DATA PREFETCHING AND DATA FORWARDING IN SHARED MEMORY MULTIPROCESSORS (*)

David K. Poulsen and Pen-Chung Yew
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{poulsen,yew}@csrd.uiuc.edu

This paper studies and compares the use of data prefetching and an alternative mechanism, data forwarding, *for reducing memory latency due to interprocessor communication in cache coherent, shared memory multiprocessors. Two multiprocessor prefetching algorithms are presented and compared. A simple blocked vector prefetching algorithm, considerably less complex than existing software pipelined prefetching algorithms, is shown to be effective in reducing memory latency and increasing performance. A* Forwarding Write *operation is used to evaluate the effectiveness of forwarding. The use of data forwarding results in significant performance improvements over data prefetching for codes exhibiting less spatial locality. Algorithms for data prefetching and data forwarding are implemented in a parallelizing compiler. Evaluation of the proposed schemes and algorithms is accomplished via execution-driven simulation of large, optimized, parallel numerical application codes with loop-level and vector parallelism. More data, discussion, and experiment details can be found in [1].*

INTRODUCTION

Memory latency has an important effect on shared memory multiprocessor performance. Communication between processors distinguishes the multiprocessor memory latency problem from the uniprocessor case. Uniprocessor caches are used to hide memory latency by exploiting spatial and temporal locality, lowering cache miss ratios. Data prefetching [2, 3] has been used successfully to enhance uniprocessor cache performance. In cache coherent multiprocessors, *sharing* misses occur in addition to uniprocessor *nonsharing* misses. Sharing accesses cause communication between processors when cache misses cause shared blocks to be brought to requesting caches from other caches. Data prefetching has the potential to hide memory latency for both sharing and nonsharing accesses; however, *data forwarding* [5] may be a more effective technique than prefetching for reducing the latency of sharing accesses.

Many different prefetching architectures and algorithms have been described in the literature. This paper focuses on *software-initiated non-binding* prefetching into cache [2, 3]. In these schemes, explicit prefetching instructions are inserted into application codes; prefetched cache blocks are still exposed to the cache replacement policy and coherence protocol. Although data prefetching has been shown to be effective in reducing memory latency in shared memory multiprocessors [6, 7], few multiprocessor studies have considered the implementation of compiler algorithms for multiprocessor prefetching [8] and the performance impact of these algorithms on large, numerical applications with loop-level and vector parallelism.

Data forwarding is a technique for integrating fine-grained message passing capabilities into a shared memory architecture. Forwarding has the capability to eliminate cache misses caused by communication between processors and to hide more latency than prefetching for accesses to shared data. Forwarding can potentially hide all the memory latency of sharing accesses since data can be sent to destination processors' caches immediately after being produced; data prefetching can only approach this level of latency tolerance.

While many data forwarding mechanisms have been proposed, many of these mechanisms have been intended primarily for use in optimizing synchronization operations. This paper considers the use of data forwarding for optimizing sharing accesses. The forwarding mechanisms considered are software-initiated non-binding mechanisms that perform *sender-initiated* forwarding of data to other processors' caches. One example of this type of mechanism is the Dash Deliver instruction [9], an operation that transmits a copy of a cache block to explicitly specified cluster caches.

PREFETCHING AND FORWARDING SCHEMES

Data prefetching is used to reduce the memory latency of both sharing and nonsharing accesses, while data forwarding is used to reduce latency exclusively for sharing accesses. Two different multiprocessor prefetching algorithms are presented that target vector statements and serial, parallel, and vectorizable loops. A data forwarding scheme is presented that focuses on sharing accesses between parallel loops, rather than fine-grained communication within a particular loop nest.

Data Prefetching

Prefetching is performed using either a blocked vector algorithm that specifically targets vector statements and vectorizable loops, or a software pipelined algorithm that extends techniques described in [3] to support serial, parallel, and vectorizable loops. The blocked vector algorithm offers the advantages of simplicity of implementation and low instruction overhead, whereas the software pipelined algorithm allows longer memory latencies to be hidden more effectively, especially for non-vectorizable loops. The experiments described in this paper use blocked vector prefetching only for explicit vector statements, so software pipelined prefetching increases the scope of prefetching to include references in other types of loops.

