INTEGRATING FINE-GRAINED MESSAGE PASSING IN CACHE COHERENT SHARED MEMORY MULTIPROCESSORS (*)

David K. Poulsen

Kuck and Associates, Inc.

1906 Fox Drive

Champaign, IL 61821

217-356-2288

poulsen@kai.com

and

Pen-Chung Yew

Department of Computer Science

University of Minnesota

4-192 EE/CS Building

200 Union Street SE

Minneapolis, MN 55455

yew@cs.umn.edu

(*) 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.

This work was performed while the authors were with the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

PROPOSED RUNNING HEAD:

Fine-Grained Message Passing in Shared Memory MPs

KEYWORDS:

Data prefetching, data forwarding, message passing, compiler algorithms, cache coherent shared memory multiprocessors, execution-driven simulation, Perfect Benchmarks.

PRIMARY, CORRESPONDING AUTHOR'S ADDRESS:

Dr. David K. Poulsen
Kuck and Associates, Inc.
1906 Fox Drive
Champaign, IL 61821
217-356-2288
poulsen@kai.com

ABSTRACT:

This paper considers the use of data prefetching and an alternative mechanism, data forwarding, for reducing memory latency caused by interprocessor communication in cache coherent, shared memory multiprocessors. Data prefetching is accomplished by using a multiprocessor software pipelined algorithm. Data forwarding is used to target interprocessor data communication, rather than synchronization, and is applied to communication-related accesses between successive parallel loops. Prefetching and forwarding are each shown to be more effective for certain types of architectural and application characteristics. Given this result, a new hybrid prefetching and forwarding approach is proposed and evaluated that allows the relative amounts of prefetching and forwarding used to be adapted to these characteristics. When compared to prefetching or forwarding alone, the new hybrid scheme is shown to increase performance stability over varying application characteristics, to reduce processor instruction overheads, cache miss ratios, and memory system bandwidth requirements, and to reduce performance sensitivity to architectural parameters such as cache size. Algorithms for data prefetching, data forwarding, and hybrid prefetching and forwarding are described. These algorithms are implemented within a parallelizing compiler and are evaluated via execution-driven simulations of large, optimized, numerical application codes with loop-level and vector parallelism.

LIST OF SYMBOLS:

Few symbols appear in the text. Those that do appear in a `constant-width font`. Examples of the most common usages appear below:

`A(i)`	array A, element i
`<`	less than
`<=`	less than or equal to
`(p: 0 <= p < P)`	`p` such that `0 <= p < P`

1. INTRODUCTION

The reduction of communication-related global memory latency is critical to the performance of shared memory multiprocessors. As these architectures scale to larger numbers of processors, memory latencies become larger and memory system performance becomes more sensitive to communication-related latency. From a technological standpoint, processor performance is advancing more rapidly than memory speed, which exacerbates the memory latency problem.

Communication between processors distinguishes the multiprocessor memory latency problem from the uniprocessor problem. In uniprocessors, data prefetching [1, 2] has been used successfully to hide memory latency and increase performance by exploiting spatial and temporal locality. In cache coherent multiprocessors, however, *sharing* misses caused by communication between processors occur in addition to uniprocessor *nonsharing* misses. Multiprocessor caches can hide memory latency for sharing accesses by exploiting spatial locality, but increasing cache block sizes may lead to undesirable false sharing [3]. Data prefetching has been shown to be effective in reducing memory latency in shared memory multiprocessors [4, 5]; however, while data prefetching has the ability to hide memory latency for both sharing and nonsharing accesses, *data forwarding* [6, 7] may be a more effective technique than data prefetching for reducing the latency of sharing accesses.

This paper studies and compares the performance advantages of data prefetching and data forwarding for reducing memory latency caused by interprocessor communication in cache coherent, shared memory multiprocessors. Data prefetching is accomplished by using a multiprocessor software pipelined algorithm based on [2]. Data forwarding integrates fine-grained message passing into a shared memory architecture by allowing individual cache blocks to be sent explicitly to other processors' caches within the context of a cache coherence protocol [6]. The experiments described in this paper utilize data forwarding to optimize interprocessor data communication, rather than synchronization, and data forwarding is applied to application codes to target communication between, rather than within, parallel loops. Experimental results are presented to show the performance advantages of data prefetching and data forwarding. While prefetching and forwarding both offer increased performance, each approach is effective for different types of access patterns and application characteristics.

Given that neither prefetching nor forwarding is consistently more effective over a range of applications, this paper proposes a new hybrid approach. A hybrid prefetching and forwarding scheme is described and evaluated that allows the relative amounts of prefetching and forwarding used to be adapted to the sharing and locality characteristics of particular applications, as well as to architectural characteristics such as cache size [8]. The use of this hybrid scheme is shown to increase performance stability, when compared to either prefetching or forwarding alone, over varying application characteristics and varying cache sizes. The hybrid scheme reduces processor instruction overhead, cache miss ratios, and memory system bandwidth requirements when compared to prefetching alone. In addition, hybrid prefetching and forwarding significantly reduces performance sensitivity to cache size, when compared to forwarding alone, by reducing undesirable cache replacement associated with forwarding in smaller caches.

Algorithms for the data prefetching, data forwarding, and hybrid prefetching and forwarding schemes are described. Experimental results are acquired through execution-driven simulations of large, optimized, parallel numerical benchmarks by using EPG-sim [9], a system of execution-driven tools for studying parallel architectures, compiler algorithms, and applications. Simulations model a shared memory architecture with directory-based cache coherence and are driven by parallel versions of Perfect Benchmarks [10] codes that have been optimized for the modeled architecture. These parallel Cedar Fortran [11] codes have prefetching and forwarding applied via algorithms implemented with the EPG source code instrumentation tools [12], a set of passes implemented within the Parafrase-2 parallelizing compiler [13].

The remainder of this paper is organized as follows. Section 2 presents a brief overview of existing data prefetching and data forwarding schemes and provides additional motivation for the schemes presented and evaluated in the following sections. Section 3 presents the multiprocessor software pipelined data prefetching algorithm, the data forwarding scheme, and the hybrid prefetching and forwarding scheme. The cost and complexity of hardware support for these schemes is also briefly discussed. Section 4 describes the experimental methodology used to evaluate the various schemes, including information on the modeled system architecture and application codes used. Section 5 presents the experimental results, including execution time, cache behavior, and memory system load information. These results are summarized and discussed in Section 6. Section 7 describes a compiler algorithm for data forwarding and the hybrid scheme, and discusses its potential effectiveness. Finally, Section 8 presents conclusions.

2. BACKGROUND

This section presents a brief survey of relevant work in data prefetching and data forwarding. The schemes presented in this paper are categorized in relation to this work, and some pertinent trade-offs are identified and discussed. The data prefetching scheme considered in this paper is related to several existing techniques; however, little previous work exists in the areas of data forwarding for optimizing interprocessor communication, compiler algorithms for this type of data forwarding, or hybrid prefetching and forwarding schemes.

Data prefetching is a popular latency reduction technique used in many uniprocessor and multiprocessor architectures. Many different prefetching architectures and algorithms have been described and evaluated in the literature. This paper considers *software-initiated, non-binding* prefetching [1, 2] into processor caches. Software-initiated schemes, as opposed to hardware-initiated schemes that do not require software intervention, accomplish prefetching by inserting explicit instructions into application codes. Such instructions may be inserted at the source or machine-language level; the algorithms described in this paper employ source-level techniques. One potential disadvantage of software-initiated schemes is the overhead introduced by prefetching and related instructions; this overhead can have a significant impact on performance [1, 2]. Non-binding prefetching refers to the idea that prefetched cache blocks are still exposed to the cache replacement policy and coherence protocol, and may be cast out or invalidated after being prefetched but before being used. This approach considerably reduces algorithm complexity, ensuring correctness and preserving cache coherence even if prefetched data cannot be guaranteed by the prefetching algorithm to be used.

Data forwarding may be a more effective mechanism than data prefetching for reducing the latency of sharing accesses. Data forwarding, as considered here, is a fine-grained message passing capability that is integrated into a shared memory architecture and is used to optimize interprocessor data communication [6, 8]. This capability is provided by a forwarding operation that allows data to be sent explicitly from one processor to another within the context of a cache coherence protocol. The forwarding capability is used to reduce the latency of communication-related accesses between nests of parallel loops. This type of data forwarding mechanism has the ability to eliminate sharing misses and to hide memory latency more efficiently than prefetching for accesses to shared data. 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.

Although many mechanisms have been proposed that could be used to implement data forwarding [14, 15, 16, 17, 18, 19], most of these mechanisms have been introduced or evaluated for other purposes, such as reducing the latency of synchronization operations. Little performance information exists regarding the usefulness of these mechanisms for optimizing interprocessor data communication. Accordingly, the design and evaluation of compiler algorithms for forwarding, as well as hybrid prefetching and forwarding, has not yet been addressed. As in the case of data prefetching, this paper considers software-initiated, non-binding data forwarding into processor caches and employs source-level insertion of forwarding instructions. The forwarding mechanism used is based on the Dash Deliver instruction [14], although the Deliver operation was not originally intended for this purpose. This data forwarding mechanism and its application are described in greater detail in the following section. The data forwarding scheme considered in this paper assumes static, round-robin processor scheduling (cyclic prescheduling) of parallel loop iterations. This assumption facilitates the construction of a forwarding compiler algorithm and of a hybrid scheme that can integrate data forwarding with data prefetching. Other mechanisms have been proposed that could be used for data forwarding in a dynamic, self-scheduling environment [7, 14, 16, 18].

