# Data Forwarding in Scalable Shared-Memory Multiprocessors

David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas, *Member, IEEE*

**Abstract**—Scalable shared-memory multiprocessors are often slowed down by long-latency memory accesses. One way to cope with this problem is to use data forwarding to overlap memory accesses with computation. With data forwarding, when a processor produces a datum, in addition to updating its cache, it sends a copy of the datum to the caches of the processors that the compiler identified as consumers of it. As a result, when the consumer processors access the datum, they find it in their caches.

This paper addresses two main issues. First, it presents a framework for a compiler algorithm for forwarding. Second, using address traces, it evaluates the performance impact of different levels of support for forwarding. Our simulations of a 32-processor machine show that an optimistic support for forwarding speeds up five applications by an average of 50% for large caches and 30% for small caches. For large caches, most sharing read misses are eliminated, while for small caches, forwarding does not increase the number of conflict misses significantly. Overall, support for forwarding in shared-memory multiprocessors promises to deliver good application speedups.

**Index Terms**—Memory latency hiding, forwarding and prefetching, multiprocessor caches, scalable shared-memory multiprocessors, address trace analysis.

## 1 Introduction

Scalable shared-memory multiprocessors are often slowed down by long-latency memory accesses. To cope with this problem, researchers have proposed techniques that reduce the latency of memory accesses. Example of such techniques are sophisticated cache hierarchies or compiler optimizations for locality enhancement. However, while these techniques indeed reduce the average memory access latency significantly, they tend to handle sharing-induced misses poorly.

To deal with sharing-induced misses, we often need techniques that overlap memory accesses with computation or with other memory accesses. These techniques include multithreading [2], relaxed memory consistency models [1], [7], data prefetching [13], and data forwarding [19]. In both prefetching and forwarding, the data is moved close to the consumer processors before it is actually needed. Then, when the consumer processors finally access the data, they can do so with low latency. In data prefetching, the consumer processors request the data in advance; in data forwarding, the producer processor, after updating its cache, sends a copy of the data to the consumer processors.

Data forwarding is different from update cache coherence protocols [22]. In an update protocol, when a processor writes, all current sharer processors are updated. In our design of data forwarding, instead, the producer processor updates a set of processors selected by the compiler. The updated processors may or may not be current sharers, and the rest of the current sharers are invalidated. Data forwarding, in fact, has much in common with data prefetching. It can be thought of as producer-initiated prefetching. Data forwarding and prefetching are complementary techniques that can be combined. In this paper, however, we focus on data forwarding exclusively.

There is little previous work on data forwarding. Forwarding was added to the DASH architecture with the *Deliver* instruction [12]. This instruction sends the data to several clusters specified in the instruction with a bit vector. A recent performance evaluation of the deliver instruction [15] shows that, in general, prefetching is more effective than the deliver instruction. The deliver instruction is reported to outperform prefetching only when the cache size and the memory latency are very large. However, the consumer processors are computed at run-time, using special hardware to predict the future communication behavior based on the recent history of the application behavior. In this paper, instead, the compiler is responsible for identifying the consumer processors. Poulsen and Yew [17], [19] studied compiler-initiated data forwarding alone and in combination with data prefetching. They showed that, in both cases, large performance gains can be achieved. However, they only examined large caches, which tend to show only the positive side of forwarding. Furthermore, they only evaluated a simple strategy for forwarding.

There are other related schemes that allow a processor to broadcast the updated value of a variable. These schemes are the KSR Poststore mechanism [21], the DASH Update Write [12], the Ultracomputer UpdateAll primitive [4], and the Multicube Notify primitive [9]. However, in both the KSR Poststore mechanism and the DASH Update Write, only the processors holding a copy of that data receive the update.

- *D.A. Koufaty and J. Torrellas are with the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1308 W. Main Ave., Urbana, IL 61801. E-mail: {koufaty, torrella}@csrd.uiuc.edu.*
- *X. Chen is with Silicon Graphics Inc., Mountain View, CA 94043. E-mail: xchen@mti.sgi.com.*
- *D.K. Poulsen is with Kuck and Associates, Inc., Champaign, IL 61820. E-mail: poulsen@kai.com.*

*Manuscript received April 18, 1995. A shorter version of this paper was presented at the Ninth International Conference on Supercomputing, July 1995. For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95235.*

Moreover, the UpdateAll and Notify primitives were both proposed and studied primarily to optimize interprocessor synchronization operations—not the sharing of data.

In this paper, we present a framework for a compiler algorithm for forwarding. In addition, using address traces, we evaluate the performance impact of different levels of support for forwarding. We simulate a 32-processor scalable shared-memory multiprocessor. We find that an optimistic support for forwarding speeds up five Perfect Club codes by an average of 50% for large caches and 30% for small caches. For large caches, most sharing read misses are eliminated, while for small caches, forwarding does not increase the number of conflict misses significantly.

The rest of this paper is organized in four sections: Section 2 describes data forwarding; Section 3 presents the framework for a compiler algorithm; Section 4 describes the method used to evaluate the performance impact of data forwarding; finally, Section 5 evaluates data forwarding.

## 2 Data Forwarding

Data forwarding is a technique that can be used in shared-memory multiprocessors to hide the latency of interprocessor communication. In data forwarding, when a processor updates a word in its cache, it also propagates the update to the caches of processors that are expected to use the word in the near future. Hopefully, when these consumer processors read the word later on, they will find the updated data in their caches and, therefore, proceed without stalling.

Data forwarding can be integrated well with invalidation-based hardware cache coherence protocols. Indeed, the producer processor sends a forward message with the update and the list of processors to receive the update. The message is sent to the directory that keeps the list of sharers of the word. The directory will update memory, send the update to the consumer processors, and mark the producer and the consumer processors as sharers. Furthermore, it will send invalidations to all the other processors that currently have the data and remove them from the sharer list. When the caches of the consumer processors receive the update, they send an acknowledgment message to the producer processor to inform it of the completion of the transaction. Note, therefore, the difference between an update protocol and data forwarding: In an update protocol, all current sharers are updated; in data forwarding, a set of processors that may or may not be current sharers are updated and any current sharers that are not in this set are invalidated. Note that data forwarding can displace data from the consumer's cache. This can cause deadlock if the displaced data is dirty and the write buffer cannot take because it is full. To avoid this deadlock the data being forwarded is discarded if the data being evicted is in state dirty and the write buffer is full.

Data forwarding can be supported efficiently in caches with multiword lines. Following Poulsen and Yew's work [19], we support it with a *Write-and-Forward* assembly instruction. This instruction is inserted by the compiler in lieu of an ordinary write instruction. It uses one more register than an ordinary write instruction: A mask register that indicates the processors that should receive the update. This register is used implicitly by the instruction, and must have as many bits as there are processors in the machine. A write-and-forward instruction does not stall the processor. If the write-and-forward access hits a line that is dirty in the producer's cache, the whole cache line is forwarded to the directory-memory and the line is marked shared in the producer's cache. Otherwise, if the access hits a shared line or misses in the producer's cache, only the updated word is forwarded to the directory-memory. If the access missed in the producer's cache, the directory-memory will send the complete memory line to the producer processor in addition to sending it to the consumer processors. In all cases, for every forward, an entry is allocated in the write buffer of the producer processor and kept until the acknowledgments from the consumers have been received. At that point, it can be retired. Note that a nonforwarding system does not allocate an entry in the write buffer for writes to dirty data. Therefore, the write buffer in forwarding system needs to be deeper than in nonforwarding systems.

The reason why forward messages go first to the directory and then to the consumer processors instead of going directly to the latter is to prevent races in the cache coherence protocol. Indeed, by sending the forwards to the directory, we make sure that two updates to the same cache line are serialized in the directory. If, instead, the forwards were sent directly from the producer to the consumer cache without a mid-trip hop in the directory, races could occur. For example, suppose that the producer processors send updates directly to the consumers and, in parallel, send notification messages to the directory. In this case, imagine that a processor issues a write-and-forward to a line and a second processor issues an ordinary write to the same line. Suppose that the directory-memory observes the resulting forward notification message first and the invalidation operation corresponding to the write later. It is possible that, because of network traffic delays, a third processor receives the messages in the opposite order: the invalidation first and the forward later. This situation would cause the cache in the third processor to wrongly store the forwarded data. If the messages had been serialized in the directory, they would have arrived at the third processor in the same order. Of course, we are assuming that we use deterministic routing of messages.

Still, there are ways to support this one-hop protocol without races. One such way is to prevent the forwarded data from immediately modifying the cache of the consumer processor. Instead, when the data arrives at the consumer node, it is deposited in a *Receive* buffer. The receive buffer can hold several cache lines. When a processor misses in the cache, it always checks the receive buffer to see if the data has been forwarded but not yet copied into the cache. If so, the line is read but not copied into the cache. A forwarded line in the receive buffer will be copied into the cache or discarded only when the consumer processor receives, from the directory-memory, a *Commit* message corresponding to that forward. The directory-memory sends a commit message to all the consumers when it receives the forward notification message coming from the producer processor. Overall, given the complexity of this aggressive scheme and the fact that consumers are unlikely to notice much of a difference whether the data is forwarded to them in one or two hops, we use the cheaper two--hop forwarding algorithm.

As indicated before, in the two-hop forwarding algorithm, the directory-memory serializes the state changes for a memory line. If a forward reaches a directory and the line that it wants is changing states at that time, the directory bounces the forward. In this case, the originating processor is forced to retry the transaction. A similar approach is used in the transactions of the DASH cache coherence protocol [12].