The blocked vector prefetching algorithm stripmines work into blocks of N elements and issues prefetches for each block immediately before the computation for that block. It works by exploiting spatial locality in vectorizable references, hiding memory latency particularly for the latter accesses in each block. An example of this type of prefetching transformation for a simple vector statement is given in Figure 1.

The multiprocessor software pipelined prefetching algorithm supports prefetching for parallel loops, vector operations, vectorizable loops, and serial loops. Round-robin scheduling and processor self-scheduling of parallel loop iterations are both

(*) This work was supported in part by the National Science Foundation under grant nos. NSF MIP 93-07910 and NSF MIP 89-20891, the Department of Energy under grant no. DOE DE FG02-85ER25001, the National Security Agency, and an IBM Resident Study Fellowship.

supported. The algorithm uses loop splitting and software pipelining transformations, estimates loop execution times, computes prefetch distances, and schedules prefetching in a manner similar to that of [3]. Loop prolog and epilog transformations are subject to thresholds in order to prevent excessive code expansion.

```
a(i:j)=b(i:j)+c(m:n)

do k = 0,j - i,N
   itmp = min(k + N - 1,j - i)
   prefetch (b(<k:itmp>+i))
   prefetch (c(<k:itmp>+m))
   a(<k:itmp>+i)=b(<k:itmp>+i)+c(<k:itmp>+m)
end do
```

Figure 1 - Vector Prefetching Strategy

Data Forwarding

Data forwarding is performed using *Forwarding Write* operations. A Forwarding Write is a single instruction that combines a write and a sender-initiated forwarding operation, where the semantics of the forwarding operation are similar to those of the Dash Deliver instruction [9]. Destination processors are specified via a bit vector; data are forwarded to the caches of each specified processor. Using Forwarding Write operations to combine writes and forwarding operations allows instruction overhead to be reduced, providing forwarding with a potential performance advantage over prefetching.

Compiler analysis for data forwarding must perform two tasks: marking of eligible write accesses, and determination of destination processors in subsequent loops. The experiments described in this paper use a profiling approach to perform this analysis. A profiling run identifies eligible writes (those that cause interprocessor communication) and records the destination processors for each such write. Simulation runs use this forwarding information to generate Forwarding Write instructions. Profiling produces a bit vector V(I) for each write reference to A(I) that specifies destination processors if that write is eligible for forwarding. Unique forwarding information is produced for each distinct source statement that writes each array; each execution of a particular statement uses the same forwarding information.

EXPERIMENTAL FRAMEWORK

Experimental results are acquired through execution-driven simulations of parallel application codes using EPG-sim [10]. Simulations are driven by parallel versions of Perfect Benchmarks (R) codes [11]. These parallel Cedar Fortran [12] codes express parallelism using parallel loops and vector statements. Prefetching algorithms and instrumentation for profile-based forwarding are implemented in the Parafrase-2 parallelizing compiler [13].

The modeled system is a 32 processor, cache coherent, distributed shared memory architecture with processor / cache / memory nodes connected via multistage networks. Uniform memory access (UMA) to main memory is assumed to simplify simulations. Cache hits incur a one cycle latency and misses incur a base 100 cycle latency barring contention. Processors are single-issue RISC processors that can retire one result per cycle. Forwarding Write instructions that specify multiple destination processors issue multiple point-to-point forwarding messages. Parallel loops employ round-robin processor scheduling. Barrier synchronization between parallel loops utilizes separate synchronization hardware and does not cause global spin-waiting or contention with normal network accesses. Each processor has a 64K word, lockup-free, 4-way set associative, write-back cache with single word cache blocks. A large cache size is used to highlight communication effects while de-emphasizing cache conflict behavior. Single word cache blocks are used to minimize false sharing effects caused by prefetching [4]. Cache coherence is implemented using a three state, directory-based invalidation protocol. Simulations model network contention and queuing delay as well as traffic due to coherence transactions.

