A Compiler-Directed Cache Coherence Scheme Using 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

Abstract

Cache coherence enforcement and memory latency reduction and hiding are very important problems in the design of large-scale shared-memory multiprocessors. In this paper, we propose a compiler-directed cache coherence scheme which makes use of data prefetching. The Cache Coherence with Data Prefetching (CCDP) scheme uses compiler analysis techniques to identify potentially-stale data references, which are references to invalid copies of cached data. The key idea of the CCDP scheme is to enforce cache coherence by prefetching the up-to-date data corresponding to these potentially-stale references from the main memory.

Application case studies were conducted to gain a quantitative idea of the performance potential of the CCDP scheme on a real system. We applied the CCDP scheme on four benchmark programs from the SPEC CFP95 and CFP92 suites, and executed them on the Cray T3D. The experimental results show that for the programs studied, our scheme provides significant performance improvements by caching shared data and reducing the remote shared-memory access penalty incurred by the programs.

1 Introduction

A major performance limitation in large-scale shared-memory multiprocessors is the large remote memory access latencies encountered by the processors. Private caches for processors have been used to reduce the number of main memory accesses by exploiting the locality of memory references in programs. However, the use of private caches leads to the classic *cache coherence problem*. *Compiler-directed cache coherence* schemes [4] offer a viable solution to the cache coherence problem for large-scale shared-memory multiprocessors. Although compiler-directed cache coherence schemes can improve multiprocessor cache performance, they cannot totally eliminate main memory accesses. Several techniques have been developed to hide memory latency in multiprocessors. In particular, much research has shown that *data prefetching* is effective in hiding memory latency and improving memory system performance.

In fact, data prefetching and compiler-directed cache coherence schemes may 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 multiprocessors. Our *Cache Coherence with Data Prefetching (CCDP)* scheme makes use of compiler analysis techniques to identify potentially-stale data references in a program. The scheme then enforces cache coherence by directing the processors to prefetch up-to-date copies of data corresponding to those potentially-stale references from the main memory. At the same time, data prefetching also provides the additional performance benefit of hiding memory latency.

The compiler algorithms used in the CCDP scheme include *stale reference analysis*, *prefetch target analysis*, and *prefetch scheduling*. These algorithms can be implemented using the compiler technology available today. We evaluated the potential performance of the scheme by applying it on four benchmark programs from the SPEC CFP95 and CFP92 suites, and executing the resulting programs on the Cray T3D. The results of our application case studies show that the CCDP scheme provided substantial performance improvements for the applications studied.

The rest of the paper is organized as follows. Section 2 gives an overview of previous work which are 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 application case studies to measure the potential performance of the CCDP scheme on the Cray T3D. We

*This work is supported in part by the National Science Foundation under Grant No. 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.

describe the target platform, the methodology of the study, the application codes used, and the experimental results obtained. Finally, we conclude in Section 6 and outline the future directions of our research.

2 Related Work

Several *hardware-supported compiler-directed (HSCD)* cache coherence schemes, which require hardware support to keep track of local cache states at run time, have been developed recently [4]. Although the HSCD schemes require less hardware support and are more scalable than the hardware directory cache coherence schemes, none of them have been implemented on real systems using off-the-shelf components yet. In several existing large-scale multiprocessors, the cache coherence problem is solved by not caching shared data at all or by caching shared data only when it is safe to do so [4]. However, this approach is too conservative and does not deliver very good performance.

Software-initiated data prefetching is also an active research area. Several prefetching algorithms have been proposed and implemented using experimental compilers [8, 10, 13]. The performances of these prefetching algorithms were evaluated by simulations. Recent efforts have focused on the implementation and performance evaluation of prefetching on real systems [2]. However, while these studies have examined the use of data prefetching to hide memory latency, they have not explored the use of data prefetching to implement and optimize a compiler-directed cache coherence scheme.

3 Cache Coherence Enforcement by Data Prefetching

Cache coherence schemes are used to ensure that the processors will 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*. If the need to access these up-to-date shared data can be predicted in advance, then the processors can issue prefetch operations to bring these data into the caches before they are actually used. In compiler-directed cache coherence schemes, this prediction is done by identifying the potentially-stale data references using compiler analyses. Data prefetching provides the additional benefit of memory latency hiding by overlapping the fetching of remote up-to-date data with computation.