Even if, in the end, the forwarded data is safely deposited in the cache of the consumer processors, three problems may occur. First, a consumer processor may request the data before it arrives from the forwarding processor. In this case, the consumer will have to stall until the data arrives and, therefore, only part of the communication latency will be hidden. A second problem occurs when the data arrives and is later displaced from the consumer cache by other data before it is used. Although this case will not have any effect on the miss rate, the instruction overhead of the forward will decrease performance. Finally, the forwarded data may displace useful data from the consumer cache. This can create an additional miss that brings the original data back into the cache and displaces the forwarded data. To avoid the last two problems, the compiler could identify the conflict and disable the forward. Alternatively, the compiler can reduce the likelihood of this problem by forwarding data only if the consumer processor is unlikely to access much data between the reception and the use of the forwarded line. We will evaluate this idea in Section 5.

Overall, we see that forwarding can interact with the miss rate of an application in several ways. In the more intuitive case of single-word cache lines, forwarding can eliminate true sharing and cold misses or induce conflict misses. These cases are illustrated in Fig. 1. In the figure, words $x$ and $y$ belong to two cache lines that conflict in the cache. The figure shows how forwarding can remove a true sharing miss (Chart a), remove a cold miss (Chart b), and induce a conflict miss (Chart c).

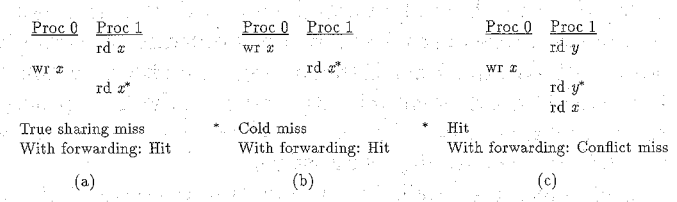



Fig. 1. Interaction of forwarding with the misses in caches with single-word lines.

In addition, depending on how we count the misses, forwarding may also appear to have other effects like eliminating conflict misses. Indeed, consider a word $x$ that is referenced and brought into the cache, then replaced by another word $y$, and finally referenced again. This second reference to $x$ causes a conflict miss. Suppose that, before this second reference to $x$, another processor forwards $x$ to the cache preventing the conflict miss. While the forward is supplying data that is truly shared, the accounting may indicate that it is eliminating a conflict miss. Obviously, the conflict miss is hiding a sharing access that is amenable to forwarding. We look at this issue of fewer conflicts in Section 5.3.

## 2.1 Optimizations

The simple forwarding scheme described so far can be improved in several ways. In this section, we describe some optimizations, although we will not support them in the evaluation section.

**Combining.** Obviously, if a processor writes the same word several times in a row before the word is needed by any other processor, only the last write needs to be forwarded. An extension of this is to combine several forward messages into one if they are induced by the same processor updating different words of the same cache line. This will reduce network traffic. Such combining can be initiated either by the compiler or by the hardware. In the former case, the compiler detects a group of write-and-forwards to different words of the same line that would be issued close in time. Then, the compiler replaces all of them except the last one by regular writes. Of course, the compiler should do this only if all the updates are still likely to arrive at the consumer caches on time. Alternatively, the hardware can perform combining if the producer cache keeps the forward in a buffer for a few cycles before sending it. If, in the meantime, the processor generates another forward to the same line, the two forwards are combined. This approach is used by Glasco et al. [8] in update protocols.

**Forwarding to the secondary cache only.** If the primary cache is not big enough to hold the working set plus the forwarded data, forwards may be sent to the secondary cache only. The system can be designed such that, if the primary cache of the consumer processor had an old copy of the data, it gets automatically updated when the forward is received. With this approach, we reduce conflict misses in small primary caches while bringing the data close to the consumer processor.

**Forwarding to memory.** In some cases, the compiler cannot determine which processors will consume the data. This may occur, for instance, when the data is consumed in a dynamically-scheduled parallel loop. In this case, the producer forwards the data to memory. As a result, the consumers can later get the data from memory instead of having to go all the way to the producer processor.

**Separate forwarding instruction.** The write-and-forward instruction as defined restricts data forwarding to happen when the write takes place. To deal with the cases where the forward should be delayed but the write should not, a special forward-only instruction can be used. An example of such instruction is the DASH deliver instruction [12].

**Software-Based Prefetching.** Forwarding is less appropriate when the consumer is unknown or when the consumer will not use the data until much later. In these cases, we can use prefetching. One hybrid forwarding-prefetching scheme has been studied by Poulsen and Yew [19]. To see the relationship between forwarding and prefetching, we finish this section with a comparison between the two schemes.

## 2.2 Comparison to Software-Based Prefetching

Data forwarding and data prefetching can be compared in terms of effectiveness, complexity of the compiler and hardware support required, and overhead involved. Starting with effectiveness, we note that prefetching can eliminate any kind of read misses, namely cold, conflict or coherence misses. Forwarding, on the other hand, can only eliminate coherence misses and the related case of cold misses on data produced by another processor (Fig. 1).

TABLE 1
COMPARING FORWARDING TO SOFTWARE-BASED PREFETCHING

| Issue | Comparison |
|---|---|
| Effectiveness | Prefetching targets all misses: cold, conflict, and coherence.<br>Forwarding can only target coherence and some classes of cold misses.<br>However, for theses misses, forwarding can outperform prefetching. |
| Compiler Support | Forwarding requires more sophisticated compiler support. |
| Hardware Support | Both strategies require a comparable amount of hardware support. |
| Overhead | Both strategies have a comparable instruction overhead and may cause cache conflicts. |

For the misses that forwarding can handle, however, forwarding can be more effective than prefetching. There are three reasons for this. Firstly, forwarding delivers the data to the consumer as soon as it is produced. Secondly, the data is transferred with small latency, namely with two hops: from producer processor to memory and a from memory to consumer processor. Finally, a producer can forward the same data to several consumers with a single instruction. With prefetching, none of this is true. First, the prefetch rarely reaches the producer right at the time when the data is produced. If the prefetch is issued too early, it will be wasteful, because it will bring useless data, while if the prefetch is issued too late it will fail to hide the memory latency completely. Secondly, prefetch transactions generally have a higher latency than forwarding transactions. This is because they involve three steps: from consumer processor to directory to find out the location of the data, from directory to producer processor, and from producer processor to consumer processor.[1] Finally, if several consumers want the data, all of them have to execute prefetch instructions. We note in passing that forwarding also has the counterpart to prefetching in exclusive mode [13]. In this case, the producer forwards the data in exclusive mode and invalidates itself.

The compiler support required for data forwarding is more sophisticated than for prefetching. Indeed, if a processor issues prefetches to eliminate conflict misses only, then the compiler analysis can focus exclusively on that processor's code. Furthermore, while in both prefetching for coherence misses and forwarding the compiler needs to analyze the code executed by different processors, prefetching requires a simpler analysis. Indeed, for prefetching, the consumer processor does not need to know the identity of the producer processor, while for forwarding the producer processor needs to know the identities of the consumer processors.

Both techniques require roughly comparable hardware support. A popular implementation of software-based prefetching requires a prefetch instruction that loads data without blocking and takes no exceptions, lock-up free caches to handle multiple outstanding prefetches, and a buffer that keeps an entry for each pending prefetch until the prefetch transaction is completed. Forwarding, on the other hand, requires a write-and-forward instruction that writes the cache through, lock-up free caches, a deeper write buffer that keeps an entry for each pending forward until the forward transaction is completed, and the ability for a cache to accept data that it has not requested.

1. It is possible, however, to modify the prefetch instruction to take the processor number as an argument and, therefore, send the request directly to the producer. In this case, prefetching the data would involve only two hops. This, however, would create races as indicated in our discussion on forwarding unless expensive hardware changes are made.

Finally, both techniques have comparable amounts of overhead in the form of extra instructions executed. These instructions are necessary to compute the destination processor in forwarding and the address of the data in prefetching. Note, however, that in forwarding we can send data to several consumers with one single instruction, while in prefetching the same behavior would require instructions in all consumers. Similarly, both schemes can cause cache conflicts by bringing too much data to the consumer cache before it is actually used. This problem is less acute in prefetching because the compiler can control the time when the prefetch is issued. In the basic form of forwarding, however, the forward is always issued when the data is written.

The previous discussion is summarized in Table 1. Overall, while the compiler support required for forwarding is more complicated, forwarding can be more effective in coping with coherence misses. Recently, Poulsen and Yew [20] made a quantitative comparison between data prefetching and data forwarding. They found that each technique was effective for application codes with different characteristics. Prefetching was better for codes with higher conflict miss rates, while forwarding achieved better performance for codes with more communication-related misses. Forwarding also performed well in codes where it was difficult to overlap communication and computation. In this paper, however, we focus on data forwarding only.

## 3 A FRAMEWORK FOR FORWARDING

After the previous description of data forwarding, we now describe a framework for a compiler algorithm to insert data forwarding instructions in the code. We focus on code that exploits loop-level parallelism with doall constructs. Therefore, we will forward from doall or serial sections to other doall or serial sections. In our analysis, we focus only on array accesses that are indexed by affine functions of the loop indices and constants. Even though this limitation is not a requirement of data forwarding, these functions are usually the only ones that can be handled well by current compiler technology. We consider three static scheduling policies for parallel loop iterations: round-robin, chunk, and block scheduling (Section 3.1). We do not consider dynamic scheduling because it is impossible to determine the mapping between processors and iterations at compile time. We divide the analysis in two parts. In the first part, described in Section 3.1, we identify the producer write and consumer read pairs and compute the forwarding expression. In the second part, described in Section 3.2, we prune some of these forwards to avoid cache pollution. We consider each part in turn.