The particular application codes studied are listed in Figure 2 along with the number of events (equivalent to trace size) simulated. The particular versions of the codes used in these experiments are optimized for Cedar, a 32 processor machine with an architecture similar (in terms of constructs used to describe and exploit parallelism) to the architecture described above. The codes were parallelized using an optimizing compiler and then hand-optimized to exploit available parallelism, increase locality, and reduce memory latency [12]. The codes were further modified to decrease simulation time while preserving parallelism and memory reference behavior characteristics.

code	description	events
TRFD	2-electron integrals	15 M
QCD	Lattice gauge theory	23 M
FLO52	2-D transonic airflow	29 M
ARC2D	2-D fluid dynamics	76 M
DYFESM	Structural dynamics	102 M

Figure 2 - Benchmark Descriptions and Statistics

EXPERIMENTAL RESULTS

The execution time results for applying prefetching and forwarding to the various Perfect codes are given in Figure 3. BASE results assume no prefetching or forwarding. PF-V, PF-S, and FWD use blocked vector prefetching (N=32), software pipelined prefetching, and forwarding, respectively. OPT provides a lower bound on execution time assuming all memory references hit in cache and all global memory latency is hidden. Execution times in Figure 3 are normalized so the BASE execution time of each code is 1.0. Execution time is broken down into contributions from various sources:

1. Processor busy - execution of BASE instructions.
2. Instruction overhead - execution of additional instructions for prefetching or forwarding.
3. Under-utilization - time that some processors are idle, while other processors are busy, between parallel loops. For example, in a serial code section, one processor executes serial code while other processors wait for the start of the next parallel loop. This under-utilization time results from the SPMD execution model of Cedar Fortran.
4. Synchronization delay - caused primarily by barrier synchronization between successive parallel loops. This delay decreases as reductions in cache miss ratios decrease skew in loop iteration execution times.
5. Memory access delay - caused by cache misses.

Figure 4 shows the distribution of destination processors for FWD and each code. The figure gives the percentage of write operations that were forwarded to zero, one, or several processors. The CLUS case logically divides the processors into four clusters of eight processors each; the destination processor set consists of one or more clusters. The MC/BC case covers the remaining possibilities: arbitrary multicast or broadcast. Most codes forwarded substantial portions of their writes to other processors. For the most part, forwarding could be accomplished by sending small numbers of point-to-point messages per Forwarding Write.

Figure 5 depicts the miss ratio seen by processors for each code and scheme. Misses are categorized into nonsharing misses, invalidation (sharing) misses, misses for references that

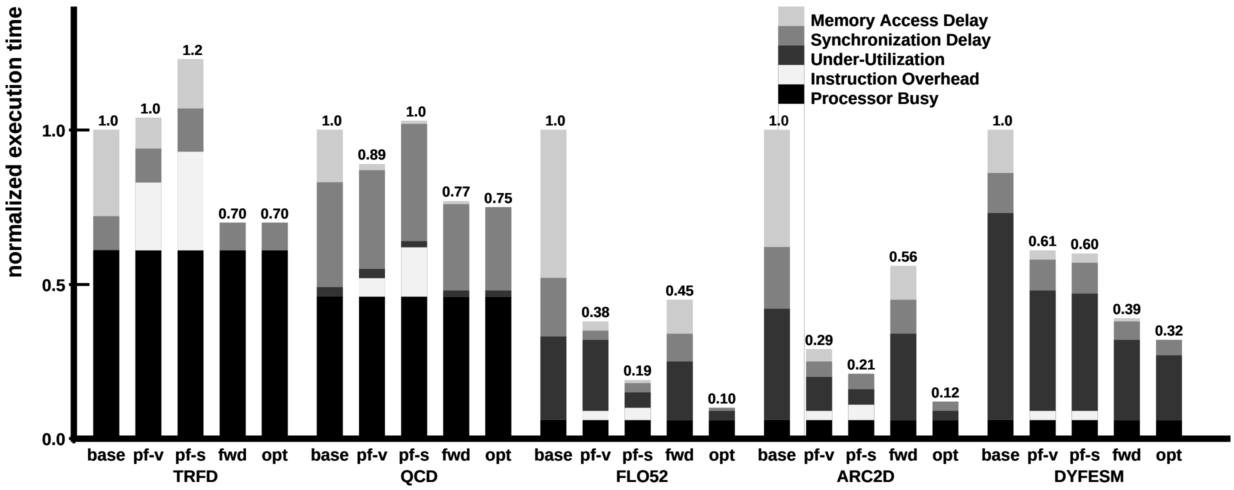

Figure 3 - Performance Benefits of Prefetching and Forwarding

were or were not prefetched, and *prefetch in progress* misses [4]. The latter represents misses where prefetched data had not yet arrived at a processor from main memory. Prefetch in progress misses incur less latency than other misses depending on the distance between the prefetch and the miss [4].

