An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors *

Edward H. Gornish Alexander Veidenbaum

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61801
gornish@csrd.uiuc.edu

Abstract

Both hardware and software prefetching have been shown to be effective in tolerating the large memory latencies inherent in in in shared-memory multiprocessors; however, both types of prefetching have their shortcomings. In this paper, we propose an integrated hardware/software prefetching method that uses simple hardware that can handle most data accesses and software prefetching for the few remaining accesses. This yields an effective scheme that minimizes both CPU overhead and hardware costs. Execution-driven simulations show our method to be very effective.

1 Introduction

Memory latency has always been a major issue in shared-memory multiprocessors. This is even more true as the gap between processor and memory speeds continues to grow. In order to fully utilize such systems, it is essential to use the memory hierarchy effectively, in order to reduce memory latency. In this paper, we study how data prefetching into the first-level cache can eliminate cache misses. In addition, our techniques can generalize to data prefetching in other levels of the memory hierarchy.

There are two main classes of data prefetching. In hardware prefetching, the hardware alone decides what data to prefetch and when and where to prefetch the data. In software prefetching, the hardware supports a prefetching instruction. The user, or compiler, then directs prefetching by inserting prefetching instructions into the code. Both hardware and software prefetching have been studied extensively, and have been shown to be effective; however, both types of prefetching have their shortcomings. For example, hardware prefetching can require complex and expensive hardware, while software prefetching requires extra CPU instructions. In this paper, we propose a method for integrated hardware/software prefetching. Our method uses simple hardware that can handle most data accesses and software prefetching for the few remaining accesses. This yields an effective scheme that minimizes both CPU overhead and hardware costs.

In order to effectively evaluate our integrated prefetching method, we analyze the performance of our scheme using a multiprocessor simulator that accurately models the memory subsystem, including network contention.

The remainder of this paper will be organized as follows. First we present related work in both hardware and software prefetching in Section 2. We then present our integrated prefetching scheme in Section 3. In Section 4, we present the methodology used in this study, and in Section 5, we present our results.

*This work was supported by NASA Ames Research Center under Grant No. NASA NCC 2-559, the Department of Energy under Grant No. DE-FG02-85ER25001, the National Science Foundation under Grant No. MIP-8920891, and an Intel Foundation Graduate Fellowship.

2 Related Work

Hardware Prefetching Smith [1] discusses *one block lookahead (OBL)* prefetching schemes (i.e., upon a reference to line l, line $l+1$ is considered for prefetching).

Two problems with OBL methods are that they might not prefetch data early enough to tolerate memory latencies and that they cannot deal with strides larger than the size of the cache line. Jouppi [2] addresses the former problem. He presents a scheme based on multiple prefetch buffers, that can prefetch data with a lookahead greater than one. The latter problem is addressed in several similar schemes proposed by Baer and Chen [3], Fu, Patel and Janssens [4] and Jegou and Temam [5]. These schemes use tables that keep track of the access history of a load instruction in an attempt to predict the reference's stride.

Dahlgren, Dubois and Stenstrom [6] present a prefetching method that varies the size of a block that is prefetched on a miss, depending on what percentage of prefetched data actually gets used.

Software Prefetching The most extensive software prefetching study was done by Mowry, Lam and Gupta [7]. They developed an extensive compiler algorithm for prefetching data into both levels of a two-level cache hierarchy. Callahan, Kennedy and Porterfield [8] presented a similar, but less sophisticated, scheme. Chen et al. [9] and Klaiber and Levy [10] proposed prefetching into a separate prefetch buffer in order to reduce cache contention. Gornish, Granston and Veidenbaum [11] developed a compiler algorithm to determine the earliest time, in a program's execution, that data can be prefetched.

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-9/0-6 $20.00 © 1994 IEEE

They showed that prefetching has tremendous potential in a vector multiprocessor system.