3. PREFETCHING AND FORWARDING SCHEMES

This section describes the data prefetching algorithm, the data forwarding scheme, and the new hybrid prefetching and forwarding scheme. These schemes are intended for use in shared memory multiprocessors executing parallel codes such as Cedar Fortran [11] codes. The pertinent language features of such codes, for purposes of this discussion, are parallel (DOALL) loops, vectorizable loops containing regular, strided array section references, serial loops, and vector statements containing array section references. 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. The prefetching algorithm is a multiprocessor software pipelined prefetching algorithm that targets vector statements and serial, parallel, and vectorizable loops. The *Forwarding Write* operation is introduced for use in data forwarding.

```
DO k = i, j                          DO k = i, i + D - 1
     A(k) = B(k) + C(k)    -->            prefetch (B(k))
END DO                                    prefetch (C(k))
                                     END DO
                                     DO k = i, j - D
                                          prefetch (B(k + D))
                                          prefetch (C(k + D))
                                          A(k) = B(k) + C(k)
                                     END DO
                                     DO k = j - D + 1, j
                                          A(k) = B(k) + C(k)
                                     END DO
```

Figure 1 - Software Pipelined Prefetching Strategy

3.1 Prefetching Algorithm

Data prefetching is performed by using a software pipelined algorithm that extends techniques described in [2] to support prefetching for parallel loops, vector operations, and vectorizable loops in addition to serial loops [6, 8]. 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 [2]. *Selective prefetching* [2] is not currently implemented, significantly reducing the complexity (but as will be shown in Section 5, not the effectiveness) of the algorithm. An example of software pipelined prefetching for a simple serial loop, when a prefetch distance of `D` iterations is assumed, is given in Figure 1. The prefetch distance, `D`, is chosen individually for each loop based on a specified memory latency as well as compiler-generated execution time estimates for loop bodies [2].

Software pipelined prefetching is supported for parallel loops when using either round-robin scheduling or self-scheduling; the experiments described in this paper employ round-robin scheduling. In this case, prefetch distances for parallel loops are multiplied by the number of processors that execute these loops to ensure that each processor will prefetch data it will ultimately use. Vector operations are treated as being equivalent to serial loops for the purpose of applying software pipelined prefetching.

3.2 Forwarding Architecture

Data forwarding is performed by 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 [14]. A Forwarding Write operation transmits copies of specified operands to a set of explicitly specified destination caches. The operand size assumed is a single cache block, and destination processors are specified via a bit vector. A Forwarding Write operation forwards a block to the caches of each specified destination processor in the read-shared state. The operation has write-allocate semantics if the processor issuing the operation specifies itself as a destination processor.

Several performance benefits can be achieved when Forwarding Write operations are used to perform data forwarding. Forwarding using Forwarding Write operations can result in reduced memory latency for sharing accesses, when compared to prefetching, because data can be sent to destination processors immediately upon being written (i.e., during the same memory and cache coherence transactions). This type of optimization can be especially useful for sequential (or *migratory* [20]) sharing patterns. Existing software-initiated prefetching schemes assume that prefetching operations are physically separated from loads in order to maximize latency hiding. Use of the Forwarding Write operation can reduce the instruction overhead associated with forwarding by combining writes and forwarding operations, giving forwarding another potential performance advantage over prefetching.

3.3 Forwarding Algorithm

The forwarding schemes presented in this paper target the reduction of memory latency for sharing accesses between successive parallel loops, rather than for fine-grained communication and synchronization within a particular parallel loop nest or DOACROSS loop [17]. A compiler algorithm for forwarding between parallel loops must perform two tasks: marking of eligible write accesses and determination of destination processors in subsequent loops; this algorithm is described in Section 7.

Figure 2 illustrates a simple example of matrix transposition, extracted from the Perfect code TRFD, that illustrates how a forwarding compiler algorithm would generate Forwarding Write operations to optimize the sharing accesses caused by the execution of this computation on a shared memory multiprocessor. Assuming round-robin scheduling, each DOALL iteration `i` executes on processor

For the loops:

```
DOALL i = 1, N
     DO j = 1, N
          VT(j,i) = V(i,j)
...
DOALL i = 1, N
     DO j = 1, N
          ... = VT(i,j)
```

the write loop is transformed to:

```
DOALL i = 1, N
     DO j = 1, N
          Forwarding_Write (VT(j,i), (j-1) mod P) = V(i,j)
```

Figure 2 - Data Forwarding Example

`p = (i-1) mod P`, where `P` is the number of processors executing the loops. For processor `p` performing an eligible write of `VT(j,i)` in the first loop nest, processor `q = (j-1) mod P` will read this element of `VT` in the second loop nest. Accordingly, the write of `VT(j,i)` in the first loop nest is replaced with a Forwarding Write that also forwards each element of `VT` to the indicated destination processor.

In light of the potential implementation complexity of a forwarding compiler algorithm (Section 7), the experiments described in Section 5 use a profiling approach for identifying eligible writes and computing destination processors for forwarding. A profiling simulation identifies eligible writes (those that cause interprocessor communication) and records the set of destination processors for each such write. Subsequent simulations use this forwarding information to generate Forwarding Write instructions. The forwarding compiler algorithm presented in Section 7 is shown to be capable of identifying most of the accesses and destination processor patterns found in the experiments, and of generating Forwarding Write instructions for these accesses.

Profiling simulations target writes of array elements and collect separate forwarding information for each unique write reference in an application code Different source references to a particular array are considered to be different unique write references. In addition, different references to subprogram formal parameter arrays corresponding to different actual parameter arrays are considered to be unique. A profiling simulation determines the destination processors for each unique write reference and stores these destination processors for each element of each reference as a bit vector, similar to the bit vectors used in the Dash Deliver instruction [14]. The end result of profiling is a data structure containing arrays of bit vectors for each eligible written array, so that for a write to `A(I)`, `V(I)` is a bit vector indicating the destination processors for the write. During a subsequent forwarding simulation, an eligible write to `A(I)` is replaced with:

```
Forwarding_Write (A(I), V(I)),
```

where `V(I)` is the bit vector of destination processors for `A(I)` from the data structure saved during a previous profiling simulation. Writes are eligible for forwarding if they have downstream reads on other processors.

3.4 Hybrid Prefetching and Forwarding

This paper introduces a new hybrid scheme that attempts to combine the best features of both data prefetching and data forwarding. The hybrid scheme is adaptable to the sharing characteristics of different application codes and to varying architectural parameters. This scheme was inspired by techniques employed in compiler-assisted adaptive cache coherence schemes [21] that use cost functions to decide whether updating or invalidating is most appropriate for particular references based on their sharing patterns. In hybrid prefetching and forwarding, an analogous cost function is used for each write/read reference pair to decide whether to forward the write or to prefetch the read. Forwarding is performed by using Forwarding Write operations, and prefetching is performed by using the prefetching algorithm described in Section 3.1.

The cost function used in the hybrid scheme is simply the distance, in execution time, between a write and its nearest subsequent read on another processor. This cost function was chosen based on the following intuitive observations. Prefetching is most appropriate for communication-related accesses with distances considerably greater than the global memory latency, since forwarding such accesses would bring data to destination processors' caches well before being used, thereby increasing the possibility of these data being cast out and replaced before being used. Forwarding is most appropriate for accesses with short distances, since forwarding is the most efficient way to pass data from one processor to another with minimal latency and coherence protocol overhead.

The hybrid scheme takes as a parameter a threshold value for this cost function. For a particular write/read reference pair, if the distance is less than the threshold value the write is forwarded and prefetching is suppressed for the read; otherwise, forwarding is suppressed for the write and prefetching is performed for the read. Increasing the threshold value increases the amount of forwarding, as opposed to prefetching, that is performed. A threshold value of zero corresponds to pure prefetching, and an infinite threshold corresponds to pure forwarding. An appropriate initial value for the threshold might be the base global memory latency, or some small multiple of this latency, in order to minimize unnecessary replacement caused by forwarding accesses with long distances and to communicate data as efficiently as possible for accesses with short distances.

Evaluation of the hybrid prefetching and forwarding scheme is facilitated by modifying the profiling techniques described in Section 3.3 so that write-to-read distances (in simulated execution time) are computed and associated with each unique write and read reference. The data prefetching and data forwarding algorithms use this distance information to select write forwarding or read prefetching for each write/read reference pair.