code	0	1	<=4	clus.	mc/bc
TRFD	66.4%	1.1%	32.5%	0.0%	0.0%
QCD	96.4%	0.0%	1.8%	1.1%	0.7%
FLO52	45.4%	8.9%	35.7%	8.0%	2.0%
ARC2D	67.9%	10.2%	21.2%	0.5%	0.2%
DYFESM	16.0%	37.3%	13.5%	32.6%	0.6%

Figure 4 - Forwarding Statistics

The following sections discuss the results for each code in more detail.

TRFD

Forwarding performed considerably better than prefetching for TRFD. Prefetching was not effective since reduced memory access delay was offset by large amounts of instruction overhead, even for PF-V, the most economical prefetching scheme. FWD hid considerably more memory access delay than prefetching and did so with significantly lower instruction overhead, achieving near-optimal performance.

The majority of cache misses in TRFD were nonsharing misses. Most of the nonsharing misses were cold start misses resulting from the generation of initial values for the problem (write misses) and the communication of these values to other processors for the initial computation (read misses). Many nonsharing accesses were prefetched under PF-V and PF-S, but many nonsharing prefetched misses still occurred. Even PF-S could not prefetch references that caused cold start write misses. Prefetching was successful, as was FWD, in eliminating invalidation misses. FWD had a higher nonsharing miss ratio than either prefetching scheme, but its overall miss ratio was lower as prefetching caused many prefetch in progress misses. This large percentage of prefetch in progress misses indicates why prefetching was not effective: miss latency could not effectively be eliminated by prefetching due to the small amount of overlap between communication and computation in TRFD.

QCD

Forwarding also performed better than prefetching for QCD, achieving near-optimal performance. Both prefetching and forwarding were effective in removing memory access delay. PF-S increased instruction overhead compared to PF-V, but did not hide any additional memory latency. FWD was slightly more successful in removing memory access delay than prefetching while introducing very little instruction overhead. The two prefetching schemes were effective in attacking both invalidation and nonsharing misses but caused some ineffective prefetching (prefetch in progress misses). PF-S reduced miss ratios slightly compared to PF-V. The nonsharing, nonprefetched misses remaining under PF-S were cold start write misses (see TRFD, above). FWD reduced nonsharing misses compared to prefetching and eliminated most invalidation misses.

FLO52 and ARC2D

The results for FLO52 and ARC2D were similar in many respects, as these two codes have similar characteristics. Prefetching was highly effective in hiding memory access delay, outperforming forwarding for both codes. PF-S was somewhat more effective than PF-V and achieved good performance compared to OPT. A major contributor to increased performance under PF-S, compared to PF-V, was decreased under-utilization time. FLO52 and ARC2D each spend approximately 25% of their execution time in serial code. PF-S increased the scope of prefetching to include more serial code, resulting in greater reductions in memory access delay. This, in turn, reduced the execution time of serial code sections and decreased under-utilization time. FWD was not as successful as prefetching in reducing memory access delay or under-utilization time.

The performance of both codes was essentially dominated by cache behavior. The BASE codes had high miss ratios due to frequent, large vector accesses, significant spatial locality, and the small cache block size simulated. Both prefetching schemes addressed these miss ratios successfully, but in different ways. Prefetching was successful in eliminating many of the invalidation misses that occurred in the BASE codes. PF-S prefetched more nonsharing misses than PF-V, but did not accordingly reduce nonsharing miss ratios. Forwarding was not nearly as successful at reducing miss ratios since it did not exploit the inherent spatial locality in the codes as effectively as did prefetching. Both FLO52 and ARC2D used substantial numbers of Forwarding Writes (Figure 4). FWD caused increased numbers of conflict and invalidation misses compared to prefetching, indicating that forwarding was somewhat aggressive and replaced needed data in destination processor caches.

