i,
        LHT_i = set of lh-prefetch targets to be issued as prefetch operations, where LHT_i ⊆ NST_i;
    assign priority levels to prefetch targets in PST_i and NST_i;
    check whether prefetch hardware resources can handle all prefetch targets in PST_i and NST_i;
    if prefetch hardware resources are sufficient then
        set CET_i = PST_i and LHT_i = NST_i to issue all prefetch targets;
    else
        select prefetch targets for CET_i and LHT_i based on priority level;
        issue prefetch targets in PST_i which are not included in CET_i as bypass-cache fetch operations;
        issue prefetch targets in NST_i which are not included in LHT_i as normal read operations;
    endif
    switch (type of LSC)
            case 1 : LSC is a serial loop
                     if the loop bound is known then
                        schedule CET_i and then LHT_i using techniques in the order of (1) VPG, (2) SP,
                        and (3) MBP;
                     else
                        schedule CET_i and then LHT_i using techniques in the order of (1) SP and (2) MBP;
                     endif
                     if SP is used and LSC contains prefetch targets with self-spatial locality then
                        perform loop unrolling before SP to exploit self-spatial locality;
                     endif
            case 2 : LSC is a parallel DOALL loop with static scheduling
                     if the loop bound is known then
                        schedule CET_i and then LHT_i using techniques in the order of (1) VPG and (2) MBP;
                     else
                        schedule CET_i and then LHT_i using MBP;
                     endif
            case 3 : LSC is a parallel DOALL loop with dynamic scheduling
                     schedule CET_i and then LHT_i using MBP;
            case 4 : LSC is a serial code section
                     schedule CET_i and then LHT_i using MBP;
            case 5 : LSC is a loop which contains if-statements
                     schedule CET_i and then LHT_i using MBP, but do not move it beyond the boundary of
                     the if-part or else-part of the if-statements;
            case 6 : LSC is a loop or serial code segment within the body of an if-statement
                     apply case (1), (2), (3), or (4), but only prefetch within the if-part or else-part of
                     the if-statement;
    endswitch
endfor
```

Figure 4: The prefetch scheduling algorithm.

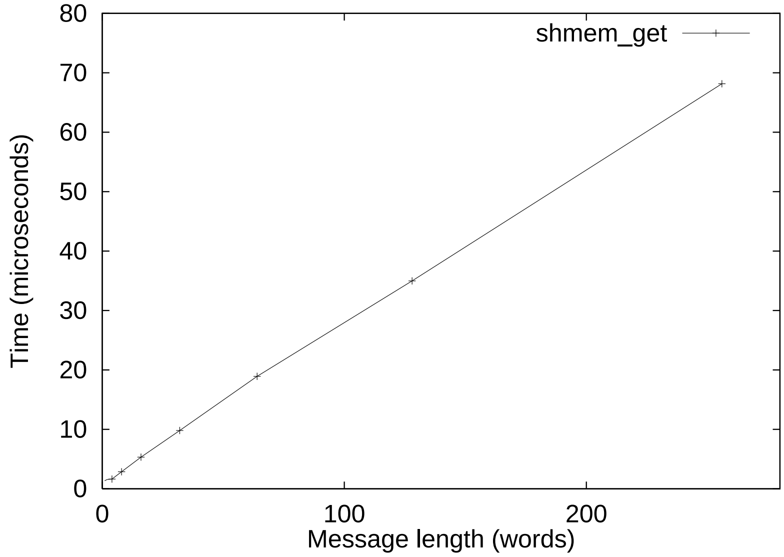
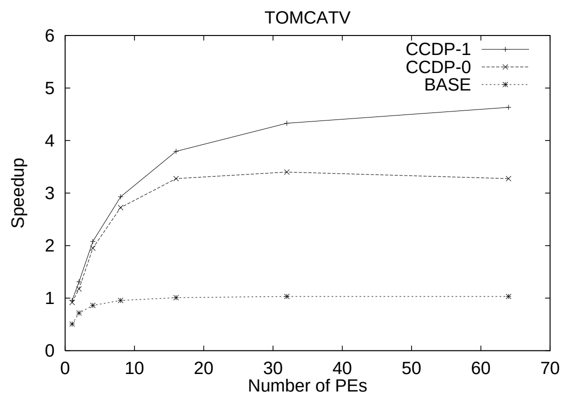
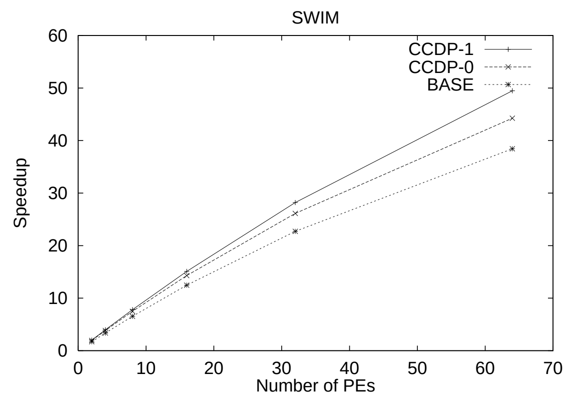


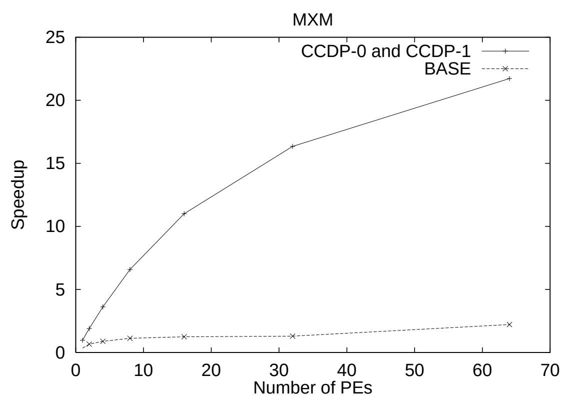
Figure 5: Time taken for *shmem_get* on the Cray T3D.