3.5 Architectural Considerations

The desire to provide software-initiated prefetching and forwarding capabilities in a hardware cache coherent multiprocessor influences all aspects of system design. This section considers the architectural support required for prefetching and forwarding and the concomitant cost of this support [8]. Adding processor support for prefetching and forwarding instructions is straightforward. Several machines already support these types of operations, including the Stanford Dash multiprocessor [14], the Kendall Square KSR1 [22], and the Cedar multiprocessor [5]. The prefetch and Forwarding Write operations assumed in this work most closely resemble the Dash Prefetch and Dash Deliver instructions. A prefetch instruction is similar to a regular load, except that it is non-blocking and is dropped on exceptions [23, 24]. A Forwarding Write is similar to a regular write, except that it has different cache coherence semantics (Section 3.2).

Cache design to support prefetching and forwarding requires the ability to support multiple outstanding prefetches and the ability to accept incoming prefetched and forwarded data. These capabilities are provided by lockup-free caches [25]; however, such full lockup-free capabilities are not necessarily required. Cache design to support software-initiated prefetching has been discussed in detail in [23, 24]. The cache architectures described therein can be used for forwarding as well as prefetching [8]. Provided that caches can accept incoming prefetched or forwarded data, prefetching and forwarding operations can be viewed as being similar to writes, which allows existing write buffer implementations to be used to make the operations appear non-blocking. Write buffers can be used to save the addresses of outstanding prefetch operations to eliminate the need for separate prefetch issue buffers [4].

Given a conventional cache design employing write buffers, the cost of adding support for prefetching and forwarding to processor caches should be small provided that write buffer sizes are reasonable. Previous studies indicate that modest buffer sizes (e.g., a single, 16 word write buffer) are sufficient to achieve reasonable performance under prefetching [4, 23]. Data forwarding using Forwarding Write operations offers the additional benefit of easing write buffer size requirements. Since Forwarding Write operations replace regular writes, write buffer size requirements should not increase significantly under forwarding. Furthermore, write buffer pressure due to prefetching can be decreased through the use of hybrid prefetching and forwarding. Simulation results in [8] indicate that small write buffer sizes should be sufficient for prefetching or forwarding and that the memory system bandwidth of the modeled system (Section 4.1) is more than sufficient to allow caches to accept incoming prefetched and forwarded data.

4. EXPERIMENTAL FRAMEWORK

This section describes the experiments performed to evaluate data prefetching and data forwarding. The system simulation model and simulation methodology are presented and the specific application codes used are described.

4.1 System Model

The modeled system is a 32 processor, cache coherent, distributed shared memory architecture with processor / cache / memory nodes connected via multistage, packet-switched networks (Figure 3). The memory system provides a one cycle cache hit latency; round trip network and main memory access delay result in a base 100 cycle cache miss latency under light network loads (this latency was chosen largely to facilitate comparison with the results of other prefetching studies [2, 3, 4, 24]). Uniform memory access (UMA) to main memory is assumed to simplify simulations. The multistage networks

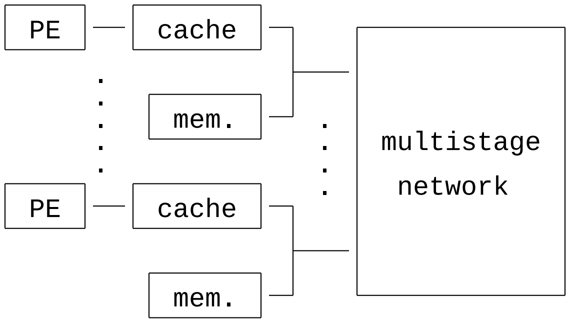


Figure 3 - Modeled System Architecture

are composed of 2x2 switches which are of the output queue variety and which perform cut-through routing [26]. Each processor is capable of in-order single instruction issue and can retire single arithmetic results in one cycle. Forwarding Write instructions that specify multiple destination processors accomplish multicast or broadcast by issuing multiple point-to-point messages. Barrier synchronization primitives and queue-based locks utilize separate synchronization hardware, so they do not contend with main memory accesses or cause global spin-waiting, and have a minimum latency of one cycle. This assumption was made so that the memory system behavior of prefetching and forwarding could be studied without being influenced by synchronization behavior.

Each processor has a four-way set-associative, write-back, write-allocate cache with write buffers and LRU replacement. Single word cache blocks are used to minimize false sharing effects, particularly those caused or aggravated by prefetching [3]. The selection of cache sizes is discussed in Section 5. Processor caches allow multiple outstanding prefetching and forwarding operations that are non-blocking to the issuing processor; regular reads block the issuing processor. Cache coherence is implemented by using a three-state, directory-based invalidation protocol. Directories are distributed across the main memory modules, are full-mapped, and are organized as pointer caches [27] to reduce their size.

4.2 Application Codes

The application codes studied in this work were selected from the Perfect Benchmarks (Table 1). The particular versions of the codes used in these experiments are parallel, Cedar Fortran versions of the applications [11] that contain parallel loops and vector statements. These codes are optimized for the Cedar multiprocessor, a 32 processor machine with an architecture similar, in terms of constructs used to describe and exploit parallelism, to the architecture described in Section 4.1. Cedar Fortran contains the

Table 1 - Parallel Benchmark Descriptions and Statistics

code	description	source lines	no. of events	% parallel execution	user CPU time	time dilat.
TRFD	molecular dynamics	265	15 M	99.9%	0:22:16	118x
QCD	quantum chromodynamics	2581	23 M	97.1%	0:29:41	217x
FLO52	transonic fluid dynamics	3987	29 M	72.6%	1:52:33	308x
ARC2D	fluid dynamics	5529	76 M	75.2%	3:48:29	214x
DYFESM	structural dynamics	8222	102 M	97.3%	6:14:17	274x

---

notion of clustering and allows two levels of parallel loops. SDOALL loops spread work across multiple clusters; CDOALL loops perform work within a single cluster and are dispatched from within iterations of SDOALL loops. These loop types were modeled by logically partitioning the available 32 processors during simulations into four clusters of eight processors each.

The codes were parallelized by using an optimizing compiler and then further hand-optimized to exploit available parallelism, increase locality, and reduce memory latency [11]. The characteristics of the resulting optimized, parallel Perfect codes vary because of the differences between the original applications and the differing methods used in their parallelization. TRFD is highly parallel but does not use particularly long vectors. FLO52 and ARC2D contain large amounts of loop-level parallelism and are highly vectorizable. QCD and DYFESM exhibit less loop-level parallelism and shorter vector lengths than TRFD, FLO52, or ARC2D. DYFESM also makes frequent use of vector dot product and summation intrinsics.

Table 1 gives several statistics for typical simulations of each code: the number of source lines, the number of events simulated (equivalent to trace size), the percentage of time spent executing parallel code (while using all 32 processors in most cases), the user CPU time required to execute a simulation (in CPU seconds on an RS/6000 model 530), and the time dilation of a simulation when compared with direct execution of the original code. These statistics give some indication of the size of the benchmarks, the level of detail of the simulations, and the efficiency of EPG-sim execution-driven simulations for these codes.

5. EXPERIMENTAL RESULTS

This section presents the experimental results of simulations that apply prefetching and forwarding to the various parallel Perfect codes. For software pipelined prefetching, the prefetching algorithm attempts to hide the base cache miss latency of 100 cycles. This base latency is also used as the threshold value for the hybrid scheme, for reasons suggested in Section 3.4. Most simulations assume a large cache size (32K words), when compared to the working set sizes of the various codes. This assumption is made in order to highlight communication effects and de-emphasize cache conflict behavior. Some results for smaller cache sizes are also presented in order to illustrate how finite cache size effects change the behavior of the simulated schemes. In these small cache size experiments, 2K word caches were used for TRFD, QCD, and DYFESM, and 8K word caches were used for FLO52 and TRFD. These sizes were determined to be appropriate to the working set sizes of each code [8].

The following specific schemes were evaluated:

Scheme	Meaning
BASE	No prefetching or forwarding; the base performance of each application.
PREF	Multiprocessor software pipelined prefetching.
FWD	Profile-based forwarding.
HYBR	Hybrid prefetching and forwarding that combines PREF and FWD.
F(s)	FWD results for smaller cache sizes.
H(s)	HYBR results for smaller cache sizes.
OPT	Optimal performance; provides a lower bound on execution time assuming that all memory references hit in cache and that all global memory latency is hidden.

Results of BASE and PREF simulations that use the smaller cache sizes are not shown. PREF results for smaller caches were almost identical to those for larger caches. BASE results for smaller caches were generally within ten percent of those for larger caches [8].