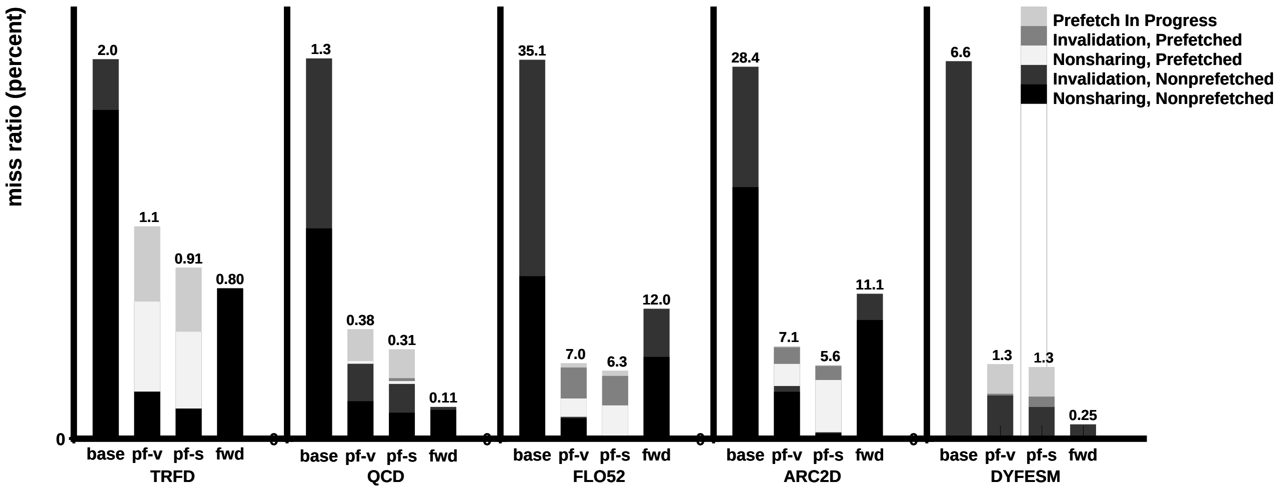

Figure 5 - Processor Cache Miss Behavior. The data presented for each code use a different y scale.

DYFESM

Forwarding outperformed prefetching for DYFESM and achieved good performance compared to OPT. Prefetching was effective in removing memory access delay for DYFESM; PF-V was only slightly less effective than PF-S. FWD eliminated more memory access delay than prefetching with little instruction overhead. Despite the success of prefetching in removing memory access delay, overall performance did not approach OPT due to the instruction overhead of prefetching and high under-utilization time. Parallelism in DYFESM consists predominantly of vector operations and single-cluster inner parallel loops, resulting in low processor utilization. This explains the high frequency of CLUS forwarding in DYFESM (Figure 4). The major contributor to miss ratios in DYFESM was invalidation misses. FWD was more effective in attacking invalidation misses and resulted in a lower overall miss ratio. Miss ratios under prefetching were aggravated by prefetch in progress misses.

Bandwidth Requirements

The availability of sufficient bandwidth is important to the performance of prefetching [4] and forwarding. Figure 6 gives the memory and network statistics for the experiments described above. For each code and scheme, the increase in memory requests (MSGS), the average main memory access latency (L AVG), and the average network utilization (U AVG) are given. While the modeled system has large amounts of memory bandwidth available, much of this bandwidth was not required to achieve the results shown in Figure 3. Network utilization was low in all cases and cache miss latencies were generally close to the base latency of 100 cycles. The exceptions were FWD for TRFD, which forwarded over 30% of writes to multiple processors, and prefetching for DYFESM, which had the largest increase in memory requests and a large percentage of prefetch in progress misses.