### 3.1 Computing the Forwarding Expression

The first part of the framework for data forwarding is to identify the producer write and consumer read pairs and compute the forwarding expression. Clearly, in the presence of procedure calls and aliasing, the compiler analysis can be complex. Such analysis needs to be handled by data dependence algorithms between loops. Discussion of data dependence algorithms [3] is beyond the scope of this paper. Instead, we will assume that we have already identified the producer and consumer loops. We now have to compute the forwarding expression in the producer loop. In the following, we will proceed with an example of how to compute the forwarding expression. For simplicity, the example will use accesses to unidimensional arrays; the expressions for multidimensional arrays can be derived similarly.

Consider the producer and consumer doall loops of Fig. 2. We proceed in three steps. First, we determine, for each iteration in the producer loop, what iteration in the consumer loop consumes its data. Then, we determine, for each iteration in the consumer loop, what processor executes it. With these two expressions, we know the consumer processor for each iteration in the producer loop and therefore can generate the forwarding expression. Finally, we need to incorporate the consumer loop bounds.

```
DOALL I = 3, 16
      A(2I+1) = ...
ENDDOALL

DOALL I = 0, 9
      ...  = A(I+2)
ENDDOALL
```

Fig. 2. Simple producer and consumer doall loops.

If we denote the producer and consumer loop indices by $I^P$ and $I^C$, respectively, the producer and consumer iterations that access the same array entry are related by:

$$2I^P + 1 = I^C + 2 \quad \text{or} \quad I^C = 2I^P - 1 \tag{1}$$

where, from the loop bounds:

$$3 \le I^P \le 16 \quad \text{and} \quad 0 \le I^C \le 9 \tag{2}$$

Equation (1) means that the array element produced by iteration $I^P$ in the producer loop will be consumed by iteration $2I^P - 1$ in the consumer loop.

To determine which processor executes each iteration in the consumer loop, we need to know the scheduling policy used. Let $K$ be the number of iterations in the consumer loop, $P$ the number of processors, and $P(i)$ the processor executing iteration $i$. We assume that the loop index is normalized to start at 0 and has step 1 (if the loop index is not normalized, we subtract its lower bound from $i$ and divide by its stride before applying the formulas below). We also assume that processor numbers start at 0. The expressions of the processors executing the consumer loop iterations for different scheduling policies are:

1. **Round-Robin Scheduling**(Fig. 3a):
$$P(i) = i \bmod P$$
2. **Chunk Scheduling** (Fig. 3b). Each processor gets a contiguous chunk of iterations while distributing the iterations as evenly as possible. If $q$ and $r$ are the quotient and the remainder respectively of $K$ divided by $P$, we have:
$$P(i) = \begin{cases} \left\lfloor \dfrac{i}{\lceil K/P \rceil} \right\rfloor & \text{if } r = 0 \text{ or } i < r(q+1) \\ \left\lfloor \dfrac{i-r}{\lceil K/P \rceil} \right\rfloor & \text{otherwise} \end{cases}$$
3. **Block Scheduling** (Fig. 3c). This is like chunk scheduling except that, when the number of iterations is not a multiple of the number of processors, we try to fill up the first $P-1$ processors. This produces a simpler expression:
$$P(i) = \left\lfloor \frac{i}{\lceil K/P \rceil} \right\rfloor$$

| P0 | P1 | P2 | P3 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | | |

(a) Round-robin

| P0 | P1 | P2 | P3 |
|---|---|---|---|
| 0 | 3 | 6 | 8 |
| 1 | 4 | 7 | 9 |
| 2 | 5 | | |

(b) Chunk

| P0 | P1 | P2 | P3 |
|---|---|---|---|
| 0 | 3 | 6 | 9 |
| 1 | 4 | 7 | |
| 2 | 5 | 8 | |

(c) Block

Fig. 3. Scheduling algorithms for 10 iterations executed by four processors.

In our experiments, we use block scheduling. If, for this example where $K = 10$ we use four processors, we have:

$$P(i) = \left\lfloor \frac{i}{3} \right\rfloor$$

Therefore, from (1), an array element produced in iteration $I^P$ of the producer loop will be consumed by processor $\left\lfloor \frac{2I^P - 1}{3} \right\rfloor$ in the consumer loop. Therefore, in Fig. 2, we will replace $A(2I+1)$ by $FW(A(2I+1), (2I-1)/3)$, where $FW$ is a write-and-forward macro that takes two arguments: the address to be written to and the ID(s) of the processor(s) that will receive the forward.

If we simply placed the write-and-forward as it is, however, we would be sending several unnecessary forwards. This is because some of the elements produced are not consumed by any iteration in the consumer loop. If we want to eliminate useless forwards, we can examine the bounds of the consumer loop and make sure that all the forwarded data is consumed. From (1) and (2), we have:

$$0 \le 2I^P - 1 \le 9 \quad \text{or} \quad 1/2 \le I^P \le 5$$

Looking at the second expression, it is clear that the leftmost inequality is always satisfied. The rightmost one is not. Therefore, the later needs to be enforced with an *if*

```
DOALL I = 3, 16
      IF (I > 5) THEN A(2I+1) = ....
      ELSE FW(A(2I+1),(2I-1)/3) = ...
ENDDOALL

DOALL I = 0, 9
      ...  = A(I+2)
ENDDOALL
```

Fig. 4. Resulting code. The arguments of the write-and-forward macro are the location to be written to and the ID(s) of the processor(s) that will receive the forward.

```
C  producer                          C  modified producer
   DOALL J=1,NUM                        DOALL J=1,NUM
      VT(J,1:NUM) = ...                    FW(VT(J,1:NUM), {0, 1, ..., 31}) = ...
   ENDDOALL                             ENDDOALL

C  consumer
   DOALL I = 1, NRS
      DO J = 1, NUM
         DO K = 1, J
            ...  = VT(K,1:NUM)
         ENDDO
      ENDDO
   ENDDOALL
                    (a)                                  (b)
```

Fig. 5. Example of a forward broadcast. Code (a) is the original code, while code (b) is the instrumented producer code.

statement enclosing the forward. The resulting code is shown in Fig. 4. To eliminate the *if* statement, we could split the producer loop into one that goes from 3 to 5 and another one that goes from 6 to 16.

### 3.1.1 Examples of Forwarding Expressions

Before finishing this section, we show three examples of the resulting forwarding expressions for loops in parallelized versions of the Perfect Club codes [5], [6]. The examples show a forward broadcast, a forward from a serial to a parallel section, and a forward between two parallel sections. They refer to a machine with 32 processors.

**Forward Broadcast.** A simplified version of a pair of producer-consumer loops in *TRFD* [6] where broadcast is required is shown in Fig. 5a. In the figure, the notation *X(1:NUM) = Y(1:NUM)* is short for a loop that assigns each of the *NUM* elements of *Y* to its corresponding elements of *X*. In the example, note that array section *VT(J, 1:NUM)* is produced by processor *J* and consumed by *NRS* processors. After range propagation, we determine that $NRS \geq 32$. As a result, the inner loop of the consumer loop is executed by all 32 processors and *VT* needs to be broadcast to all processors. The resulting producer loop code is shown in Fig. 5b. When the FW macro gets translated into assembly, the only instruction overhead is one assembly instruction that loads the mask register. The register is loaded with `0xffffffff`, meaning that all processors will receive the data. Note that this assembly instruction can even be moved out of the loop, thereby making the overhead practically negligible. Of course, the producer processor is forwarding to itself too. This redundant forward, however, can be easily eliminated in hardware.

**Forward from Serial to Parallel Section.** One important parallel loop in *QCD* [6] consumes data produced in a serial section. The serial section and the loop are shown in Fig. 6a. Constant propagation is used to determine all the constants that appear in the loop bounds. Using our approach from Section 3.1 and extending the formulas to support two index

```
C  producer loop:  it is filling an
C  array with 0's, except that for
C  every group of 18 elements,
C  location 1, 9, and 17 are set to 1
   DO I = 0, 1023
      DO J = 1, 17, 8
         U1(18*I+J) = 1.0
      ENDDO
      DO J = 18*I+2, 18*I+8
         U1(J) = 0.0
      ENDDO
      DO J = 18*I+10, 18*I+16
         U1(J) = 0.0
      ENDDO
      U1(18*I+18) = 0.0
   ENDDO

C  consumer loop:  each consumer is
C  consuming chunks of 72 elements
   DOALL I = 0, 255
      DO K = 0, 71
         ...  = U1(72*I+K+1)
      ENDDO
   ENDDOALL
                    (a)
```

```
C  modified producer
   DO I = 0, 1023
      DO J = 1, 17, 8
         FW(U1(18*I+J), ⌊(18*I+J-1)/(72*8)⌋) = 1.0
      ENDDO
      DO J = 18*I+2, 18*I+8
         FW(U1(J), ⌊(J-1)/(72*8)⌋) = 0.0
      ENDDO
      DO J = 18*I+10, 18*I+16
         FW(U1(J), ⌊(J-1)/(72*8)⌋) = 0.0
      ENDDO
      FW(U1(18*I+18), ⌊(18*I+18-1)/(72*8)⌋) = 0.0
   ENDDO
                    (b)
```

Fig. 6. Example of forwarding from a serial to a parallel section. Code (a) is the original code, while code (b) is the instrumented code for the producer.

```
C  producer:  each processor produces 8
C  chunks of 72 consecutive elements
   DOALL I = 0, 255
      DO J = 72*I + 1, 72*I+72
         U1(J) = ...
      ENDDO
   ENDDOALL

C  consumer:  only half of the processors
C  have work.  Each processor reads 16
C  groups of elements from a subarray
C  of 1152 elements
   DOALL I = 0, 15
      DO J = 0, 15
         DO K = 54, 71
            ...  = U1(72*16*I+72*J+K)
         ENDDO
      ENDDO
   ENDDOALL
                    (a)
```

```
C  modified producer
   DOALL I = 0, 255
      DO J = 72*I + 1, 72*I+72
         IF (J mod 72 ≥ 54) THEN
            FW(U1(J), ⌊J/(72*16)⌋) = ...
         ELSE U1(J) = ...
         ENDIF
      ENDDO
   ENDDOALL
                    (b)
```

Fig. 7. Example of forwarding from a parallel to a parallel section. Code (a) is the original code, while code (b) is the instrumented code for the producer.

variables $I^P$, $J^P$ and $I^C$, $K^C$, we produce the code in Fig. 6b. Note that, in Fig. 6b, the expressions in the array subscripts and the numerator of the expressions to be loaded in the mask register are very similar. Therefore, to set up the mask register, we only need the following three assembly instructions: a subtraction by one, a division, and a shift that, at the same time, loads the mask register.

