Compiler Support for Maintaining Cache Coherence Using Data Prefetching *
(Extended Abstract)

Hock-Beng Lim[1], Lynn Choi[2] and Pen-Chung Yew[3]

[1] Center for Supercomputing R & D, Univ. of Illinois, Urbana, IL 61801
[2] Microprocessor Group, Intel Corporation, Santa Clara, CA 95095
[3] Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN 55455

1 Introduction and Motivation

A major performance limitation in large-scale shared-memory multiprocessors is the large remote main memory latencies encountered by the processors. Private caches for processors have been used to reduce the number of main memory accesses. However, the use of private caches leads to the classic *cache coherence problem*. *Compiler-directed cache coherence* schemes [2] offer a viable solution to this 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. Thus, *data prefetching* schemes have been developed to hide the memory latency.

Actually, data prefetching and compiler-directed cache coherence schemes may be combined in a complementary manner. We propose a software-only compiler-directed cache coherence scheme that incorporates data prefetching. The *Cache Coherence with Data Prefetching (CCDP)* scheme is designed for large-scale non- cache-coherent shared-memory multiprocessors that have hardware support for data prefetching. Our scheme would enable such systems to cache shared data without requiring additional hardware support.

2 Cache Coherence Enforcement by Data Prefetching

Cache coherence schemes are used to ensure that the processors will always access the most up-to-date copy of shared data. 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 using compiler analyses to identify the potentially stale data references. Data prefetching provides the additional benefit of memory latency hiding by overlapping the fetching of remote up-to-date data with computation.

* 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.

3 Compiler Support for the CCDP Scheme

3.1 Stale Reference Analysis

The stale reference analysis algorithm identifies the potentially stale data references in a program by detecting stale reference sequences. Three main program analysis techniques are used in stale reference analysis : *stale reference detection*, *array data-flow analysis*, and *interprocedural analysis* [1].

3.2 Prefetch Target Analysis

Since prefetch operations introduce instruction execution and network traffic overheads, it is important to minimize the number of unnecessary prefetches. The prefetch target analysis algorithm determines which potentially stale references should be prefetched. The remaining potentially stale references that are not prefetched will then be issued as either normal read operations or bypass-cache fetch operations. Our prefetch target analysis algorithm focuses on the potentially stale references in the inner loops of a program, where prefetching is most likely to contribute performance benefits. It also exploits self-spatial locality in the memory references of the program to eliminate some unnecessary prefetch operations.

3.3 Prefetch Scheduling

After identifying which potentially stale references should be prefetched, the next important task is to schedule these prefetches. There are four main design considerations for the prefetch scheduling algorithm.

First, our prefetch scheduling algorithm should ensure that the correctness of the program's memory references is not violated as a result of the prefetch operations. Second, the algorithm should improve the effectiveness of the prefetch operations so that the prefetched data will often arrive in the caches in time to be used. Third, the 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 associated with each processor, the maximum number of outstanding prefetches allowed by the processor, and the average memory latency for a prefetch operation. Last but not least, since prefetch operations incur a significant overhead, the algorithm should try to minimize the prefetch overhead.

Our prefetch scheduling algorithm considers each inner loop and serial code section of the program, and attempts to make a good engineering decision to use one of the following prefetch scheduling techniques :

- **Vector prefetch generation**
  Gornish developed an algorithm that can be used to *pull out* [3] array references from loops for prefetching. We adapted his approach for the CCDP

scheme by imposing a restriction on the number of loop levels from which array references should be pulled, in order to maximize the effectiveness of the vector prefetches. We also check if the amount of data to be prefetched will exceed the available prefetch queue size or the cache size. A vector prefetch will be issued only if these hardware constraints are satisfied.

– **Software pipelining**
  Mowry [4] used software pipelining to schedule single cache-line prefetch operations. We adapted the software pipelining algorithm to suit the CCDP scheme. First, we impose a restriction that software pipelining will be used only for inner loops that do not contain branches or recursive procedure calls. Next, the number of iterations to prefetch ahead is computed and the compiler uses this information to decide whether it is profitable to use software pipelining. Finally, our algorithm considers the hardware constraints by dropping the prefetch operations when the prefetch queue is full or if the amount of data to be prefetched exceed the cache size.
– **Moving back prefetches**
  If neither vector prefetch generation nor software pipelining can be applied, then our algorithm tries to move the prefetch operations as far away as possible from the point where the data will be used. This is done in the same manner as Gornish's algorithm to *pull back* [3] references. To ensure correctness and 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 range of distances to move back the prefetch operation.

4 Present and Future Work

At present, we are conducting application case studies to get quantitative measurements of the performance improvements provided by the CCDP scheme on the Cray T3D. We manually insert prefetch operations into several application benchmarks according to the CCDP scheme. In the future, we plan to do a full compiler implementation of the scheme and to study the interaction of the compiler implementation with various important architectural parameters.

References

1. L. Choi. *Hardware and Compiler Support for Cache Coherence in Large-Scale Multiprocessors.* PhD thesis, Center for Supercomputing R & D, UIUC, March 1996.
2. L. Choi, H.-B. Lim, and P.-C. Yew. Multiprocessor cache coherence : A compiler-directed approach. *To appear in IEEE Parallel & Distributed Tech.*, Winter 1996.
3. E. Gornish. Compile time analysis for data prefetching. Master's thesis, Center for Supercomputing R & D, UIUC, December 1989.
4. T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Dept. of Elect. Eng., Stanford University, March 1994.