The *Cache Coherence with Data Prefetching (CCDP)* scheme is applicable to large scale non- cache-coherent shared-memory multiprocessors which have hardware support for data prefetching. Our scheme would allow such systems to cache shared data and improve performance without requiring additional 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 some system-level prefetch hardware support. 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.

3.1 Parallel Execution Model

We assume that a parallel program can be explicitly (i.e. by the programmer) or implicitly (i.e. by the compiler) partitioned 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 comprise 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.

3.2 Overview of the CCDP Scheme

The CCDP scheme consists of three major steps :

1. **Stale reference analysis**. The compiler identifies the potentially-stale data references in a parallel program by using several program analysis techniques. Those references which are not potentially-stale will be the normal read references.

2. **Prefetch target analysis**. The potentially-stale data references are the possible references to be prefetched. However, it might not be necessary or worthwhile to prefetch all of these references. Thus, the prefetch target analysis step determines the potentially-stale references which should be prefetched. This minimizes the number of unnecessary prefetches, which should actually be normal read references.

3. **Prefetch scheduling**. Having identified the potentially-stale data references which are suitable targets for prefetching, we need to schedule these prefetch operations. The prefetch scheduling algorithm inserts the prefetch operations at appropriate locations in the program.

In addition to these steps, the CCDP scheme must also ensure that the program will execute correctly. This means that the prefetch operations for potentially-stale references should not violate cache coherence. If there is no special prefetch hardware support on the system to prevent the processors from accessing potentially-stale data in the caches while the prefetch operations are still in progress, then the cache entries should be invalidated before the prefetches are issued. Also, if the prefetch operations are dropped due to lack of prefetch hardware resources, then the CCDP scheme should use bypass-cache fetch operations to read the up-to-date data directly from the main memory.

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 by Choi and Yew [3], and they have been implemented using the Polaris parallelizing compiler [12]. We make use of these algorithms in the CCDP scheme. The interested reader can refer to [3] for the details of these algorithms.

4.2 Prefetch Target Analysis

Our prefetch target analysis algorithm makes use of simple heuristics which are easy to implement and are likely to be effective. The algorithm starts by including all potentially-stale references of the program in the target set for prefetching. As the algorithm proceeds, the potentially-stale references which should not be prefetched are removed from the set. Thus, when the algorithm terminates, the resulting set will contain the potentially-stale references which should be prefetched. The algorithm is shown in Figure 1.

Prefetch Target Analysis Algorithm

Input: Set of potentially-stale references, P.
Output: Set of potentially-stale references to be prefetched, S.

let $S = P$;
examine all nested loops in the program and eliminate the potentially-stale references which are not located in the innermost loop from S;
for each inner loop or serial code segment, LSC, in the program **do**
  let P_i = set of potentially-stale references enclosed by LSC;
  construct linear expressions for the addresses of references in P_i in terms of loop induction variables and constants;
  detect presence of group-spatial locality amongst references in P_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 S, and these references will be issued as normal read operations;
  endif
endfor

Figure 1. Prefetch target analysis algorithm.

Like several previous prefetching algorithms [2, 10], our prefetch target analysis algorithm focuses on the potentially-stale references in the inner loops, where prefetching is most likely to be beneficial. Our algorithm also exploits spatial reuses to eliminate some unnecessary prefetch operations. We only need to prefetch the leading reference in a group of references which exhibit *group-spatial* locality. 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, 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 (such as byte, word, double word, etc), 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 a linear expression, our algorithm will conservatively treat them as references to be prefetched.

Note that it is possible to further reduce the number of unnecessary prefetch operations by also exploiting *group-temporal*, *self-spatial*, and *self-temporal* localities. However, this would require additional compiler analyses and transformations, such as the estimation of *loop volume* and loop unrolling [2, 10]. Some arbitrary decisions and approximations must be made to handle complications such as unknown loop bounds.

4.3 Prefetch Scheduling

Our prefetch scheduling algorithm can generate 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.