Integrated Prefetching There is only one other proposal for an integrated scheme that we are aware of. Chen [12] describes a scheme where software prefetching is used to prefetch large chunks of data from global memory to the second level cache. His original hardware algorithm ([3] described above) is then used to prefetch data from the second level cache to the first level cache. This scheme differs from the one that we present in this paper, in that both software and hardware prefetching prefetch data into the first level cache, in our case.

3 Integrated Hardware and Software Prefetching

We now summarize some of the differences between hardware and software prefetching as a motivation for our integrated prefetching scheme.

Software schemes requires less hardware support than hardware schemes. However, software schemes requires many more CPU instructions; therefore, code size increases both statically and dynamically. Software schemes can often determine, at compile time, data access strides and the appropriate prefetching lookahead. Hardware schemes have to get progressively more complex to be able to handle large strides and to increase the prefetching lookahead. Software schemes can also handle irregular accesses, such as those generated by references with subscripted subscripts.

Based on this discussion, we now present an integrated hardware/software prefetching scheme that combines the best aspects of both types of prefetching.

Our method is geared to array accesses in loops. We classify such accesses by the type of access stream they form. In a reference such as $A(ci)$, where i is the loop index and c is a constant, we say that the elements of A that are referenced form a *constant stride access stream (CSAS)*. Assume that the cache line size is L. If $c \leq L$, we say that the reference generates a *small constant stride access stream (SCSAS)*. If $c > L$, then we say that the reference generates a *large constant stride access stream (LSCAS)*. In a reference such as $A(f(i))$, where the stride is unpredictable, we say that the reference generates a *non-constant stride access stream (NCSAS)*.

Our goal is to provide the simplest hardware possible that can still handle the bulk of the prefetching possibilities. All other cases are handled in software. This way we minimize both the hardware costs and the number of CPU prefetching instructions that have to be executed. To this end, we divide up the prefetching tasks as follows. Software prefetching is responsible for LCSASs, NCSASs and the first accesses of an SCSAS, while hardware prefetching is responsible for subsequent accesses of an SCSAS.

Our method is based upon the concept of the *prefetching degree*. We define the *prefetching degree (PD)* as the number of lines in advance of the first reference to a line, l, that l is prefetched[1]. That is, line $l + PD$ of an access stream would be prefetched during an access to line l. An OBL scheme always uses $PD = 1$.

We now describe the necessary hardware and software support for our integrated data prefetching method.

3.1 Hardware Support

The hardware mechanism to support this scheme is based on the tagged prefetching (OBL) scheme (i.e., on the first access to line l, line $l+1$ is prefetched). However, we extend the basic scheme to support a prefetching degree greater than one. This is accomplished as follows. Each cache line, l, has an additional field designated DEGREE(l). In addition, the software prefetching instruction takes a second operand that specifies a prefetching degree. When a line, l, is prefetched, the prefetching degree specified by the prefetching instruction is stored in DEGREE(l). Upon the first demand access to line l, line $l+$DEGREE(l) is prefetched, and DEGREE($l +$ DEGREE(l)) is set to DEGREE(l). In this way the prefetching degree gets propagated throughout the lifetime of an access stream. In addition, the prefetching degree specified by the last prefetching instruction is saved in a location designated DEFAULTDEGREE. In the event that a read access misses in the cache, the DEGREE field of the new line gets set to DEFAULTDEGREE.

3.2 Compiler Support

It is the job of the compiler to determine which data will be automatically prefetched by the hardware and which data need to be prefetched by explicit software instructions. There are three basic steps to the compiler analysis:

1. Access Stream Detection
2. Memory Latency Analysis
3. Temporal Locality Analysis

Access Stream Detection Access streams and their types are determined in this phase. In the case of a CSAS, spatial locality analysis is used to determine if the access stream is an SCSAS or an LCSAS. Prefetches are generated for each reference of an LCSAS or an NCSAS. In the case of an SCSAS, prefetches are generated as described below in the **Memory Latency Analysis** phase.

Redundant software prefetches are eliminated by detecting data reuse and locality—i.e. when the same datum is used more than once and will not be replaced before the subsequent uses. In this case, the datum only needs to be prefetched once[2]. Using reuse analysis, the compiler can also determine that the elements used by two different references are actually part of the same access stream.