SUMMARY AND DISCUSSION

Prefetching was effective in terms of overall performance for all codes except TRFD. Software pipelined prefetching outperformed blocked vector prefetching in FLO52 and ARC2D, the two most heavily parallel, vectorizable codes with higher spatial locality. For these two codes, PF-S was more effective than PF-V, despite its higher instruction overhead, since it increased the scope of prefetching to include parallel loops and nonvectorizable serial loops. For the remaining three codes, the potential benefits of software pipelined prefetching were outweighed by the instruction overhead caused by loop splitting and software pipelining; this was particularly true in TRFD. Blocked vector prefetching offered reasonable performance at a lower cost in instruction overhead than software pipelined prefetching for all five codes.

code / vers		msgs	l avg	u avg
TRFD	base	+0%	100.04	1.6%
	pf-v	+46.5%	104.34	4.4%
	pf-s	+73.8%	105.20	4.4%
	fwd	+0%	169.31	14.2%
QCD	base	+0%	100.19	0.4%
	pf-v	+13.6%	100.97	0.9%
	pf-s	+37.0%	101.61	0.9%
	fwd	+0%	100.08	5.1%
FLO52	base	+0%	100.09	1.7%
	pf-v	+69.9%	101.42	11.0%
	pf-s	+76.0%	100.85	8.6%
	fwd	+0%	100.44	10.6%
ARC2D	base	+0%	100.06	1.8%
	pf-v	+70.4%	100.61	10.8%
	pf-s	+79.9%	100.50	8.7%
	fwd	+0%	100.28	7.1%
DYFESM	base	+0%	111.88	0.4%
	pf-v	+93.0%	123.91	1.3%
	pf-s	+96.3%	123.71	1.3%
	fwd	+0%	103.18	0.7%

Figure 6 - Network and Memory Traffic Statistics

Forwarding was highly effective for TRFD, QCD, and DYFESM, resulting in near-optimal performance, and eliminating misses and reducing memory latency more effectively than prefetching. Forwarding was also more effective for these codes because it eliminated the instruction overhead of prefetching through the use of Forwarding Write operations. Reduced overlap between communication and computation in TRFD allowed forwarding to outperform prefetching since data were forwarded to destination processors immediately after being produced.

The modeled system has a large cache size in order to emphasize communication between processors, yet some codes still exhibited nonsharing misses. TRFD and QCD had low overall miss ratios and the nonsharing misses that occurred in these codes were primarily cold start misses. Many of these cold start misses were actually related to communication between processors (see TRFD, above). FLO52 and ARC2D had significant numbers of BASE conflict misses, but conflict miss ratios were significantly reduced when exploiting spatial locality via prefetching (or by increasing the cache block size, not shown).

Both prefetching algorithms were successful in reducing miss ratios. Reductions in miss ratios were offset by increases in prefetch in progress misses in TRFD, QCD, and DYFESM. Prefetching was effective in attacking both conflict (FLO52, ARC2D) and invalidation misses. Prefetching was found to be less effective in attacking invalidation misses in [4]; however, false sharing was shown to be a contributor to this effect. A small cache block size was used in the experiments described above to eliminate false sharing and increase prefetching effectiveness for invalidation misses.

Forwarding dramatically reduced miss ratios for QCD and DYFESM; some improvement was also seen for TRFD. Forwarding was more successful in eliminating cold start and invalidation misses than conflict misses. This is expected since writes were marked eligible for forwarding only when they caused interprocessor communication. In FLO52 and ARC2D, aggressive forwarding caused increased numbers of conflict misses as needed data were replaced in destination processor caches.

Profile-based forwarding provides an optimistic estimate of the performance benefits of forwarding based on static compiler analysis. Profiling is realistic compared to static analysis, however, in that a single forwarding pattern is developed for each source statement containing an eligible write and is used for each execution of each such statement. The bit vectors generated during profiling allow non-uniform and multicast forwarding patterns to accurately be represented, but bit vectors can incur high storage overhead. The data in Figure 4 indicate that most forwarding operations require small numbers of destination processors. Furthermore, a preliminary examination of the forwarding patterns produced during profiling indicates that most forwarding patterns are regular; this is to be expected given the nature of the codes and that round-robin loop scheduling was employed. These facts suggest that static analysis can focus on unicast and point-to-point forwarding and achieve good performance without requiring the generation of bit vectors.