The relative amounts of prefetching and forwarding used under HYBR are indicated by the data in Table 2. The table shows the percentage of unique read and write references in each application code with write-to-read distances falling in various ranges. Writes are forwarded if their write-to-read distances are less than the specified threshold value (100 cycles), otherwise their corresponding reads are prefetched. Few read references were prefetched under HYBR for QCD for the threshold value used;

Table 2 - Breakdown of Execution Time Between Writes and Reads

	execution time range (cycles)						
code	1..49	50..99	100..499	500..999	1000..4999	5000..9999	10000+
reads	no pref.		prefetched				
TRFD	41%	0%	14%	0%	0%	0%	45%
QCD	89%	0%	1%	0%	0%	0%	10%
FLO52	36%	1%	7%	4%	12%	5%	35%
ARC2D	40%	1%	2%	1%	9%	4%	43%
DYFESM	34%	1%	10%	1%	7%	2%	44%
writes	forwarded		not forwarded				
TRFD	53%	0%	20%	7%	7%	0%	13%
QCD	85%	4%	5%	1%	2%	0%	3%
FLO52	62%	1%	11%	3%	8%	2%	13%
ARC2D	71%	1%	2%	1%	7%	1%	17%
DYFESM	69%	1%	6%	5%	8%	0%	11%

more read references were prefetched in the remaining codes. Substantial numbers of write references were forwarded in all codes; TRFD had the lowest percentage of forwarded writes. The data in Table 2 were collected on a per-array basis, not a per-element basis, in order to facilitate the source-level implementation of the hybrid scheme, as described in Section 3.4, and to illustrate how a compiler algorithm would apply prefetching and forwarding under the hybrid scheme for these codes. This organization of the data accounts for the high percentage of references with distances less than 50 cycles: the write-to-read distance recorded for a particular array reference is the time between the last write of any element of the array and the first read of any element by another processor.

The following sections present and discuss the results for each code in more detail, showing the performance of the various specific schemes. The execution time and cache behavior results presented are the arithmetic means of the results for all 32 processors executing each application code.

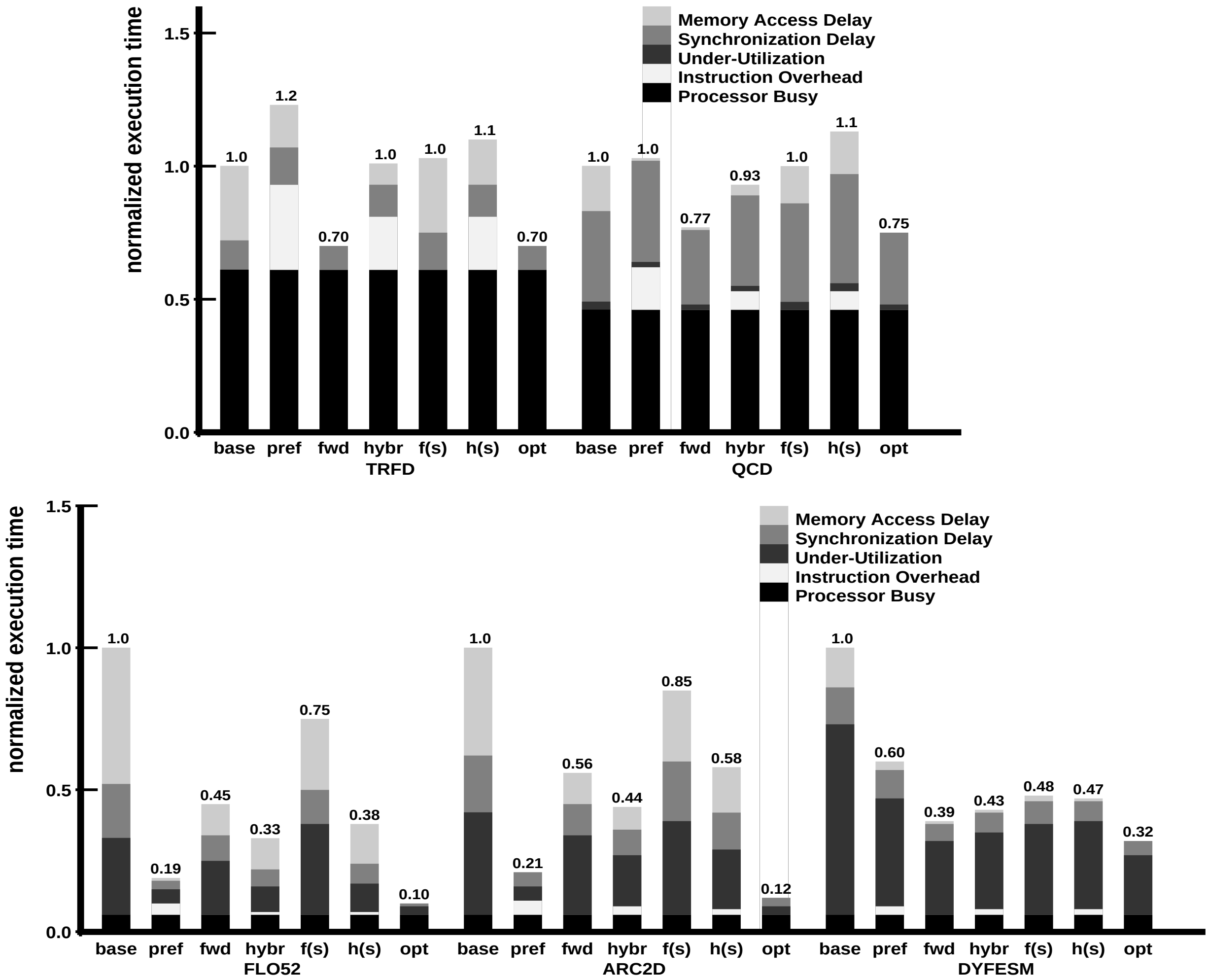


Figure 4 - Improvement in Execution Time

5.1 Improvement in Execution Time

The overall execution time results are given in Figure 4. Execution times are normalized to the BASE case for each code and are broken down into contributions from various sources:

1. Processor busy - time spent executing BASE instructions.
2. Instruction overhead - time spent executing additional instructions required to perform 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. In Cedar Fortran SDOALL loops, one processor in each cluster executes cluster serial code while the remaining processors in each cluster wait for the start of the next CDOALL loop enclosed by the SDOALL. Under-utilization time is a direct consequence of 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 - delay seen by processors on cache misses.

Forwarding performed considerably better than prefetching for TRFD and achieved near-optimal performance. Prefetching was not effective since reduced memory access delay was offset by large amounts of instruction overhead. FWD hid considerably more memory access delay than prefetching and did so with significantly lower instruction overhead, achieving near-optimal performance. Using the hybrid scheme in TRFD allowed instruction overhead and memory access delay to be reduced when compared to PREF; instruction overhead was reduced since less prefetching was performed. PREF was less effective than FWD in TRFD, in part, due to a lack of overlap between communication and computation. HYBR applied forwarding to sharing accesses with shorter overlap, resulting in greater reductions in memory access delay than those achieved when using prefetching alone.

Forwarding also performed better than prefetching for QCD and achieved near-optimal performance. Both prefetching and forwarding were effective in removing memory access delay, although FWD was slightly more effective and introduced very little instruction overhead. QCD has little memory access delay in the BASE code and has high instruction overhead under PREF. Execution time under HYBR improved somewhat compared to PREF, primarily due to a decrease in instruction overhead.

Prefetching and forwarding produced similar results in FLO52 and ARC2D. Prefetching was highly effective in hiding memory access delay, outperforming forwarding for both codes. A major contributor to increased performance under PREF was decreased under-utilization time. FLO52 and ARC2D each spend a considerable amount of their execution time in serial code (Table 1). PREF 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. HYBR improved performance in FLO52 and ARC2D, when compared to FWD, by decreasing under-utilization time through prefetching in parallel loops and serial code. Both FLO52 and ARC2D have significant amounts of spatial locality that is efficiently exploited by PREF to reduce memory latency. HYBR decreased instruction overhead compared to PREF but was not as effective overall due to higher memory access delay and under-utilization time.

Forwarding outperformed prefetching for DYFESM and achieved good performance when compared to OPT. Prefetching was effective in removing memory access delay. DYFESM makes extensive use of vector dot product and summation intrinsics; prefetching was applied to these operations to achieve the results shown. 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. The large percentage of under-utilization time in DYFESM is due to poor processor utilization, despite the fact that most of the program's execution is spent in parallel code (Table 1). This is caused by the high percentage of single-cluster CDOALL loops in DYFESM. HYBR performed nearly as well as did FWD for DYFESM. Execution time was slightly higher for HYBR due to a small amount of instruction overhead caused by prefetching.

The performance of HYBR fell somewhere between that of PREF and FWD for each code, improving on the performance of prefetching for TRFD, QCD, and DYFESM, and improving on the performance of forwarding for FLO52 and ARC2D. The performance of forwarding was noticeably worse in smaller caches because of significant increases in memory access delay, which were in turn caused by increases in conflict miss ratios (Section 5.2). The use of the hybrid scheme resulted in considerably less performance degradation in smaller caches when compared to forwarding alone. Forwarding for FLO52 and ARC2D in smaller caches also caused increases in under-utilization time that were minimized through the use of the hybrid scheme.

5.2 Cache Miss Behavior

Figure 5a - Figure 9a depict the miss ratio seen by processors for each code and scheme, broken down into contributions from various sources [3, 8]:

1. Nonsharing misses - conflict, capacity, or cold start misses.

2. Invalidation misses - sharing misses caused by invalidations by other processors.
3. Prefetched vs. nonprefetched misses - misses caused when accessing variables whose references were, or were not, prefetched.