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 should improve the effectiveness of the prefetch operations, so that the prefetched data will often arrive in time. Third, the prefetch scheduling algorithm should take into account important hardware constraints and architectural parameters of the system, such as the size of the data cache in each processor, the size of the prefetch queue or issue buffer for each processor, the maximum number of outstanding prefetches allowed by the processor, and the average memory latency for a prefetch operation. Finally, the prefetch scheduling algorithm should try to minimize the prefetch overhead.

4.3.2 Scheduling techniques

Our prefetch scheduling algorithm makes use of three scheduling techniques : *vector prefetch generation*, *software pipelining*, and *moving back prefetches*.

Vector prefetch generation Gornish [8] developed an algorithm for conservatively determining the earliest point in a program during 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, the 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 cache size and the size of the prefetch queue or issue buffer. 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 [8]. We modify the algorithm by imposing a restriction on the number of loop levels that array references should be pulled out from, in order to maximize the effectiveness of the vector prefetches. Our algorithm pulls out an array reference one loop level at a time. It then constructs a vector prefetch operation and check if the number of words to be prefetched will exceed the cache size or the available prefetch queue size. The vector prefetch operation will be issued only if these hardware constraints are satisfied and that 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 [10], *software pipelining* is used to schedule cache-line prefetch operations. Software pipelining is an effective scheduling strategy to hide memory latency by overlapping the prefetches for a future iteration of a loop with the computation of the current iteration. We adapt the software pipelining algorithm to suit the CCDP scheme. First, in order to simplify the computation of the loop execution time, we impose a restriction that software pipelining will only be used for inner loops which do not contain recursive procedure calls. The compiler can compute the loop execution time since the number of clock cycles taken by each instruction is known. After computing the number of iterations to prefetch ahead, the compiler has to decide whether it is profitable to use software pipelining. This is a design issue which is machine dependent. Our algorithm uses a compiler parameter which specifies 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 dropping the prefetches when the amount of data to be prefetched exceed the cache size or the available prefetch queue size.

Moving back prefetches If both vector prefetch generation and software pipelining cannot 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, if we prefetch the potentially-stale references in certain loops using vector prefetch generation or software pipelining, the prefetch hardware resources might be exceeded. 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 [8]. The original algorithm tries to move references as far back as possible subjected to control and data dependence constraints. However, the algorithm might move a prefetch operation so far back that the prefetched data might be replaced from the cache by the time it is used. Another situation which might arise is that the distance which a prefetch operation can be moved back is so small that the prefetched data might not 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 by using experimental study on the desired system.

4.3.3 Prefetch scheduling algorithm

Our prefetch scheduling algorithm combines the strengths of Gornish's and Mowry's approaches. It considers each inner loop or serial code segment of the program. Depending on the type of loop or code segment, it attempts to make a good engineering decision to use a suitable scheduling technique for the prefetch targets within the loop or code segment. The algorithm is shown in Figure 2.

Prefetch Scheduling Algorithm

Techniques: Vector Prefetch Generation (VPG), Software Pipelining (SP), and Moving Back Prefetches (MBP).
Input: Set of potentially-stale references to be prefetched, S.

for each inner loop or serial code segment, LSC, in the program **do**
  let T_i = set of prefetch targets enclosed by LSC, $T_i \subset S$;
  switch (type of LSC)
  case 1 : LSC is a serial loop
      if the loop bound is known **then**
        schedule T_i using techniques in the order of (1) VPG, (2) SP, and (3) MBP;
      else
        schedule T_i using techniques in the order of (1) SP and (2) MBP;
      endif
  case 2 : LSC is a parallel DOALL loop with static scheduling
      if the loop bound is known **then**
        schedule T_i using techniques in the order of (1) VPG and (2) MBP;
      else
        schedule T_i using MBP;
      endif
  case 3 : LSC is a parallel DOALL loop with dynamic scheduling
      schedule T_i using MBP;
  case 4 : LSC is a serial code section
      schedule T_i using MBP;
  case 5 : LSC is a loop which contains if-statements
      schedule T_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 2. Prefetch scheduling algorithm.