**Forward from Parallel to Parallel Section.** This common case is illustrated by two parallel loops from *QCD* shown in Fig. 7a. This time, when we apply our analysis, we have to insert a branch (Fig. 7b). In this case, four to five assembly instructions are required: two to three for the *mod*, test, and branch plus two for the division and shift and load to the mask register.

Overall, each forwarding expression requires a few (one to five) assembly instructions. The real overhead, however, is smaller because the mask register can often be loaded outside the loop and reused in several forwards. In general, the complexity of the forwarding expressions in these examples is representative of most loops in the Perfect Club codes. Note, however, that the forwarding analysis needs to be integrated with other compiler passes that perform induction variable elimination, constant propagation, and range and symbolic analysis.

### 3.2 Pruning Forwards

If we forwarded all the data that can be forwarded, we might cause conflicts in the caches of the consumer processors and degrade overall performance. For this reason, in the second part of our framework, we selectively prune some of the forwards identified in the first part if we suspect that they can cause conflicts.

To decide whether or not to prune a forward, we need to estimate the number of different cache lines that the consumer processor will need in its cache between the arrival and the use of the forwarded datum. This value can be estimated by using locality analysis as in [11], [23] where spatial, temporal, and group locality carried by loops are identified. If such value is larger than a threshold value, the forward is pruned; otherwise the forward is left in the code. In a fully-associative cache, such threshold can be set to the size of the cache. For direct-mapped or set-associative caches, however, we need to reduce the threshold to account for cache conflicts. Mowry et al. [14] suggest a threshold approximately equal to 1/16th of the size of the cache.

To understand our approach, consider the example in Fig. 8. In the figure, processor *P1* writes to variable *a* at point *S1* and processor *P2* reads the data at *S2*. The amount of different data that processor *P2* will need in its cache sometime between *S1* and *S2* is the sum of four components:

1. Data accessed by *P2* from the time the forward arrives until the end of the parallel section (area *A*). To estimate the time when the forward arrives, we assume that loop iterations are executed in lockstep. For example, if *P1* wrote while executing its third iteration, we assume that *P2* was also in its third iteration when it received the data.
2. Data accessed by *P2* during any code executed between the two parallel sections involved (area *B*).
3. Data accessed by *P2* in the consumer loop before the use of *a* (area *C*).
4. Data forwarded to *P2* for statements after *S2* that arrives before *S2*.

To compute the forwards to prune, we proceed in two steps. First, for each producer write and consumer read pair, we add up components 1 to 3 above. The resulting value we call the *distance* between producer and consumer. If the distance is larger than the threshold, we discard the forward. In the second step, for each of the remaining forwards, we examine its consumer point *S2*. We compute the amount of data being forwarded to statements after *S2* such that the statements are at a distance of *S2* less than the threshold. Clearly, this amount of forwarded data is an upper bound of the value of component 4 above. If this value plus components 1 to 3 above exceeds the threshold, the forward is discarded too. This second step proceeds from the bottom of the code to the top of the code. In this way, when we discard a forward we do not have to recompute previous computations.

## 4 Experimental Methodology

Any implementation of an algorithm for data forwarding requires a significant effort to implement. Indeed, the forwarding analysis needs to be integrated with other compiler passes that perform interprocedural analysis, induction variable elimination, constant propagation, and range and symbolic analysis. This is necessary because, to determine the producer-consumer pairs, we need precise information about aliasing, loop bounds, and expressions in the array indices. Since we have not yet finished our implementation, we will only perform an approximate estimation of the performance impact of data forwarding. This estimation will be based on using trace-based profiling information that detects some of the producer-consumer pairs.

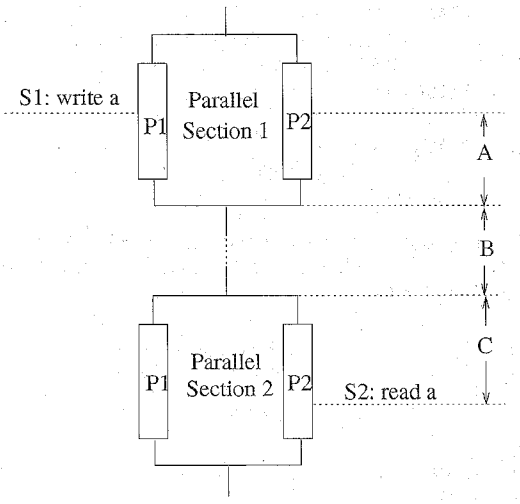



Fig. 8. Computing the amount of data that the consumer processor needs to keep in its cache.

Our evaluation proceeds in two passes. In a first execution of the applications, we collect data about the sharing patterns present in the applications. Based on a trace, we measure what data will be read by what processor and what processor will produce it. This step corresponds to a compiler-analysis of the code where the compiler has the ability to detect some of the true sharing accesses. We will explore different levels of compiler sophistication. Next, we feed this data to a second execution of the code. In this execution, whenever we have a write that was detected as forwardable in the previous pass, we replace it by a write-and-forward. Before presenting the results, we now discuss the simulation system and the applications used.

### 4.1 Simulation System

For our simulations, we use the EPG-sim execution-driven simulator [18]. We start by compiling our applications with the Parafrase2 compiler [16]. For each array access in the program, the compiler introduces a call to the simulator. Each call takes as arguments the address of the data accessed, the processor ID, the type of access, and the time-stamp. It is assumed that each basic operation like an arithmetic operation, a test, or a loop index increment takes one cycle to execute. The applications are then linked with the EPG-sim architecture simulator. Finally, the applications are executed. At that time, the applications call the simulator, which simulates the memory accesses.

The architecture modeled is a 32-processor cache-coherent UMA shared-memory multiprocessor. The machine uses a base invalidate protocol. Processors have private caches and are connected to memory via an Omega

TABLE 2
APPLICATIONS STUDIED

| Application | What It Does | Lines (Thous.) | Refs. (Mill.) |
|---|---|---|---|
| *TRFD* | Solves 2-electron integrals | 0.3 | 63.7 |
| *QCD* | Solves lattice gauge problems | 2.6 | 25.0 |
| *DYFESM* | Analysis of symmetric anisotropic structures | 8.2 | 33.1 |
| *FLO52* | Analysis of 2D transonic airflow | 4.0 | 29.3 |
| *ARC2D* | Solves a 2D fluid dynamics problem | 5.5 | 32.1 |

network. Each network link is 64-bit wide and cycles at the same speed as the processors. Without contention, a cache read miss takes 100 cycles for data clean in memory and 200 cycles for data dirty in another processor. Since the applications that we run on the simulator are necessarily small (Section 4.2), we simulate small caches, ranging from 16 to 256 Kbytes in size. To simplify the simulations, we use only one level of caches. To reduce the conflicts in these small, simple cache hierarchy, we make the caches 4-way set associative and use 8-byte lines. Since we use the release consistency model, write misses do not stall the processors. We accurately model all the contention in the system. For the network, we use the analytical delay model for indirect multistage networks presented in [10]. In our evaluation, we assume that each forward requires four assembly instructions in the producer code.

### 4.2 Applications

We trace the parallel versions of the five Perfect Club codes [5] presented in Table 2. These versions run with 32 processors and were parallelized using a parallelizing compiler and later by hand [6]. Since the codes take a long time to run, we reduced their time requirements while preserving their parallelism and reference behavior. We did this by reducing the number of iterations rather than the data set sizes where possible. Table 2 shows the number of lines of code and the number of array references simulated in each application.

## 5 Evaluation of Data Forwarding

In this section, we evaluate the performance impact of data forwarding. We start by analyzing the application characteristics that determine the effectiveness of forwarding (Section 5.1). Then, we present different levels of support for data forwarding (Section 5.2). Finally, we measure the impact of forwarding in the execution time of the simulated architecture in two scenarios: when the caches are larger than the working set size of the applications (Section 5.3) and when they are smaller (Section 5.4). To determine the working set size of the applications, we plot the miss rate as a function of the cache size. The cache size at the knee of the resulting curve is approximately equal to the working set size [17]. In our experiments, we choose a size of 256 Kbytes for a cache larger than the working set size of the applications. For a cache smaller than the working set size of the applications, we choose 16 Kbytes for *TRFD*, *QCD*, and *DYFESM*, and 64 Kbytes for *FLO52* and *ARC2D*.

### 5.1 Impact of the Application Characteristics

The potential of forwarding is determined by several application characteristics, namely fraction of reads that consume data produced by a write from another processor (consumer reads), distance between the consumer reads and their producer writes, and distribution of the number of consumer reads per producer write. We consider each characteristic in turn.