Figure 5b - Figure 9b depict the miss ratio seen by the memory subsystem for each code and scheme, broken down into contributions from various sources [3, 8]:

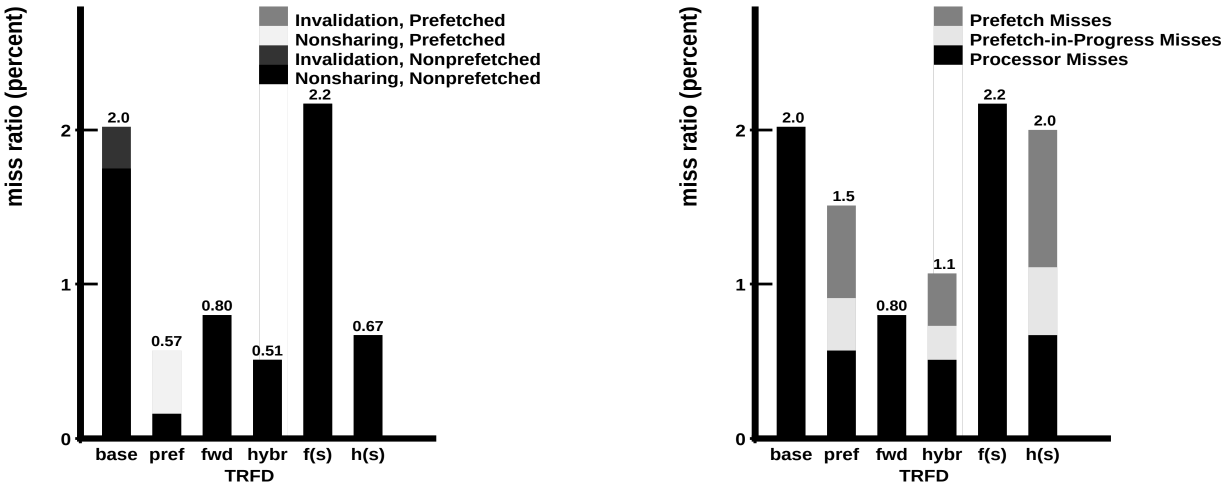
1. Processor misses - total misses seen by processors, from Figure 5a - Figure 9a.

The following types of misses do not contribute *directly* to processor miss latency and occur only under prefetching:

2. Prefetch-in-progress misses - caused by sharing or nonsharing accesses to prefetched variables where the prefetched data had not yet arrived at processor caches from main memory. These misses incur less latency than processor misses depending on the distance between the prefetches and the corresponding misses.
3. Prefetch misses - caused by prefetching instructions themselves when prefetched data are not already in requesting processors' caches. Although these misses cause more memory system activity, and thus increase contention delay for other accesses, they do not contribute directly to the latency observed by particular processors since prefetching instructions are non-blocking.

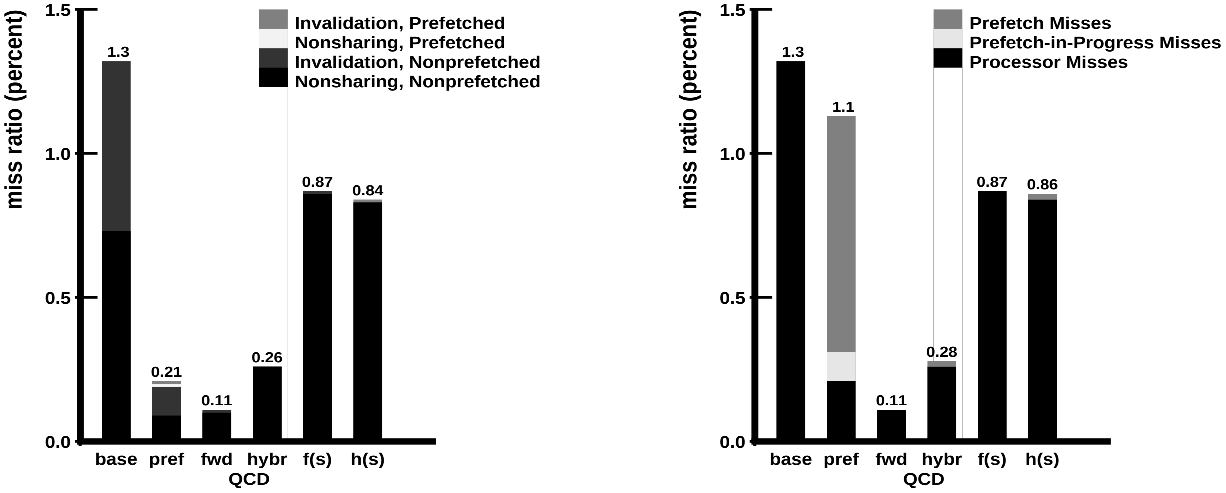
The majority of cache misses in TRFD were nonsharing misses. Most of the nonsharing misses were cold start misses resulting from the initial generation of program data structures and their communication to other processors at the start of the computation [8]. Many nonsharing accesses were prefetched under PREF, but some nonsharing prefetched misses still occurred; prefetching could not be used to eliminate cold start write misses. Prefetching was successful, as was FWD, in eliminating invalidation misses. FWD had a higher nonsharing miss ratio than PREF, but FWD had a lower overall miss ratio as prefetching caused many prefetch-in-progress and prefetch misses. The 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. HYBR applied forwarding to sharing accesses with less overlap, resulting in lower prefetch-in-progress and prefetch miss ratios when compared to PREF. The overall miss ratio for HYBR was greater than that of FWD, despite the fact that HYBR achieved the lowest processor miss ratio, due to prefetch-in-progress and prefetch misses.

Prefetching was effective for QCD in attacking both invalidation and nonsharing misses but caused



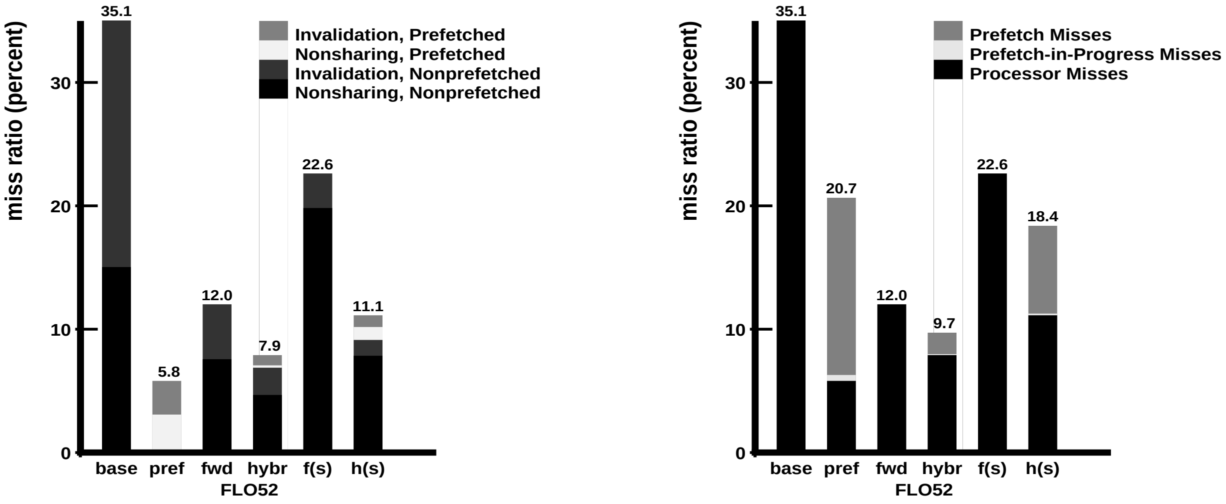
(a) classification of processor miss types **(b) processor, prefetch-in-progress, and prefetch misses**

Figure 5 - TRFD Cache Miss Behavior



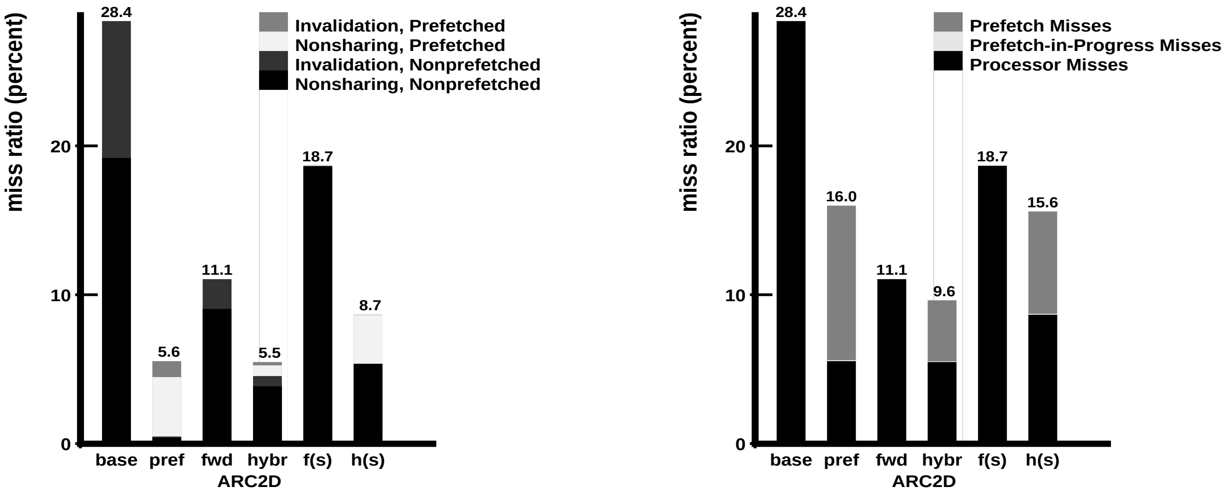
(a) classification of processor miss types **(b) processor, prefetch-in-progress, and prefetch misses**