5 Application Case Studies

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 [5]. We adapted the CCDP scheme to suit the hardware constraints and architectural parameters of the system.

5.1 Target Platform

The Cray T3D is physically distributed memory MPP system which provides hardware support for a shared address space [11]. 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 extracts the prefetched word from the queue when it is needed. The prefetch queue can only store 16 words of data. Previous studies [1, 9] 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 [6] 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. 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 [11]. Second, the programmer can explicitly perform remote memory reads and writes in a cached or non-cached manner. Finally, the Cray T3D also provides a high-level shared-memory communication library called the *SHMEM library* [7]. The *shmem_get* primitive in the library provides similar functionality as a vector prefetch.

5.2 Methodology

We first parallelize the application codes using the Polaris compiler [12]. Then, we manually convert the parallelized codes into two versions of CRAFT programs. The baseline version, which we call the BASE version, makes use of the default software support for shared address space available in CRAFT. It is important to note that the BASE codes do not cache shared data, and thus they do not violate cache coherence. Furthermore, they incur full memory access latencies for all references to remote shared data. On the other hand, the optimized version of the codes, which we call the CCDP version, allows the caching of shared data and makes use of the CCDP scheme to maintain cache coherence.

We use the following procedure to produce the two versions of codes. First, we have to distribute shared data to the PEs. In the BASE codes, we use the shared data distribution directives available in CRAFT directly. For the CCDP codes, a shared array is divided into equal portions and each portion is stored in a PE's local memory. Since the array is not declared as a CRAFT shared variable, it can be cached by each PE. Next, the DOALL loop iterations are distributed amongst the PEs. For the BASE codes, a DOALL loop is directly converted into a *doshared* loop. Instead of using the

# PEs	MXM		VPENTA		TOMCATV		SWIM	
	BASE	CCDP	BASE	CCDP	BASE	CCDP	BASE	CCDP
1	0.34	0.97	0.87	0.99	0.51	0.92	-	0.98
2	0.66	1.90	1.71	1.98	0.72	1.17	1.70	1.95
4	0.88	3.61	3.58	3.94	0.86	1.95	3.36	3.84
8	1.12	6.58	7.45	7.80	0.96	2.72	6.55	7.50
16	1.24	11.01	14.98	15.76	1.01	3.28	12.48	14.30
32	1.30	16.34	28.77	30.90	1.03	3.40	22.72	26.13
64	2.21	21.73	45.30	59.53	1.03	3.28	38.43	44.25

Table 1. Speedups over sequential execution time.

doshared directive, each CCDP code assigns loop iterations directly to the PEs using a similar loop scheduling policy as the corresponding BASE code. Finally, we use a Polaris implementation of the stale reference analysis algorithms [3] to identify the potentially-stale references in the programs. Then, we manually select prefetch targets and insert prefetch operations into the CCDP codes according to the prefetch target analysis and prefetch scheduling algorithms of the CCDP scheme.

5.3 Application Codes

We selected four benchmark programs from the SPEC CFP92 and CFP95 floating-point benchmark suites for our study. The application codes from SPEC CFP92 are MXM and VPENTA. They are numerical kernels in the NASA7 collection. The application codes from SPEC CFP95 are TOMCATV and SWIM. All the programs are written in Fortran. In all of the applications, we use the full input data set sizes as specified by the benchmark suites.

MXM multiplies a 256×128 matrix by another 128×64 matrix. In this code, the columns of the shared matrices are distributed amongst the PEs in a block distribution manner. To match the shared data distribution, the iterations of the middle parallel loop are also divided in a block distribution manner for both the BASE and CCDP version of MXM. VPENTA inverts three matrix pentadiagonals. It uses 7 shared matrices of size 128×128. As in the case of MXM, we also distribute the columns of the shared matrices used by VPENTA in a block distribution manner. The parallel loop iterations are block distributed accordingly.