#### 5.1.1 Fraction of Consumer Reads

Data forwarding can potentially eliminate the misses on reads that consume data produced by a write from another processor. These reads we call consumer reads. A consumer read is, therefore, the first read issued by a processor to a datum that was written by a different processor. Fig. 9 shows the percentage of reads that are consumer reads in each application. From the figure, we see that, in *TRFD* and *QCD*, only about 2% of the reads are consumer reads. However, for *DYFESM*, *FLO52*, and *ARC2D*, this fraction is significantly larger, namely between 8% and 23%. Note that, without forwarding and with single-word cache lines, these reads are guaranteed to cause misses. With multiple-word cache lines, some of these accesses do not cause misses. For small caches, these misses may be a minority, since conflict misses may dominate. However, as the cache grows in size, these misses will tend to dominate. In particular, for an infinite cache with single-word lines, consumer reads would be the only cause of misses other than cold-start misses. From this data, we expect to see more benefits from forwarding in *FLO52*, *ARC2D*, and *DYFESM* than in *TRFD* and *QCD*.

#### 5.1.2 Distance Between Producer Writes and Consumer Reads

With a limited cache size, it is important to prevent data forwarding from causing cache conflicts. Intuitively, forwarding data that will be used soon is less likely to create cache conflicts than forwarding data that will not be used until much later. In Section 3.2, we defined the distance between a producer write and a consumer read as the amount of different data that the consumer processor accesses between the two points. Fig. 10 plots the distribution of the distance between

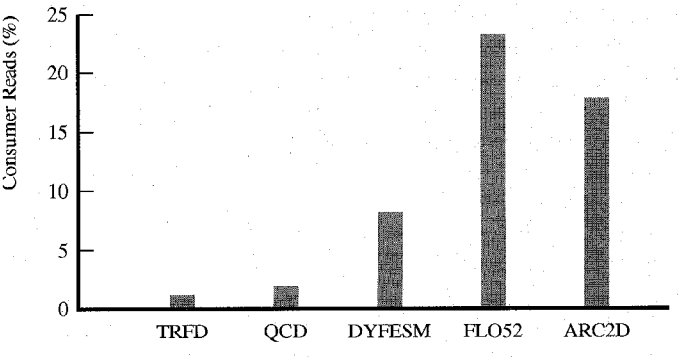



Fig. 9. Fraction of reads that are consumer reads.

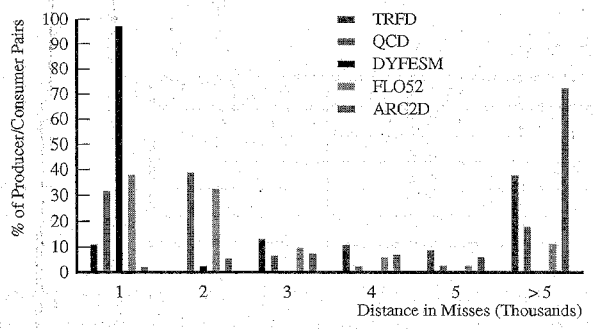



Fig. 10. Distribution of distances between producer writes and consumer reads. Distances are measured in misses in the simulated architecture with 256-Kbyte caches and 8-byte cache lines.

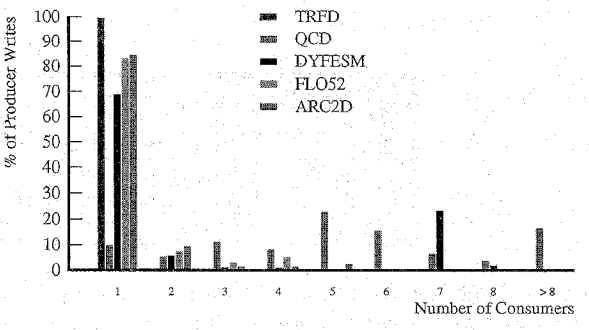



Fig. 11. Number of consumer reads per producer write.

producer and consumer accesses for the applications. For simplicity, we approximate the distance with the number of misses that the consumer cache suffers between the producer and consumer points. The misses measured include both read and write misses. Using misses does not give the whole picture because the number of misses does not tell us how much data already in the cache is being used. Intuitively, however, the higher the number of misses between producer and consumer points, the less attractive that forwarding looks. The figure corresponds to the simulated architecture with 256-Kbyte caches.

The figure indicates that most consumer reads are close to their producer writes. This is specially true for *DYFESM*, where nearly 100% of the consumer reads are less than 1,000 misses away from their producer writes. Two applications, however, exhibit a somewhat bad behavior, namely *TRFD* and *ARC2D*. In these applications, 37% and 72% of the consumer reads respectively have a distance larger than 5,000 misses. Overall, therefore, we expect that in *DYFESM*, *FLO52*, and *QCD*, forwarded data will cause few conflicts, while in *TRFD* and *ARC2D*, forwarding will have a higher chance of causing conflicts, specially for small caches.

### 5.1.3 Number of Consumer Reads per Producer Write

Finally, the number of consumer reads per producer write tells us what kind of forwarding instruction is required, namely point to point forwarding, multicasting, or broadcasting. Fig. 11 presents the distribution of such number for the applications. Clearly, the number of consumers per write can range from one to 31. As the figure shows, however, in all cases except *QCD*, the number of consumers is usually one. This implies that a simpler point to point forwarding hardware will be enough in most cases. For the cases with more than one consumer, multiple forwarding instructions could have been inserted. We do not do this in our simulations.

To summarize, we have seen that, while the fraction of consumer reads varies significantly across applications, producer writes tend to be followed soon by their, often single, consumer reads. Among the applications, *FLO52*, *DYFESM*, and *ARC2D* have a higher potential because they have a large fraction of consumer reads. In *ARC2D*, however, forwarding may be less effective because of the long distances involved.

## 5.2 Forwarding Strategies and Their Potential

To gain insight into the effectiveness of forwarding, we will compare five different levels of compiler support for forwarding:

- *Base*: Forwarding is not supported.
- *Analyz*: Forwarding is supported only for the arrays that are indexed by affine functions of loop indices and constants. These expressions we call compiler-analyzable expressions. Note that this is a fairly conservative definition of compiler-analyzable: We neither include loop-invariant variables nor perform constant propagation, range analysis or symbolic analysis. A real compiler can surely analyze more expressions.
- *All*: Forwarding is supported for all the producer-consumer pairs of array references. This is an optimistic scenario.
- *Local*: Like *All* except that forwarding is not supported when there is a procedure call or return between the producer and consumer accesses. This implementation would not require interprocedural analysis.
- *Dis8/Dis32/Dis128*: Like *All* except that forwarding is not supported when the distance between the producer and consumer accesses is larger than a certain threshold. The threshold is reached when the consumer cache has accessed as much data as one half the size of the cache via misses. Since we perform experiments for 16-Kbyte, 64-Kbyte, and 256-Kbyte caches, this corresponds to 8 Kbytes of data (*Dis8*), 32 Kbytes of data (*Dis32*), and 128 Kbytes of data (*Dis128*), respectively. This implementation corresponds to a more complex compiler support that involves pruning the forwards.

To understand the potential of these different schemes, we examine their *coverage* of producer-consumer pairs. A scheme covers a producer-consumer pair if it generates a forward between the two. Note that covering a producer-consumer pair does not mean that, at run time, the corresponding consumer read will not miss. The read may still miss because of cache conflicts, false sharing, or because the forwarded data does not reach the consumer on time. The amount of producer-consumer coverage, however, is a good indicator of how well a given forward scheme can potentially do.

Fig. 12 shows the number of producer-consumer pairs covered by *Analyz*, *Local*, *Dis8/Dis32* (small caches), and *Dis128* (large caches), relative to the number covered by the *All* scheme. For each scheme, the figure shows the five applications considered. Recall that for a cache smaller than

the working set of the application we chose 16 Kbytes for *TRFD*, *QCD*, and *DYFESM* and 64 Kbytes for *FLO52* and *ARC2D*. Hence, in the *Dis8/Dis32* bars, *Dis8* applies to the *TRFD*, *QCD*, and *DYFESM*, while *Dis32* applies to *FLO52* and *ARC2D*.

Focusing first on *Analyz*, we see that this scheme only covers a large fraction of the producer-consumer pairs in *QCD* and *DYFESM*. In the other three applications, the coverage is very small. Therefore, this scheme will speed up two applications at most. It is interesting, however, to see why the scheme fails and what a real compiler would need to do to improve on it. In *DYFESM* and *TRFD*, not much more can be done, since most of the remaining producer-consumer pairs involve array accesses that have subscripted subscripts. In *FLO52*, *ARC2D*, and *QCD*, however, the large majority of the producer-consumer pairs not covered involve very simple array expressions. They are not affine functions of loop indices and constants for two reasons. The first one is that, in an effort to reduce the overhead of scheduling loop iterations, the compiler or programmer has hard-coded the scheduling algorithm in each parallel loop. Consequently, at the start of each parallel loop, each processor computes the expression of the set of iterations that it has to work on. The resulting function of loop indices is assigned to temporary variables. Then, the rest of the loop uses affine functions of these temporary variables to index the arrays. The resulting subscript expressions, therefore, do not qualify for *Analyz*. Nevertheless, a good compiler would detect this situation and replace each use of the temporary variables in subscripts by the corresponding functions of the indices. This will create many more opportunities to find affine functions of index variables and constants.

The second reason is that, often, the subscript expression includes a variable. Therefore, the *Analyz* scheme would not handle it. In nearly all cases, however, the variable is loop-invariant and a compiler could find it out. In that case, the variable can be treated like a constant. Furthermore, a constant propagation pass often uncovers that the variable is indeed a constant. To summarize, if we use an aggressive compiler, both issues can be successfully solved and 80-90% of the producer-consumer pairs in *QCD*, *ARC2D*, and *FLO52* can be covered. We can say, therefore, that the coverage that a good compiler can provide is probably closer to *All* than to *Analyz* for these three applications.

Examining now the *Local* bars, we note that the coverage of the *Local* scheme is low. Indeed, while *FLO52* and *ARC2D* have a 25-50% coverage, the rest of the applications have a null coverage. It is clear, therefore, that any forwarding algorithm must include interprocedural analysis.