Figure 6 - QCD Cache Miss Behavior



(a) classification of processor miss types **(b) processor, prefetch-in-progress, and prefetch misses**

Figure 7 - FLO52 Cache Miss Behavior



(a) classification of processor miss types **(b) processor, prefetch-in-progress, and prefetch misses**

Figure 8 - ARC2D Cache Miss Behavior

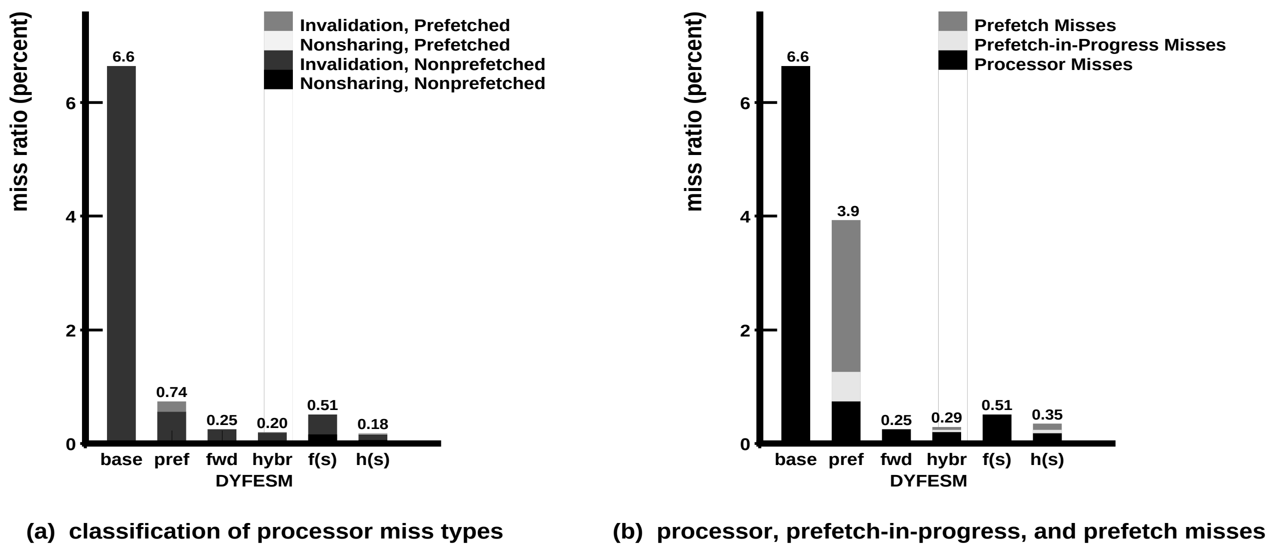


Figure 9 - DYFESM Cache Miss Behavior

some ineffective prefetching (prefetch-in-progress and prefetch misses). The nonsharing misses in QCD were primarily cold start misses; nonsharing, nonprefetched misses remaining under PREF were cold start write misses (see TRFD, above). FWD achieved a lower processor miss ratio than PREF by eliminating most invalidation misses. Like TRFD, QCD exhibits a high percentage of prefetch misses; however, HYBR was able to make a substantial reduction in prefetch misses in QCD.

Cache effects dominated the performance of prefetching and forwarding in FLO52 and ARC2D. The BASE codes had high processor miss ratios due to frequent, large vector accesses, significant spatial locality, and the small cache block size used. Prefetching was successful in eliminating many of the invalidation misses that occurred in the BASE codes. Despite a high percentage of prefetch misses, PREF resulted in substantially lower overall miss ratios compared to BASE. FWD was not nearly as effective as was PREF in reducing miss ratios since FWD did not exploit the inherent spatial locality in the codes as effectively as did PREF. FWD caused increased numbers of conflict and invalidation misses when compared to PREF, indicating that forwarding was somewhat aggressive and replaced needed data in destination processor caches. HYBR resulted in reasonable processor miss ratios compared to PREF or FWD and achieved the lowest overall miss ratios since it significantly reduced prefetch misses.

Invalidation misses were the major contributor to miss ratios in DYFESM. Most of these misses were not prefetched unless vector dot product and related intrinsics were prefetched as well. FWD was more effective in attacking invalidation misses and resulted in lower processor and overall miss ratios when compared to PREF. Miss ratios under prefetching were aggravated considerably by prefetch-in-progress and prefetch misses. Miss ratios were comparable between FWD and HYBR. HYBR was able to make a substantial impact on prefetch misses, considerably decreasing bandwidth requirements when compared to PREF.

Processor and overall miss ratios under FWD were considerably worse in smaller caches because of increases in conflict (nonsharing) misses. The use of HYBR resulted in considerably less degradation in miss ratios in smaller caches, when compared to FWD, for TRFD, FLO52, and ARC2D. Miss ratios under HYBR were also less sensitive to decreased cache size than those of FWD for QCD, and reducing the cache size resulted in a slight decrease in miss ratios under HYBR for DYFESM.

6. DISCUSSION OF RESULTS

Prefetching was effective in terms of overall performance for all codes except TRFD. Prefetching for vector intrinsic functions was necessary to achieve the results shown for DYFESM, which indicates the need to consider these intrinsics as part of an overall prefetching strategy. For FLO52 and ARC2D, the two most heavily parallel, vectorizable codes with higher spatial locality, PREF was effective, despite its high 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.

Forwarding was highly effective for TRFD, QCD, and DYFESM, and resulted in near-optimal performance, the elimination of cache misses, and decreased memory latency when compared to prefetching. Forwarding was also more effective than prefetching for these codes because forwarding eliminated the instruction overhead of prefetching through the use of Forwarding Write operations. Reduced overlap between communication and computation allowed forwarding to outperform prefetching in TRFD since data were forwarded to destination processors immediately after being produced. Forwarding was less successful at decreasing memory access delay in FLO52 and ARC2D when compared to prefetching.

For TRFD, QCD, and DYFESM, HYBR used forwarding to enhance the performance of prefetching. For FLO52 and ARC2D, the addition of prefetching under HYBR increased the performance of forwarding. For all five codes, HYBR performance fell somewhere between the performance of PREF and FWD; the hybrid scheme offered more performance stability for the codes studied than either prefetching or forwarding alone. HYBR was effective in reducing instruction overhead due to prefetching as well as under-utilization time. HYBR was also extremely effective in reducing prefetch misses except in TRFD, because of the latter code’s low overlap between communication and computation and high prefetch-in-progress miss ratio. The ability to reduce prefetch miss ratios makes the hybrid scheme a more efficient memory latency reduction mechanism that requires less memory system bandwidth. The hybrid scheme resulted in lower processor cache miss ratios than either prefetching or forwarding alone in three of the five codes, but instruction overhead, prefetch-in-progress misses, and prefetch misses due to prefetching negated some of this performance advantage.

The threshold value used by the hybrid scheme can be changed to tune performance for a particular application or architectural configuration. The best performance for TRFD, QCD, or DYFESM would be obtained with a high threshold value to allow more aggressive forwarding. For FLO52 and ARC2D, prefetching is efficient and more effective, therefore a low threshold value should be used. Higher threshold values decrease high instruction overhead by decreasing the amount of prefetching. Lower threshold values allow spatial locality to be exploited and under-utilization time to be decreased by increasing the amount of prefetching.

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. Not using selective prefetching may tend to cause ineffective prefetching and increased prefetch miss ratios since nonsharing accesses that would not have caused cache misses may be prefetched. This effect is seen in TRFD and QCD, two codes with large numbers of nonsharing and prefetch misses. Despite the fact that selective prefetching was not used, software pipelined prefetching performed well. This was due, in part, to the ample memory bandwidth provided by the modeled system.

Prefetching was effective in attacking both conflict (FLO52, ARC2D) and invalidation misses. Prefetching was found to be less effective in attacking invalidation misses in [3]; however, false sharing was shown to be a contributor to this effect. A small cache block size was used in the experiments described in Section 5 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 forwarding can affect only sharing accesses. In FLO52 and ARC2D, aggressive forwarding caused increased numbers of conflict misses as needed data were replaced in destination processor caches.