TOMCATV is a highly vectorizable mesh generation program. The main data structures it used are 7 shared matrices of size 513×513. We use the generalized distribution [6] directive to distribute the shared matrices and loop iterations in the BASE version of TOMCATV. The CCDP version of TOMCATV in turn follows a similar data and loop distribution method. Last but not least, SWIM solves a system of shallow water equations using finite difference approximations. It is a highly parallel code whose major loops are doubly-nested, whereby the outer loops are parallel. Since SWIM also makes use of shared matrices (14 of them) of size 513×513, we follow the same strategy used in TOMCATV to partition the matrices and loop iterations. In both TOMCATV and SWIM, we set the number of iterations executed to 50.

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

Table 2. Improvement in execution time of CCDP codes over BASE codes.

5.4 Experimental Results

Table 1 shows the speedups of the BASE and CCDP codes over the sequential execution times of the four applications. The percentage improvements in execution time of the CCDP codes over the BASE codes are presented in Table 2.

For MXM, the BASE code does not exhibit much speedup even though the middle loop of the triple-nested matrix multiply loop is parallel. In each iteration of the outermost loop, each PE accesses 4 columns of the input matrix A, which are usually owned by a remote PE. Thus, the BASE version of MXM incurs large remote memory latencies, which negates the performance gains through parallelism. By using the CCDP scheme, we can achieve a much better speedup and a performance improvement of 64.5% to 89.8% depending on the number of PEs used.

VPENTA is a highly parallelizable code. Furthermore, during the execution of the program, each PE will only access the portion of shared data which is stored in its local memory. Therefore, it is not surprising that the BASE code performs quite well. However, there is overhead associated with the shared data and computation distribution primitives in CRAFT. For larger number of PEs, the relative proportion of such overhead increases. For the CCDP code, the potentially-stale references of each PE also access data locally. However, the overhead of the CCDP scheme is lower than those of the BASE code. As a result, the CCDP scheme gives a performance improvement of 4.4% to 23.9% for the number of PEs used, as well as close to ideal linear speedups.

The BASE version of TOMCATV does not perform very well. It has a major doubly-nested loop (loop 60) with parallel outer loop, and two major doubly-nested loops (loops 100 and 120) with parallel inner loop and serial outer loop. When executing these three nested loops, each PE has to access shared data which are owned by another PE. By caching shared data and prefetching potentially-stale references, the CCDP scheme is able to achieve a significant performance gain of 44.8% to 68.5% for the range of PEs used.

Finally, SWIM is a highly parallelizable code with three major subroutines. Each of these subroutines contain a doubly-nested loop with its outer loop being parallel. The performance of the BASE code is quite good because the proportion of remote shared memory references in these major loops are relatively small compared to the total amount of data referenced. As usual, the CCDP scheme provides performance improvement over the BASE implementation. The improvement in execution time due to the CCDP scheme is 2.5% to 13.2% for the range of PEs used.

The results of our application case studies indicate that the CCDP scheme can correctly enforce cache coherence, and also significantly enhance the performance of the Cray T3D by enabling it to cache shared data and to prefetch potentially-stale references to reduce remote shared-memory access latencies.

6 Conclusions

In this paper, we proposed a compiler-directed cache coherence scheme called the CCDP scheme which enforces cache coherence through data prefetching. We discussed the compiler support required by the scheme; namely, stale reference analysis, prefetch target analysis and prefetch scheduling. We also conducted application case studies on the Cray T3D to measure the potential performance of the scheme. Significant performance improvements were obtained for the applications studied.

The present CCDP scheme only prefetches the potentially-stale references. Intuitively, we should be able to obtain further performance improvement by prefetching the non-stale references as well. In future, we plan to do a full compiler implementation of the prefetch target analysis and prefetch scheduling algorithms so that programs can be automatically analyzed and transformed for cache coherence. We will also perform detailed simulation studies to evaluate the performance of the CCDP scheme and the interaction of the compiler implementation with various important architectural parameters.

Acknowledgements

We thank L. Choi for the stale reference detection implementation. We thank R. Numrich, K. Feind, G. Elsesser and V. Ngo from Cray Research for providing information and help. We also thank the reviewers for their comments.

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

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

[5] Cray Research, Inc. *Cray T3D System Architecture Overview*, Mar. 1993.

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

[7] Cray Research, Inc. *SHMEM User's Guide, Revision 2.0*, May 1994.

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

[9] 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.

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

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

[12] 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.

[13] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.