Finally, the last two bars show the effect of a pruning algorithm. The *Dis128* bars correspond to a distance of 128 Kbytes, or 16,000 misses, while the *Dis8* and *Dis32* bars correspond to 1,000 and 4,000 misses, respectively. The chart shows that, under *Dis128*, all producer-consumer pairs are covered in all applications except for *ARC2D*. This is consistent with the data in Fig. 10. That figure showed that, while most of the applications had relative short producer-consumer distances, *ARC2D* had over 70% of the producer-consumer pairs separated more than 5,000 misses. Overall, therefore, we expect that the performance of *All* will be similar to that of *Dis128* for most applications. In *Dis8/Dis32*, however, the coverage varies among applications. In agreement with Fig. 10, *TRFD*, *QCD*, and *ARC2D* have a low coverage while the rest of the applications have a high one. The impact of the pruning will depend on the relative weight of coherence misses versus conflict misses.

## 5.3 Forwarding in Caches Larger than the Working Set

We now measure the impact of forwarding on the execution time for caches larger than the working set of the applications. We use 256-Kbyte caches. This case is a favorable one for forwarding: Forwarding is less likely to induce conflict misses. In the following, we first examine how successfully the forwards are being used and then the actual execution time change.

### 5.3.1 Effectiveness of the Forwards

While the data in Fig. 12 give a first insight into the relative impact of the different schemes, it does not give the complete picture. The reason is that the data may reach the consumer cache and get replaced due to cache conflicts or get invalidated due to false sharing before the consumer processor uses it. In our applications, it is not possible that the forwards arrive at the consumer cache after the consumer read. This is because there is always a synchronization step between the producer and the consumer accesses and processors stall at synchronization points until all pending requests, including forwards, have been completed. The impact of cache conflicts and false sharing is shown in Fig. 13. The figure shows the fraction of the forwarded cache lines that are used by the consumer processor before they are displaced or invalidated from the consumer cache. Note

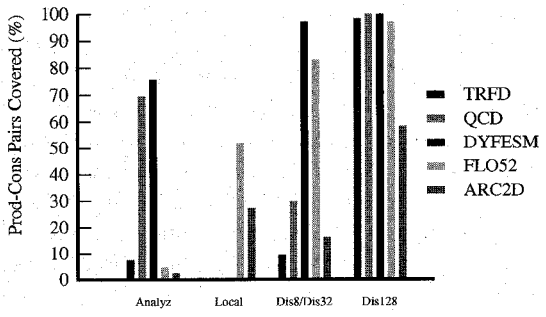



Fig. 12. Fraction of producer-consumer pairs covered by different schemes relative to the *All* scheme.

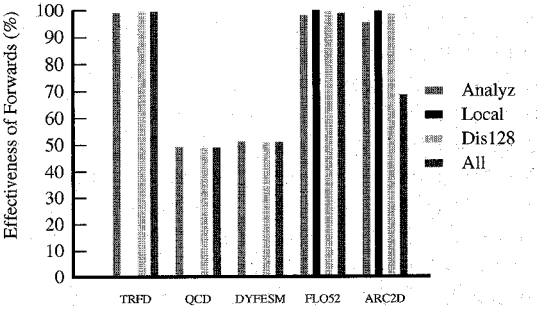



Fig. 13. Effectiveness of the forwards measured by the fraction of the forwarded cache lines that are used by the consumer processor before they are displaced or invalidated from the cache of the consumer processor. The data corresponds to 256-Kbyte caches.

that the displacement may be caused by the arrival of another forward. The figure is organized differently than Fig. 12: The bars are grouped by application, not by forwarding scheme. For *TRFD*, *QCD*, and *DYFESM*, we do not show the bar for *Local*. The reason is that the number of forwards is very small.

The figure shows that, for a given application, all the forwarding schemes tend to have a similar effectiveness. For *TRFD*, *FLO52*, and *ARC2D*, the effectiveness is close to 100%. This indicates that, for these large caches, most of the forwards reaching the consumer caches in *TRFD*, *FLO52*, and *ARC2D* are used before being displaced or invalidated. The only exception is *All* for *ARC2D*, where the bar drops below 70%. This is due to the large size of the working set of the application and the large distance between some producer-consumer pairs.

In *QCD* and *DYFESM*, however, the effectiveness is only 50%. The reason for this low value is that, in a very short interval, a processor writes to two consecutive words that share the same cache line. This generates two forward messages closely spaced in time that, in addition, go to the same consumer processor. Consequently, the line updated by the first forward is immediately updated by the second forward. However, the consumer processor hides all latency because it accesses the two words after the second forward. In sum, therefore, the real effectiveness of the forwards is twice that shown, namely, about 100%.

Overall, we see that, for the five applications considered, if caches are larger than the working set, then the effectiveness of forwarding is very close to 100% for all schemes. Furthermore, the particular behavior of *QCD* and *DYFESM* suggests that some applications will benefit from the combining optimization suggested in Section 2.1. For these applications, the network traffic due to forwarding would be cut by half.

### 5.3.2 Impact on the Execution Time

After the previous analysis of the effectiveness of forwarding, we now consider the impact on the execution time. Fig. 14 presents the miss rates and execution times of the five applications. In each plot, the horizontal axis shows the five levels of support for forwarding, while the vertical axis presents the miss rate (leftmost plots) or the execution time (rightmost plots). Miss rates are broken down into cold (or start-up), conflict, and true and false sharing. Miss rates include both read and write misses, although processors do not stall on write misses. Execution times are normalized to *Base* for each application and are broken down into busy time, idle time, synchronization time, read miss stall time, and forwarding instruction overhead (*Inst. Over.*). Busy time is the time spent by the processors executing the instructions of the applications without missing on the cache. Idle time appears because we take the average of the 32 processors and, in a serial section, only one processor is busy, while the others are idle. Synchronization time is largely due to load imbalance in parallel loops.

Starting first with *Analyz*, we see from the rightmost charts that this level of support speeds up only some of the applications. Indeed, while two applications run 17-26% faster, the remaining three barely change. In these three applications (*TRFD*, *FLO52*, and *ARC2D*), the read miss stall time changes little. This is because, as it was shown in Fig. 12, little forwarding takes place. As a result, the leftmost charts show that the miss rates barely change. For the other two applications (*QCD* and *DYFESM*), forwarding decreases the read miss stall, synchronization, and idle times. The former is accomplished by reducing the cold and sharing miss rates (leftmost charts). Synchronization time is reduced indirectly by speeding-up execution: The absolute amount of load imbalance time is smaller. Finally, idle time decreases because the processor executing the serial section runs faster. Overall, *Analyz*, a conservative estimate of what a compiler can do, is not enough to provide better performance for all codes.

However, if we examine the *All* bars, we see that all applications are sped up. They run 23-81% faster. Except for *ARC2D*, the rightmost charts show that the read miss stall time practically disappears. This effect is the result of the elimination of the large majority of cold, sharing, and conflict read misses. The miss rates remaining in the leftmost charts are mostly caused by write misses. These misses are not targeted by our scheme. We also see that forwarding removes many conflict misses. This is because, as indicated in Section 2, while we count the misses as conflict misses, they hide sharing accesses that are amenable to forwarding. The reason why not all read miss stall time is removed in *ARC2D* is because some of the conflict misses do not hide sharing accesses. Overall, however, the results are encouraging: With *All*, all programs run much faster. While this level of compiler support is optimistic, we feel that advanced compilers can get closer to *All* than to *Analyz*. This is because they can analyze subscripts with loop-invariant terms and some kinds of nonaffine functions. The exceptions are *TRFD* and *DYFESM*, where many array accesses have subscripted subscripts.

From the *Local* bars in the rightmost charts, we see that local-only forwarding is not effective for the majority of the applications. Three applications show no gains, while the other two are sped up by 11 and 48%. The reason for the overall low impact is the small number of forwards that can be issued locally. This was show in Fig. 12. Therefore, it is clear that forwarding requires interprocedural analysis.

From the *Dis128* bars in the rightmost charts, we see that forwarding using a limited distance is not better than the *All* scheme. This is because, as shown in the miss rate charts, forwarding in *All* does not increase the conflict miss rate over *Base*. Therefore, cache conflicts in *All* are not caused by forwarding. Consequently, we cannot expect *Dis128* to reduce the miss rate over *All* much. Instead, the miss rates in *Dis128* are similar or slightly higher than in *All*. For *ARC2D* they are higher because *Dis128* had a low coverage (Fig. 12). As a result, for these large caches, forwarding using a limited distance is unnecessary and even undesirable, since you may miss some forwards. We also note in passing that the instruction overhead of forwarding is negligible.