The performance of prefetching was similar in both larger and smaller caches for each of the five codes studied, despite the fact that selective prefetching [2] was not used. The large increases in memory access delay under forwarding in smaller caches were caused by significant increases in conflict miss ratios. This suggests that FWD was too aggressive and caused undesirable replacement in destination processors' caches. Receiver-initiated latency reduction mechanisms, such as prefetching, may be better equipped to handle finite cache size effects, since they can reload data that were unnecessarily replaced before being used. Sender-initiated forwarding mechanisms cannot resend data if they are inadvertently replaced in destination processors' caches before being used.

The hybrid scheme significantly reduced the sensitivity of the performance of forwarding to processor cache size. The use of the hybrid scheme in smaller caches resulted in decreased conflict miss ratios, decreased overall miss ratios, decreased memory access delay, and even decreased under-utilization time, when compared to forwarding alone in smaller caches. The threshold value in the hybrid scheme can be adjusted to account for cache size and organizational parameters, as well as other application and architecture characteristics. The use of an alternative cost function in the hybrid scheme may also improve the performance of forwarding in smaller caches. For example, Koufaty et al. [28] propose a cost function based on the number of cache lines used by a destination processor between a Forwarding Write and its subsequent read on that processor.

7. COMPILER ALGORITHMS FOR FORWARDING

Given that previous studies have not considered data forwarding for the purposes described in this paper, little attention has been paid to the construction of compiler algorithms for forwarding. The experimental results described in Section 5 evaluate forwarding through the use of the profiling techniques described in Section 3.3. This section describes a compiler algorithm for software-initiated data forwarding [8] and discusses its effectiveness with respect to identifying and forwarding data for sharing accesses between successive parallel loops.

7.1 Description of Algorithm

The forwarding compiler algorithm consists of several steps:

1. Mark eligible write references for forwarding. The current implementation of the forwarding algorithm assumes that all shared read/write accesses are potential candidates for forwarding. Classification of access types is performed by using extensions to the profiling instrumentation described in Section 3.3.

For each eligible write:

2. Locate the most likely downstream read for this write, forming a write/read reference pair. Downstream reads are only identified in successive parallel loops or loop nests. During this step, writes and reads are matched by comparing only symbol names, not subscript expressions. Branch profiling is used to predict the outcome of conditional branches.
3. Compute a destination processor expression for this write/read reference pair, and transform the write to a Forwarding Write that uses this destination processor expression. This step is discussed in more detail below.

An extension to the forwarding compiler algorithm implements a fully compiler-based version of the hybrid prefetching and forwarding scheme. This is accomplished by estimating write-to-read execution time distances while identifying write/read reference pairs. This type of execution time estimation is similar to that used in [2] for computing prefetch distances for software pipelined prefetching.

The third step in the forwarding compiler algorithm attempts to determine, for a given write/read reference pair, an expression for a single processor number to which each write should be forwarded. This is easily accomplished, if round-robin scheduling is employed, by analyzing loop index, loop bounds, and access subscript information. This type of analysis has been applied for generating explicit message passing within a single loop nest for forwarding in DOACROSS loops [17], as well as for compiling languages such as Fortran D for message passing machines [29]. In cache coherent, shared memory architectures, data distributions are implied for each loop and are based on loop partitioning and scheduling, which considerably simplifies compiler analysis for forwarding; however, compiler analysis for forwarding is more complex, when compared to these other cases, since there are two data distributions involved: one for the loop containing the write, and one for the downstream loop containing the read.

For the loops:

```
DOALL i = L_i , H_i , S_i
     A(i) = ...
...
DOALL j = L_j , H_j , S_j
     ... = A(j)
```

$$
\begin{aligned}
&\texttt{DOALL i = } L_i \texttt{ , } H_i \texttt{ , } S_i \\
&\quad \texttt{A(i) = ...} \\
&\texttt{...} \\
&\texttt{DOALL j = } L_j \texttt{ , } H_j \texttt{ , } S_j \\
&\quad \texttt{... = A(j)}
\end{aligned}
$$

the write loop is transformed to:

$$
\begin{aligned}
&\texttt{DOALL i = } L_i \texttt{ , } H_i \texttt{ , } S_i \\
&\quad \texttt{if \{ [(i - } L_j\texttt{) mod } S_j \texttt{ <> 0] or (} N_i \texttt{ = p) \} A(i) = ...} \\
&\qquad \texttt{else Forwarding\_Write (A(i) , p) = ...}
\end{aligned}
$$

where

$$
\begin{aligned}
&p &&= [\ (i - L_j) / S_j\ ] \bmod P \\
&N_i &&= \text{the normalized write loop index} \\
& &&= (i - L_i) / S_i \\
&E_i &&= \text{the write loop extent, or trip count} \\
& &&= \text{ceil}\ [\ (H_i - L_i + 1) / S_i\ ] \\
&(\ 0 <= N_i < E_i\ )
\end{aligned}
$$

Figure 10 - Example Forwarding Transformation

An example of one case that is handled by the forwarding compiler algorithm is shown in Figure 10. In this example, a write to an array `A` has a downstream read in a subsequent loop. The write loop is transformed so that each write to `A(i)` is transformed to a Forwarding Write that forwards `A(i)` to processor `p` (`p: 0 <= p < P`, the number of processors executing the loops). Two conditional tests are performed before each Forwarding Write; if either test fails, a regular write is performed. The first test ensures that `A(i)` will be read in the second loop, while taking the strides of the two loops into account. The second test prevents unnecessary forwarding to the processor that is issuing the write. Additional conditional tests, as well as loop splitting transformations (similar to those described in [2]), to prevent forwarding past loop bounds are described in [8].

The forwarding compiler algorithm handles arbitrary nested loops by assuming that round-robin

Table 3 - Forwarding Pattern Statistics

code	distribution of destination processors						percentage of dynamic pattern regularity
	0	1	<=4	cluster	multi-cast	broad-cast	
TRFD	66.4%	1.1%	32.5%	0.0%	0.0%	0.0%	99.1%
QCD	96.4%	0.0%	1.8%	1.1%	0.0%	0.7%	91.9%
FLO52	45.4%	8.9%	35.7%	8.0%	2.0%	0.0%	76.0%
ARC2D	67.9%	10.2%	21.2%	0.5%	0.2%	0.0%	81.8%
DYFESM	16.0%	37.3%	13.5%	32.6%	0.6%	0.0%	93.3%

scheduling is reset at the start of each parallel loop; irregular and non-perfect loop nests are also handled under this assumption. Loop bounds and strides are not required to be known statically to the compiler. Many forms of subscript expressions can appear in write/read reference pairs. The example solution in Figure 10 can be generalized to linear subscript expressions by substituting for the loop bounds and/or stride. Arbitrarily complex subscript expressions can appear, including *coupled subscripts* [30] among enclosing loop index variables, as long as the innermost parallel loop in a write or read loop nest has a regular, strided access pattern (i.e., it can utilize a solution such as that of Figure 10) and its subscript and index are not coupled to any other array dimensions or enclosing loops.

7.2 Algorithm Effectiveness

The effectiveness of the forwarding compiler algorithm follows from the assumption that the use of point-to-point forwarding will achieve reasonable performance and that the compiler algorithm will be able to determine forwarding patterns. Forwarding patterns produced during profile-based forwarding were examined to determine the relative amounts of point-to-point forwarding performed and the degree of pattern regularity (Table 3).

The first part of Table 3 shows the distributions of destination processors for FWD and each code. These data summarize the dynamic percentage of write operations that were forwarded to zero, one, or several processors. The cluster case logically divides the processors into four clusters of eight processors each; the destination processor set consists of one or more clusters. The remaining cases are arbitrary multicast and broadcast. Most codes forwarded substantial portions of their writes to other processors; however, for the most part, forwarding could be accomplished by sending small numbers of point-to-point messages per Forwarding Write. The high percentage of cluster forwarding in DYFESM is caused by the high frequency of single-cluster parallel loops in this code.

The second part of Table 3 shows the degree of forwarding pattern regularity for each code. Destination processor patterns for FWD and each code were analyzed to determine the most likely (most frequently occurring) destination processor for each element of each forwarded array. The result of this analysis for a particular forwarded array is a vector of processor numbers (`p: 0 <= p < 32`). Each vector was summarized by dividing it into regular and irregular references, where regular references were defined to be those that fit into a pattern that could be characterized by a start value, length (with a minimum length of four references), and constant stride. The dynamic percentages of regular references reported for each code were weighted by the relative execution frequencies of each forwarded array. The high degree of pattern regularity is to be expected, given the nature of the applications and their use of vector operations, parallel loops, and round-robin scheduling.

The data in Table 3 indicate that large numbers of sharing accesses exist in the application codes studied. For most of these accesses, data forwarding could be accomplished by sending data to a small number of destination processors by using point-to-point messages. Furthermore, the patterns of destination processors for these sharing accesses were predominantly regular and strided. Although a more detailed experimental analysis of an actual implementation of the proposed forwarding compiler algorithm is required to assess its overall applicability and performance, the results presented in this section suggest that the algorithm targets many of the important sharing accesses in the codes studied and should achieve reasonable performance.