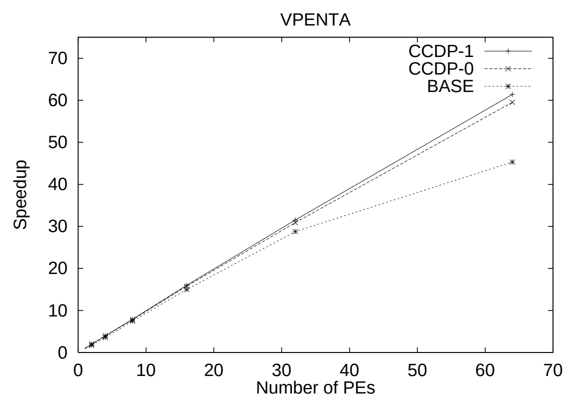
(a) Speedup for TOMCATV



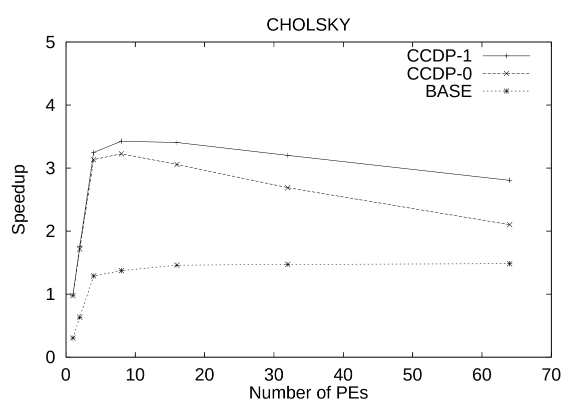
(b) Speedup for SWIM



(c) Speedup for MXM



(d) Speedup for VPENTA



(e) Speedup for CHOLSKY

Figure 6: Speedups for BASE, CCDP-0 and CCDP-1 codes.

Table 1: Improvement in execution time of CCDP-0 codes over BASE codes.

# PEs	TOMCATV	SWIM	MXM	VPENTA	CHOLSKY
1	44.83%	-	64.54%	12.53%	68.77%
2	38.97%	12.54%	65.25%	13.58%	63.05%
4	55.85%	12.50%	75.52%	9.23%	58.86%
8	64.91%	12.66%	82.96%	4.44%	57.46%
16	69.22%	12.75%	88.72%	4.98%	52.29%
32	69.64%	13.07%	92.07%	6.90%	45.27%
64	68.51%	13.16%	89.81%	23.90%	29.41%

Table 2: Improvement in execution time of CCDP-1 codes over CCDP-0 codes.

# PEs	TOMCATV	SWIM	MXM	VPENTA	CHOLSKY
1	4.06%	2.62%	NA	0.90%	0.48%
2	10.36%	2.91%	NA	0.88%	2.60%
4	6.14%	3.23%	NA	1.00%	3.53%
8	7.00%	4.03%	NA	1.20%	5.83%
16	13.66%	5.25%	NA	1.46%	10.23%
32	21.46%	7.32%	NA	1.96%	16.00%
64	29.30%	10.55%	NA	3.04%	25.07%

Table 3: Overall performance improvement by the CCDP scheme.

# PEs	TOMCATV	SWIM	MXM	VPENTA	CHOLSKY
1	47.07%	-	64.54%	13.31%	68.92%
2	45.30%	15.09%	65.25%	14.34%	64.01%
4	58.56%	15.33%	75.52%	10.13%	60.31%
8	67.36%	16.18%	82.96%	5.59%	59.94%
16	73.42%	17.33%	88.72%	6.36%	57.17%
32	76.16%	19.43%	92.07%	8.72%	54.03%
64	77.73%	22.33%	89.81%	26.22%	47.11%

Author Biographies

HOCK-BENG LIM received his B.S. in computer engineering and his M.S. in electrical engineering from the University of Illinois at Urbana-Champaign in 1989 and 1991, respectively. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. Previously, he was an engineer in the Defense Science Organization in Singapore from 1991 to 1993. From 1993 to 1994, he was an associate member of technical staff in the Information Technology Institute in Singapore. His research interests include computer architecture, parallelizing and optimizing compilers, and performance evaluation.

PEN-CHUNG YEW received his Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 1981. He has been a full professor in the Department of Computer Science, University of Minnesota, since 1994. Previously, he was an associate director of the Center for Supercomputing Research and Development, and an associate professor in the Department of Electrical and Computer Engineering and the Department of Computer Science 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, DC.

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 has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* and *Journal of Parallel and Distributed Computing*. He was also a distinguished visitor of the IEEE Computer Society. His research interests include high-performance multiprocessor system design, parallelizing compilers, computer architecture, and performance evaluation.