To summarize, forwarding using the *All* scheme in caches larger than the working set achieves substantial

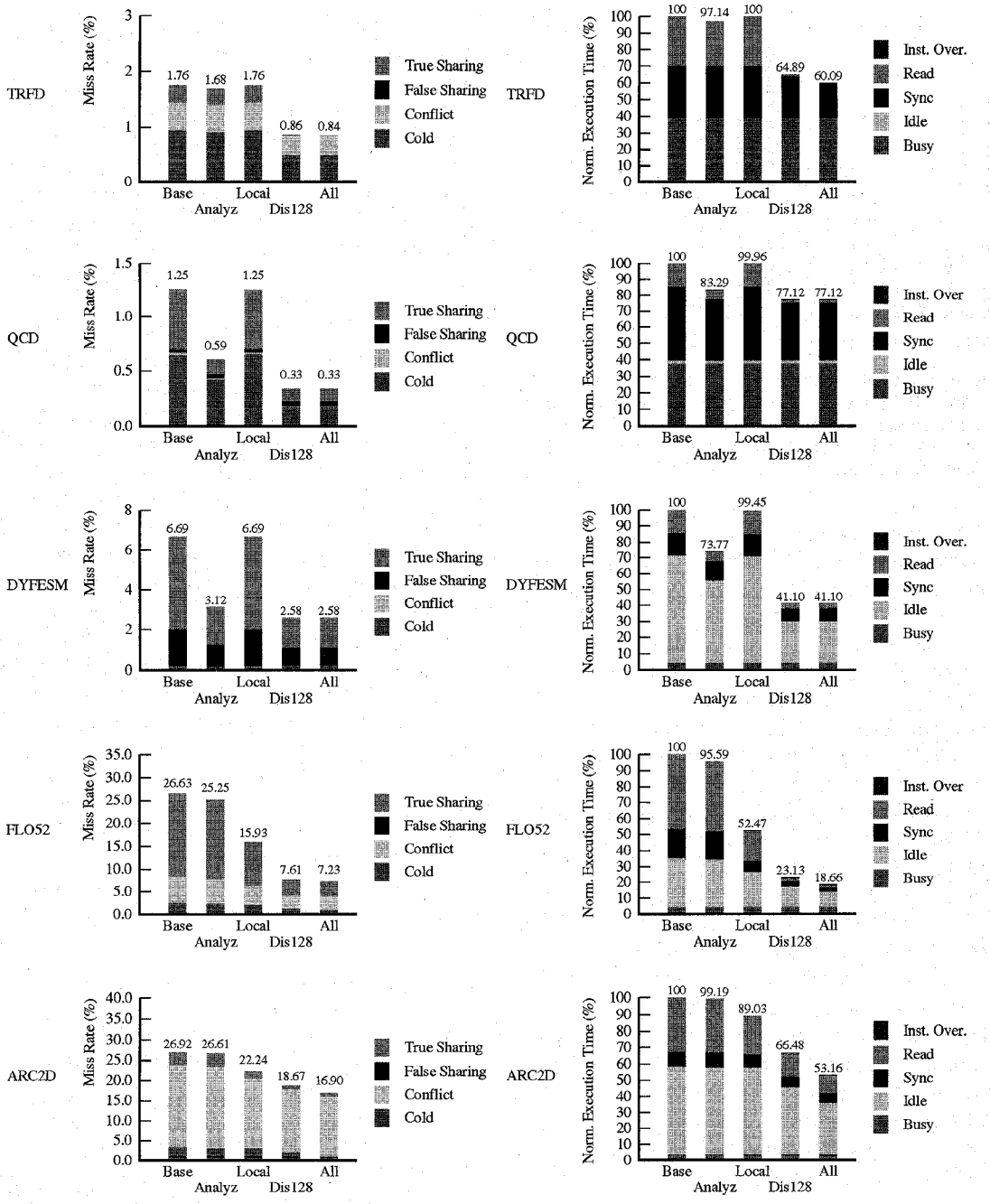



Fig. 14. Miss rates (left) and normalized execution times (right) with 256-Kbyte caches. The miss rates include both reads and writes.

reductions in execution time. While *All* is a slightly optimistic scenario, it may be more realistic than *Analyz* for all applications except *DYFESM* and *TRFD*. *Analyz* is not enough to speed up most applications. Similarly, neither local-only forwarding nor limiting the distance of forwarding in caches larger than the application's working set is attractive.

## 5.4 Forwarding in Caches Smaller than the Working Set

We now measure the impact of forwarding on the execution time for caches smaller than the working set of the applications. We chose 16-Kbyte caches for *TRFD*, *QCD*, and *DYFESM*, and 64-Kbyte caches for *FLO52* and *ARC2D*. The working set size was computed by increasing the cache size until no large reductions in cache miss rates are obtained [17]. As in the previous section, we first examine how well the forwards are being used and then the actual execution time change.

### 5.4.1 Effectiveness of the Forwards

To determine the effectiveness of the forwards in small caches, we plot in Fig. 15 the fraction of the forwarded cache lines that are used by the consumer processor before they are displaced or invalidated from the cache of the consumer processor. The figure is organized like Fig. 13.

As expected, forwards are less effective for these caches than for the larger caches measured in Fig. 13. This is seen by the generally shorter bars in Fig. 15. The lower effectiveness is due to the higher number of cache conflicts re-

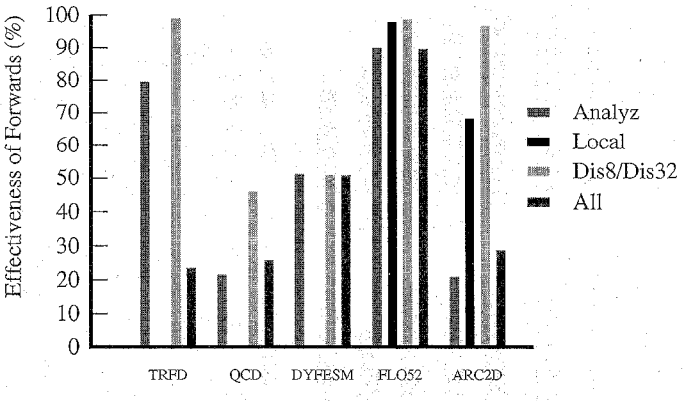



Fig. 15. Effectiveness of the forwards measured by the fraction of the forwarded cache lines that are used by the consumer processor before they are displaced or invalidated from the cache of the consumer processor. The data corresponds to small caches: 16 Kbytes for *TRFD*, *QCD*, and *DYFESM*, and 64 Kbytes for *FLO52* and *ARC2D*.

corded. Different bars, however, shrink by different degrees. For example, in *DYFESM* and, to a lesser extent, *FLO52*, the bars maintain their size. This means that the forwards suffer roughly the same number of conflicts as before. In the other three applications, however, the bars shrink, revealing an increase in the conflicts suffered by the forwards as a result of the reduced cache size.

For the three applications where the effect of conflicts is large (*TRFD*, *QCD*, and *ARC2D*), we note that the *Dis8/Dis32* bars maintain their size while the remaining bars shrink relative to them. In fact, for all applications, the forwards in *Dis8/Dis32* have a real effectiveness close to 100%. This higher effectiveness of the forwards in the *Dis8/Dis32* scheme results from the selectivity used in the scheme. This, however, does not mean that the *Dis8/Dis32* scheme will perform better than the *All* scheme, which has shorter bars. The reason is that, while the *All* scheme has low effectiveness in Fig. 15, it has a higher coverage than the *Dis8/Dis32* scheme as shown in Fig. 12. As a result, the absolute number of successful forwards tends to be higher in *All*. We note, however, that if the network bandwidth is scarce or if the application requires a lot of communication, then the higher effectiveness of the *Dis8/Dis32* scheme may give it an advantage.

### 5.4.2 Impact on the Execution Time

After the analysis of the effectiveness of forwarding, we now consider the impact of forwarding on the execution time. Fig. 16 presents the miss rates and normalized execution times of the five applications. The figure is organized like Fig. 14.

As shown in the miss rate charts, conflict misses dominate in most applications. This is because caches are smaller than the working sets. As expected, forwarding is generally unsuccessful at removing conflict misses. This is true irrespective of the forwarding scheme used. The data, however, leads to two main conclusions.

The first conclusion is that, in small caches that already suffer many conflicts without forwarding, it is remarkably hard for forwarding to increase the number of conflict misses. Indeed, consider the case of *All*, the scheme that involves the largest amount of forwarding messages. Fig. 15 shows the low effectiveness of forwarding: Most of the forwarded cache lines are replaced in the consumer cache before being used. However, even under these conditions, none of the five applications records any increase in the number of conflict misses. In fact, in *FLO52*, conflict misses even decrease for the reason described in Section 2: Conflicts hide sharing misses available for forwarding. The reason why conflict misses do not increase is because the data displaced by the unsuccessful forwards would have been displaced by other data anyway. This is intuitive from the high miss rates in *FLO52* and *ARC2D*. Overall, since conflict misses do not increase and cold and sharing misses decrease, the total number of misses decreases.

The second conclusion is that, while this reduction of misses at the expense of high traffic may successfully decrease the execution time in high-bandwidth networks like the one simulated, in lower-bandwidth networks it may be necessary to use a limited distance scheme like *Dis8/Dis32*. Indeed, in the current system, *All* does very well: Three applications speed up significantly and none suffers a slowdown. On average, *All* speeds up the five applications by 31%. It is interesting, however, to examine the *Dis8/Dis32* scheme. While the execution times of *All* are only slightly smaller than those of *Dis8/Dis32* for most of the applications, the numbers of forwards in *All* are up to 10 times larger than in *Dis8/Dis32* (2 times on average). This can be seen in Fig. 12. Therefore, much of the forward traffic in *All* is useless. The network that we use has enough bandwidth to tolerate this useless traffic. However, in lower-bandwidth networks, *Dis8/Dis32* is likely to perform better than *All*.

We also note that, as before, the *Analyz* and *Local* schemes are not attractive.

## 6 Conclusions

Long-latency memory accesses are a major source of slowdown in scalable shared-memory multiprocessors. One way to cope with these latencies is to overlap memory accesses with computation or other memory accesses via data forwarding. In this paper, we have presented a framework for a compiler algorithm for forwarding and, based on address traces, estimated the performance impact of forwarding.

This paper shows that support for forwarding in shared-memory multiprocessors promises to deliver good application speedups. An optimistic support for data forwarding (*All*) achieves large reductions in execution time. We show that five applications are sped up by an average of about 50% for large caches and 30% for small caches. For large caches, most sharing read misses and cold read misses are eliminated. For small caches, forwarding does not increase the number of conflict misses significantly. For forwarding to be effective, however, the compiler must perform interprocedural analysis, as well as symbolic analysis, range analysis, and constant propagation.