8. 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, shared memory multiprocessors executing large, optimized, numerical application codes. A multiprocessor software-pipelined data prefetching algorithm has been described and evaluated. The Forwarding Write operation was introduced and used to evaluate the performance of data forwarding by using a profiling approach. Data forwarding focused on reducing memory latency for sharing accesses caused by interprocessor communication between, rather than within, parallel loops. These data prefetching and data forwarding strategies were combined in a new hybrid scheme that provided several performance advantages when compared to prefetching or forwarding alone. A compiler algorithm for data forwarding was introduced that obviates the need for the profiling methodology used to evaluate forwarding. This algorithm was shown to address the types of forwarding patterns most frequently encountered in the application codes studied.

Prefetching and forwarding were each effective for application codes with different characteristics. Prefetching outperformed forwarding for codes with higher spatial locality and higher conflict miss ratios. 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. Forwarding provided significant reductions in memory latency for sharing accesses in these codes and reduced instruction overhead through the use of Forwarding Write operations. Prefetching offered comparable performance in both larger and smaller caches despite the fact that selective prefetching was not used. Forwarding in smaller caches suffered from performance degradation that was caused by unnecessary cache replacement, but the hybrid prefetching and forwarding scheme was used successfully to counter this effect.

The new hybrid prefetching and forwarding scheme offered more stable performance than either prefetching or forwarding alone over varying application code characteristics and varying cache sizes. When compared to prefetching alone, the use of the hybrid scheme reduced instruction overhead, under-utilization time, and memory system bandwidth requirements, and produced lower processor cache miss ratios in several codes. The hybrid scheme significantly reduced the sensitivity of forwarding performance to processor cache size. In smaller caches, this reduced sensitivity resulted in lower conflict and overall cache miss ratios, lower memory access delay, and decreased under-utilization time when compared to forwarding alone.

The threshold parameter value specified to the hybrid scheme can be adjusted to account for architectural parameters, such as cache size, as well as application characteristics. Application characteristics that favor forwarding, and thus higher threshold values, include low overlap between communication and computation, increased levels of sharing, and high prefetching instruction overhead.

Conditions that favor prefetching, and thus lower threshold values, include smaller caches, higher levels of spatial locality, and high under-utilization time. The use of alternative cost functions may yield more robust, efficient hybrid schemes, particularly with respect to the performance of forwarding in smaller caches; this remains a topic for further study.

A compiler algorithm for data forwarding has been presented. In contrast with previous forwarding approaches, this algorithm considers forwarding for communication between successive parallel loops. The compiler algorithm implements point-to-point forwarding for regular, strided array accesses involved in interprocessor communication between successive parallel loops. The performance of forwarding when using this algorithm hinges on the ability to identify point-to-point forwarding patterns involving regular, strided array accesses. Experimental results show that the majority of global array accesses in the codes studied fall into this category, which suggests that the forwarding compiler algorithm should be effective for this class of applications.

ACKNOWLEDGMENTS

The authors would like to thank John Andrews and Carl Beckmann for their comments on drafts of this paper.

REFERENCES

1. Callahan, D., Kennedy, K., and Porterfield, A. K. Software prefetching. *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1991, pp. 40-52.
2. Mowry, T. C., Lam, M. S., and Gupta, A. Design and evaluation of a compiler algorithm for prefetching. *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1992, pp. 62-73.
3. Tullsen, D. M., and Eggers, S. J. Limitations of cache prefetching on a bus-based multiprocessor. *Proc. 20th Annual International Symposium on Computer Architecture*. 1993, pp. 278-288.
4. Mowry, T. C., and Gupta, A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.* 12, 2 (June 1991), 87-106.
5. Kuck, D., et al. The Cedar system and an initial performance study. *Proc. 20th Annual International Symposium on Computer Architecture*. 1993, pp. 213-223.

6. Poulsen, D. K., and Yew, P.-C. Data prefetching and data forwarding in shared memory multiprocessors. *Proc. 1994 International Conference on Parallel Processing*. 1994, pp. II-276-280.
7. Andrews, J. B., Beckmann, C. J., and Poulsen, D. K. Notification and multicast networks for synchronization and coherence. *J. Parallel Distrib. Comput.* 15, 4 (August 1992), 332-350.
8. Poulsen, D. K. Memory latency reduction via data prefetching and data forwarding in shared memory multiprocessors. Ph.D. dissertation, CSRD Report 1377, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, September 1994.
9. Poulsen, D. K., and Yew, P.-C. Execution-driven tools for parallel simulation of parallel architectures and applications. *Proc. Supercomputing '93*. 1993, pp. 860-869.
10. Berry, M., et al. The Perfect Club benchmarks: effective performance evaluation of supercomputers. *Int'l. J. Supercomp. Applications* 3, 3 (Fall 1989), 5-40.
11. Padua, D., et al. The Cedar Fortran project. CSRD Report 1262, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1992.
12. Poulsen, D. K., and Yew, P.-C. EPG source code instrumentation tools - user manual. CSRD Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1994.
13. Polychronopoulos, C. D., et al. Parafrase-2: an environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. *Proc. 1989 International Conference on Parallel Processing*. 1989, pp. II-39-48.
14. Lenoski, D. E., et al. The Stanford Dash multiprocessor. *IEEE Computer* 25, 3 (March 1992), 63-79.
15. Frank, M. I., and Vernon, M. K. A hybrid shared memory / message passing parallel machine. *Proc. 1993 International Conference on Parallel Processing*. 1993, pp. I-232-236.
16. Hill, M. D., et al. Cooperative shared memory: software and hardware for scalable multiprocessors. *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1992, pp. 262-273.
17. Su, H.-M., and Yew, P.-C. Efficient doacross synchronization on distributed shared-memory multiprocessors. *Proc. Supercomputing '91*. 1991, pp. 842-853.
18. Lee, J., and Ramachandran, U. Architectural primitives for a scalable shared memory

multiprocessor. *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*. 1991, pp. 103-114.

19. Goodman, J. R., Vernon, M. K., and Woest, P. J. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 1989, pp. 64-75.
20. Weber, W.-D., and Gupta, A. Analysis of cache invalidation patterns in multiprocessors. *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 1989, pp. 243-256.
21. Mounes-Toussi, F., and Lilja, D. J. Using compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics. *IEEE Trans. Parallel and Distr. Sys.* (1994), to appear.
22. KSR1 Principles of Operation. Kendall Square Research Corporation, 1991.
23. Mowry, T. C. Tolerating latency through software-controlled data prefetching. Ph.D. dissertation, Department of Electrical Engineering, Stanford University, March 1994.
24. Chen, T.-F. Data prefetching for high-performance processors. Ph.D dissertation, Technical Report 93-07-01, Department of Computer Science and Engineering, University of Washington, July 1993.
25. Kroft, D. Lockup-free instruction fetch / prefetch cache organization. *Proc. 8th Annual International Symposium on Computer Architecture*. 1981, pp. 81-87.
26. Kruskal, C. P., and Snir, M. The performance of multistage interconnection networks for multiprocessors. *IEEE Trans. Comput.* C-32, 12 (December 1983), 1091-1098.
27. Gupta, A., Weber, W.-D., and Mowry, T. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. *Proc. 1990 International Conference on Parallel Processing*. 1990, pp. I-312-321.
28. Koufaty, D. A., et al. Data forwarding in scalable shared-memory multiprocessors. Submitted to *1995 International Conference on Supercomputing*.
29. Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiling Fortran D for MIMD distributed-memory machines. *Comm. ACM* 35, 8 (August 1992), 66-80.
30. Li, Z., Yew, P.-C., and Zhu, C.-Q. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Par. and Distr. Sys.* 1, 1 (January 1990), 26-34.

FIGURE LEGENDS

AUTHOR BIOGRAPHIES

David K. Poulsen received the B.S. in Electrical Engineering from the University of Wisconsin-Madison in 1984, the M.S. in Computer Engineering from Syracuse University in 1988, and the Ph.D. in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1994. Employed by the IBM Corporation from 1984 to 1993 as an ES/9000 processor designer, he was awarded the IBM Resident Study Fellowship in order to pursue his graduate studies at the University of Illinois. He was a research assistant at the Center for Supercomputing Research and Development at the University of Illinois from 1988 to 1994. He is currently employed by Kuck and Associates, Inc., of Champaign, Illinois as a Senior Developer. His research interests include scalable parallel architectures, cache coherence, memory consistency, parallelizing compilers, and performance evaluation.

Pen-Chung Yew received the B.S. in Electrical Engineering from the National Taiwan University, the M.S. in Electrical and Computer Engineering from the University of Massachusetts at Amherst, and the Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. He is currently on the faculty of the Department of Computer Science at the University of Minnesota. From 1991 to 1992, he served as the Program Director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, D.C.. Dr. Yew was a Distinguished Visitor of the IEEE Computer Society from 1990 to 1993. He is on the advisory committee of the IEEE Technical Committee on Computer Architecture and the IEEE Technical Committee on Parallel Processing. He served as a co-chairman of the 1990 International Conference on Parallel Processing, and was a general co-chairman of the 1994 International Symposium on Computer Architecture. His research interests include high-performance multiprocessor system design, parallelizing compilers, computer architecture, and performance evaluation.