[1] The term prefetching degree is used differently from its use in [6].

[2] Redundant hardware prefetches can be eliminated if the cache is first checked to see if the requested data is already present before issuing a prefetch.

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-9/0-6 $20.00 © 1994IEEE

Memory Latency Analysis In this step, PD is calculated for each SCSAS detected in the previous step. The prefetching degree for stream a can be calculated as follows. Given a cache line size, L, and a stride, $S(a)$, a new cache line is needed every L/S iterations. Therefore, given an iteration time of $T_I(a)$ and a memory latency of T_M:

$$PD(a) = T_M S(a)/T_I(a)L \quad (1)$$

The prefetching degree can vary from program to program, and even within the same program depending on the strides of the different access streams and the iteration times of the surrounding loops.

The compiler generates software instructions to prefetch the first PD elements of an SCSAS, and it passes the value PD to the hardware, as an operand of the prefetching instruction, as described above. In this way the prefetching degree is determined separately for each SCSAS.

Temporal Locality Analysis A hardware prefetch of a particular cache line is triggered by the use of a previous cache line. If there are too many intervening data references between the use of the two lines, then the prefetched data might already be replaced before it is needed. In this case, it would be necessary to reschedule the appropriate prefetches or issue additional prefetches.

We have incorporated this compile time framework into the PARAFRASE compiler [13].

4 Experimental Methodology

In order to evaluate the performance of our prefetching schemes, we have developed a detailed architecture simulator of a shared-memory multiprocessor. Processing nodes are connected to shared memory nodes through forward and reverse omega networks. Network contention is accurately modeled.

Each processing node consists of a processor, a 16K direct-mapped data cache, an instruction cache, and a fetch unit. The processor models a RISC architecture. The timings for floating point operations are based on the MIPS R3000. All other instructions execute in one cycle. All instructions are assumed to hit in the instruction cache.

The cache uses the write-through write-allocate no-fetch policy, and a software mechanism is used to maintain cache coherence. Each processing node is allowed 2 outstanding reads, 32 outstanding writes and 16 outstanding prefetches. A maximum of 16 requests per processing node are allowed in the network. In the event that the processor attempts to issue more than 16 requests, demand requests take priority over prefetch requests. Requests for cache lines that are outstanding are not duplicated.

We use the Parafrase compiler to generate input for the simulator. PARAFRASE restructures sequential programs and parallelizes them. Prefetches are generated, based on the framework developed in Section 3. PARAFRASE then generates pseudo-assembly level output. Some basic peephole optimizations are then applied to this code.

Benchmark	Description
CG	conjugate gradient solver
EFLUX	subroutine from the Perfect Club benchmark FLO52
INTERP	subroutine from the NAS Parallel benchmark MG

Table 1: Benchmarks

Config. No.	No. of Procs.	Cache Line Size	Switch Size	Min. Read Latency
1	64	32	4×4	60
2	64	32	2×2	108
3	64	16	4×4	58
4	64	16	2×2	106

Table 2: Simulated system configurations

5 Performance Analysis and Comparison of Prefetching Schemes

We tested the effectiveness of our prefetching method on the three benchmarks listed in Table 1. We ran each benchmark using three different prefetching techniques:

NP—no prefetching used as a baseline result

HP—hardware prefetching We use the tagged prefetching method [1].

IP—integrated prefetching We use our scheme presented in Section 3.

For each prefetching technique, we used the four different system configurations listed in Table 2. Each memory module and network switch has a delay of 8 cycles.