Future work will proceed in two directions. First, we will automate an algorithm for data forwarding and integrate it in a compiler. Second, we will focus on understanding the interaction between forwarding and prefetching and use both techniques at the same time.

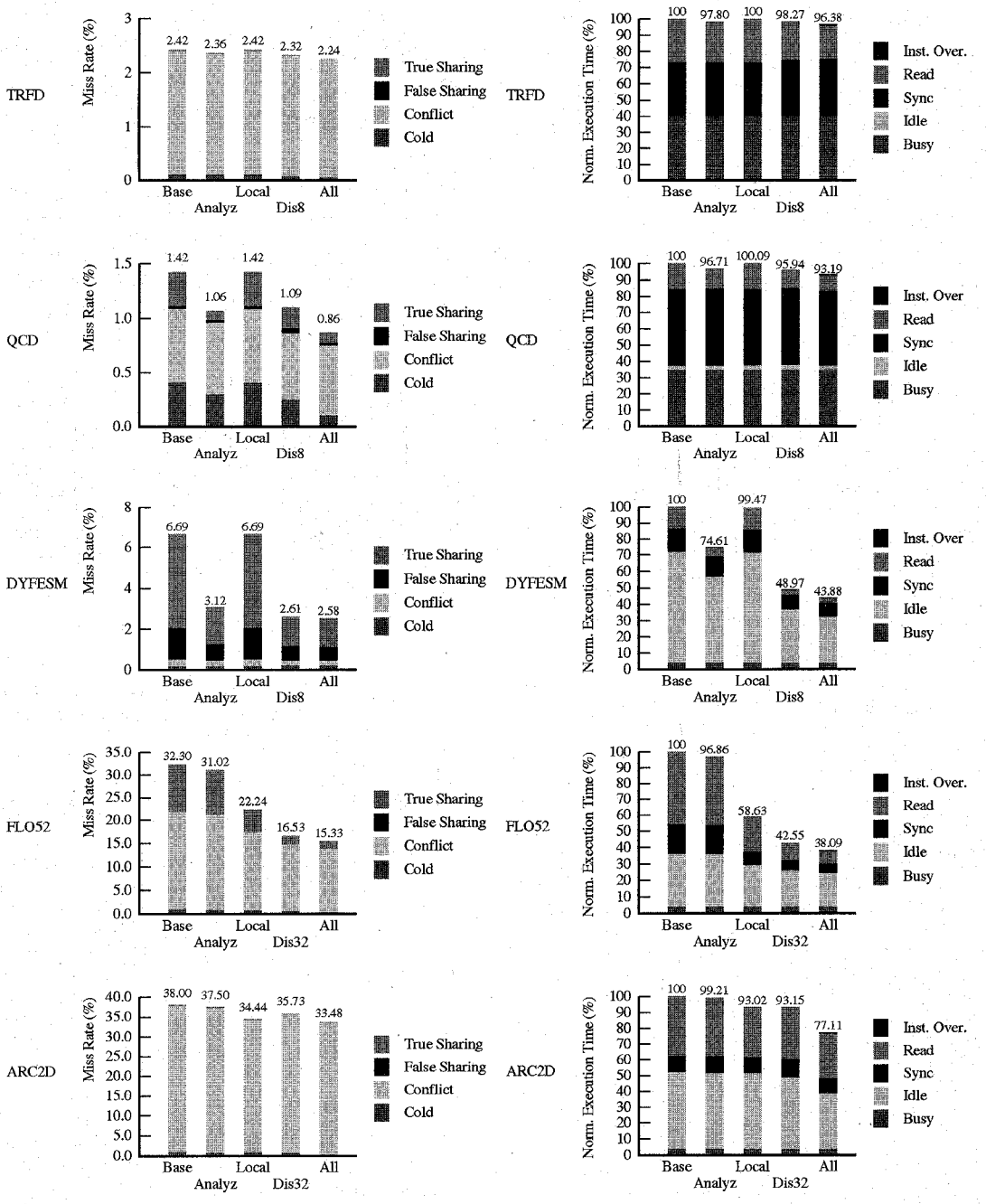



Fig. 16. Miss rates (left) and normalized execution times (right) with 16-Kbyte caches for *TRFD*, *QCD*, and *DYFESM*, and 64-Kbyte caches for *FLO52* and *ARC2D*. The miss rates include both reads and writes.

## Acknowledgments

We thank the referees for their feedback. We also thank the rest of the graduate students in the I-ACOMA multiprocessor group for their feedback.

This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098, ARPA Contract No. DABT63-95-C-0097, NASA Contract No. NAG-1-613, Intel Corporation, and by a scholarship from the Universidad Simón Bolívar and CONICIT, both of Venezuela.

## References

[1] S. Adve and M. Hill, "Weak Ordering—A New Definition," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 1-13, May 1990.

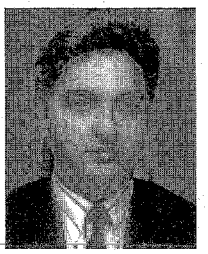
[2] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, pp. 525-539, Sept. 1992.

[3] U. Banerjee, *Dependence Analysis for Supercomputing*. Norwell, Mass.: Kluwer Academic, 1988.

[4] W. Berke, "A Cache Technique for Synchronization Variables in Highly Parallel, Shared Memory Systems," *Ultracomputer Note* 141, Dec. 1988.

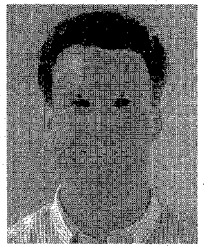
[5] M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l J. Supercomputer Applications*, vol. 3, no. 3, pp. 5-40, Fall 1989.

[6] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua, "The Cedar Fortran Project," Technical Report 1262, Center for Supercomputing Research and Development, Oct. 1992.
[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 15-26, May 1990.
[8] D. Glasco, B. Delagi, and M. Flynn, "Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors," *Proc. 27th Ann. Hawaii Int'l Conf. System Sciences*, pp. 543-545, Jan. 1994.
[9] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 64-73, Apr. 1989.
[10] C. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. Computers*, vol. 32, no. 12, pp. 1,091-1,098, Dec. 1983.
[11] M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 63-74, Apr. 1991.
[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford Dash Multiprocessor," *Computer*, pp. 63-79, Mar. 1992.
[13] T. Mowry and A. Gupta, "Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87-106, June 1991.
[14] T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, Oct. 1992.
[15] M. Ohara, "Producer-Oriented versus Consumer-Oriented Prefetching: A Comparison and Analysis of Parallel Application Programs," Technical Report CSL-TR-96-695, Computer Systems Laboratory, Stanford Univ., June 1996.
[16] C.D. Polychronopoulos et al., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *Proc. 1989 Int'l Conf. Parallel Processing*, vol. II, pp. 39-48, Aug. 1989.
[17] D.K. Poulsen, "Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared Memory Multiprocessors," PhD thesis, Univ. of Illinois at Urbana-Champaign. Also Center for Supercomputing Research and Development Report 1377, July 1994.
[18] D.K. Poulsen and P.-C. Yew, "Execution Driven Tools for Parallel Simulation of Parallel Architectures and Applications," *Proc. Supercomputing '93*, pp. 860-869, Nov. 1993.
[19] D.K. Poulsen and P.-C. Yew, "Data Prefetching and Data Forwarding in Shared Memory Multiprocessors," *Proc. 1994 Int'l Conf. Parallel Processing*, vol. II, pp. 276-280, Aug. 1994.
[20] D.K. Poulsen and P.-C. Yew, "Integrating Fine-Grained Message Passing in Cache Coherent Shared Memory Multiprocessors," *J. Parallel and Distributed Computing*, to appear, 1996.
[21] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy, "The KSR1: Experimentation and Modeling of Poststore," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, pp. 74-85, May 1993.
[22] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, pp. 12-23, June 1990.
[23] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30-44, June 1991.

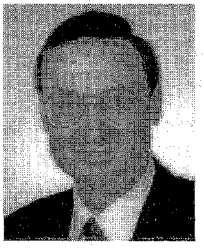


**David A. Koufaty** received the BS and MS in computer science from the Universidad Simón Bolívar in Caracas, Venezuela, in 1988 and 1991, respectively. Employed by the Universidad Simón Bolivar since 1988 as an instructor, he was awarded a scholarship in 1992 to pursue his PhD at the University of Illinois at Urbana-Champaign. Since 1993, he has been with the Center for Supercomputing Research and Development, where he is now a research assistant. His primary research interests include the design of scalable parallel architectures, cache-coherence, and parallelizing compilers. In 1994, he received the C.W. Gear Outstanding Graduate Student Award from Department of Computer Science at the University of Illinois.

**Xiangfeng Chen** received the BS in mathematics from Nanjing University, Nanjing, China, in 1987, and the MS in computer science from the University of Illinois in 1995. He is now employed by Silicon Graphics Inc. working on microprocessor design. His primary research interests include integrated chip design, low power design, and physical design.



**David K. Poulsen** received the BS in electrical engineering from the University of Wisconsin-Madison in 1984, the MS in computer engineering from Syracuse University in 1988, and the PhD 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.



**Josep Torrellas** received a PhD in electrical engineering from Stanford University in 1992. He is an assistant professor in the Computer Science Department and Center for Supercomputing Research and Development of the University of Illinois at Urbana-Champaign. Professor Torrellas' primary interests are in the design of hardware and software for scalable shared-memory multiprocessors. He is currently leading the design of the Illinois Aggressive Cache Only Memory Architecture (I-ACOMA), a novel scalable shared-memory multiprocessor. Professor Torrellas received the National Science Foundation Research Initiation Award in 1993 and the National Science Foundation Young Investigator Award in 1994.