CONCLUSIONS

The performance of data prefetching and data forwarding have been studied and compared in the context of reducing memory latency for sharing accesses in cache coherent multiprocessors executing large, optimized, numerical application codes. Two multiprocessor prefetching algorithms were presented, implemented, and compared. The Forwarding Write operation was introduced and used to evaluate the performance of data forwarding using a profiling approach. Forwarding focused on reducing memory latency for sharing accesses caused by interprocessor communication between, rather than within, parallel loops.

Forwarding provided significant improvements in memory latency for sharing accesses in the codes studied. Prefetching and forwarding were each effective for application codes with different characteristics. Forwarding achieved near-optimal performance for codes with more communication-related misses and less spatial locality. Forwarding also performed well in cases where codes exhibited less overlap between communication and computation since data were forwarded to destination processors immediately after being produced. Prefetching outperformed forwarding for codes with more spatial locality and more conflict misses. Blocked vector prefetching performed well, achieving reasonable performance for the codes studied, producing lower instruction overhead, and allowing a simpler compiler implementation than software pipelined prefetching.

More work is required to develop multiprocessor prefetching and forwarding strategies that handle varying application characteristics in a more robust manner. Given that prefetching and forwarding are each effective for different types of accesses, a hybrid prefetching and forwarding scheme is being developed that adapts to the sharing characteristics of different references. Future studies need to consider the effect of cache size on the performance of prefetching and forwarding. Prefetching may be more effective in dealing with finite cache size effects given its effectiveness in dealing with conflict misses. Over-aggressive forwarding may cause unnecessary cache replacement, especially in smaller caches. A hybrid prefetching and forwarding scheme may be useful in countering this effect. Another important area of future work is the development of forwarding algorithms based on static compiler analysis rather than profiling. Experimental results suggest that unicast and point-to-point forwarding can achieve good performance; this should considerably simplify compiler analysis without sacrificing performance objectives.

ACKNOWLEDGMENTS

The authors would like to thank John Andrews, Ding-Kai Chen, Lynn Choi, and Josep Torrellas for their comments on drafts of this paper, Pavlos Konas for his help in developing EPG-sim, and Angeline Gliniecki for detailed, last-minute proofreading.

REFERENCES

[1] Poulsen, D. K., and Yew, P.-C., *Data Prefetching and Data Forwarding in Shared Memory Multiprocessors*, University of Illinois, CSRD Report 1330, April 1994.

[2] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching", Proceedings of ASPLOS IV, pp. 40-52.

[3] Mowry, T. C., Lam, M. S., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching", Proceedings of ASPLOS V, pp. 62-73.

[4] Tullsen, D. M., and Eggers, S. J., "Limitations of Cache Prefetching on a Bus-Based Multiprocessor", Proceedings of ISCA 1993, pp. 278-288.

[5] Andrews, J. B., Beckmann, C. J., and Poulsen, D. K., "Notification and Multicast Networks for Synchronization and Coherence", *Journal of Parallel and Distributed Computing*, 15(4), August 1992, pp. 332-350.

[6] Mowry, T. C., and Gupta, A., "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors", *Journal of Parallel and Distributed Computing*, 12(2), June 1991, pp. 87-106.

[7] Kuck, D., et. al., "The Cedar System and an Initial Performance Study", Proceedings of ISCA 1993, pp. 213-223.

[8] Mowry, T. C., *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph.D Thesis, Stanford University, Dept. of Elec. Engr., March 1994.

[9] Lenoski, D., et. al., "The Stanford Dash Multiprocessor", *IEEE Computer*, 25(3), March 1992, pp. 63-79.

[10] Poulsen, D. K., and Yew, P.-C., "Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications", Proceedings of Supercomputing 1993, pp. 860-869.

[11] Berry, M., et. al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers", *International Journal of Supercomputer Applications*, 3(3), Fall 1989, pp. 5-40.

[12] Padua, D., Hoeflinger, J., Jaxon, G., and Eigenmann, R., *The Cedar Fortran Project*, University of Illinois, CSRD Report 1262, October 1992.

[13] Polychronopoulos, C. D., et. al., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Proceedings of ICPP 1989, pp. II-39-48.