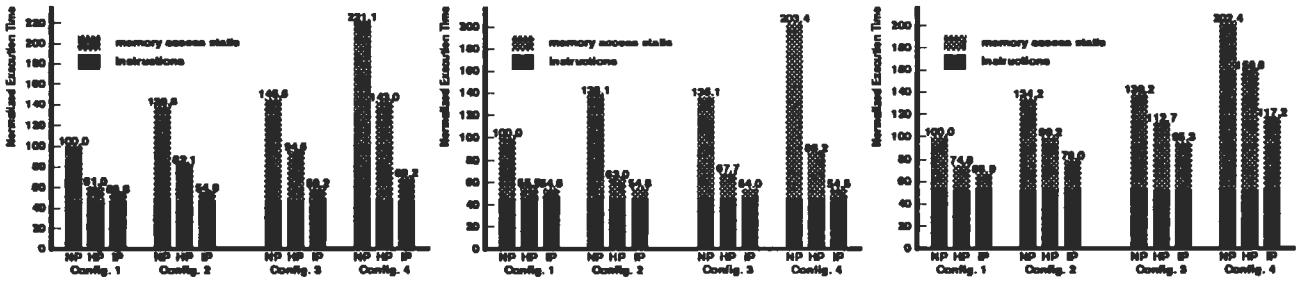
As can be seen from Figure 1, hardware prefetching is effective at removing some of the memory access stalls, but it is not as powerful as the integrated scheme. In addition, the integrated scheme performs well for all four configurations. We chose these configurations because they are practical and feasible, and because the parameters that are varied directly affect the prefetching degree. By varying the prefetching degree for each access stream, the integrated scheme is able to conform to the different architectural configurations.

6 Prefetching Adaptivity

As can be seen in Figure 1, there is often a significant portion of the memory stall time that the integrated scheme does not eliminate. A possible reason is that a prefetching degree based on the minimum latency might not be sufficient.

This suggests two possible improvements to the integrated scheme that we are currently investigating:

1. Perhaps the compiler should use a larger *effective* latency, for calculating the prefetching degree.
2. Adapting the prefetching degree at runtime, on a program by program and stream by stream basis, might be beneficial.



CG EFLUX INTERP

Figure 1: Performance Comparison Between Different Prefetching Strategies for Configurations Listed in Table 2 (**NP** = No Prefetching, **HP** = Hardware Prefetching only, **IP** = Integrated Prefetching)

7 Conclusion

Data prefetching has been shown to be a very promising technique for tolerating the large memory latencies common in shared-memory multiprocessors. Both hardware and software data prefetching schemes have been proposed and evaluated; however, both types of prefetching have their shortcomings. We propose an integrated hardware/software prefetching scheme that incorporates the best aspects of both forms of prefetching. We demonstrate the effectiveness of our integrated scheme through the detailed simulation of several benchmarks.

Our integrated scheme is more efficient than a pure software approach, since our scheme requires fewer prefetching instructions. Mowry, Lam and Gupta [7] found that software prefetching can increase the number of instructions by up to 50%, whereas our scheme has relatively little instruction overhead. We are currently performing quantitative comparisons between our integrated scheme and software prefetching.

Hardware prefetching schemes need to get progressively more complex in order to handle large strides and a prefetching lookahead greater than one block. These two facets are handled by the software support in our integrated scheme; therefore, our hardware support is simpler than that of other hardware prefetching schemes. However, in our scheme, most data accesses can still be handled in hardware.

References

[1] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[2] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *International Symposium on Computer Architecture*, pp. 364–373, 1990.

[3] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing*, pp. 176–186, 1991.

[4] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *International Symposium on Microarchitecture*, pp. 102–110, Dec. 1992.

[5] Y. Jegou and O. Temam, "Speculative prefetching," in *International Conference on Supercomputing*, 1993.

[6] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *International Conference on Parallel Processing*, 1993.

[7] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, Oct. 1992.

[8] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, Apr. 1991.

[9] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *International Symposium on Microarchitecture*, pp. 69–73, Nov. 1991.

[10] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefething," in *International Symposium on Computer Architecture*, pp. 43–53, 1991.

[11] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *International Conference on Supercomputing*, June 1990.

[12] T.-F. Chen, *Data Prefetching for High-Performance Processors*. PhD thesis, Dept. of Computer Science and Engineering, University of Washington, 1993.

[13] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Fourth International Computer Software and Applications Conference*, Oct. 1980.

0-8493-2493-9/0-6 $20.00 © 1994 IEEE