Data Forwarding in Scalable Shared-Memory Multiprocessors[1]

D. A. Koufaty, **X. Chen**, **D. K. Poulsen**[2] and **J. Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
koufaty,xchen,poulsen,torrella@csrd.uiuc.edu

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, on average, a slightly-optimistic support for forwarding speeds up five applications by 50% for large caches and 30% for small caches. For large caches, most read sharing misses can be eliminated, while for small caches, forwarding rarely increases the number of conflict misses. Overall, support for forwarding in shared-memory multiprocessors promises to deliver good application speedups.

1 Introduction

Scalable shared-memory multiprocessors are often slowed down by long-latency memory accesses. To cope with this problem, techniques that reduce the latency of memory accesses like sophisticated cache designs or locality optimizations have been used. However, while these techniques indeed reduce average latencies significantly, they tend to handle sharing-induced misses poorly.

To deal with this last situation, other techniques that overlap memory accesses with computation or other memory accesses are often necessary. These techniques include multithreading [2], relaxed memory consistency models [1, 6], data prefetching [11] and data forwarding [16]. In both prefetching and forwarding, the data is moved close to the consumer processor before it is actually needed. Therefore, when the processor finally accesses the data, it can do so with low latency. In data prefetching, the processor that is going to consume the data requests it in advance; in data forwarding, when a processor produces the data, in addition to updating its cache, it sends a copy of it to the processors that will consume it.

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and MIP 93-08098; by NASA Contract No. NAG-1-613; and by a grant from Intel Corporation.

[2]Now at Kuck and Associates, Inc., Champaign IL 61820, poulsen@kai.com.

Data forwarding is different from an update cache coherence protocol. In an update protocol, when a processor writes, all current sharer processors are updated. In our design of data forwarding, instead, a forward 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 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 exclusively on data forwarding.

There is little previous work on data forwarding. Forwarding was added to the DASH architecture with the *Deliver* instruction [10]. This instruction sends the data to several clusters specified in the instruction with a bit vector. In addition, the directory is updated in parallel. Unfortunately, there is no performance evaluation of the deliver instruction. Poulsen and Yew [14, 16] studied data forwarding alone and in combination with data prefetching. They showed that large gains can be achieved with forwarding alone and in combination with prefetching. However, they only studied large caches, which tend to show the positive side of forwarding only. Furthermore, they only evaluated a simple strategy for forwarding.

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. On average, a slightly-optimistic support for forwarding speeds up five Perfect Club codes by 50% for large caches and 30% for small caches. For large caches, most sharing read misses can be eliminated, while for small caches, forwarding rarely increases the number of conflict misses.

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; and, 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 inter-processor communication. In data forwarding, when a processor updates a word in its cache, it also propagates the update to the caches of all the processors that will 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 message with the update and the list of processors to receive the forward 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

Proc 0	Proc 1
	rd *x*
wr *x*	
	rd *x**

* True sharing miss
With forwarding: Hit
(a)

Proc 0	Proc 1
wr *x*	
	rd *x**

* Cold miss
With forwarding: Hit
(b)

Proc 0	Proc 1
	rd *y*
wr *x*	
	rd *y**
	rd *x*

* Hit
With forwarding: Conflict miss
(c)

Figure 1: Interaction of forwarding with the misses in single-word cache lines.

from the sharer list. When the caches of the consumer processors receive the update, they send an acknowledgement 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 not in this set are invalidated.

Data forwarding can be supported efficiently with single or multiple words per cache line. Following Poulsen and Yew's work [16], 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 contains the processors that should receive the update. This register is used implicitly by the instruction, and has to 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 write-and-forward access missed in the producer's cache, the directory-memory will send the complete memory line to the producer in addition to sending it to the necessary consumer processors. For every forward, an entry is allocated in the write buffer of the producer processor and kept until the acknowledgements from the consumers have been received. At that point, it can be retired. For this reason, the write buffer needs to be deeper than in non-forwarding systems.

The reason why the 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, with the two-hop scheme described, the order in which transactions to the same cache line modify a given consumer cache is the same as the order in which they are serialized in the directory-memory. This is because two messages between the same source and destination nodes are kept in order by the network.

If, instead, the forwards were sent directly from producer to consumer cache without a mid-trip hop in the directory, races could occur. Indeed, suppose a one-hop scheme where the producer processor sends the updates directly to the consumers and sends a notification message to the directory in parallel. For 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 wrong order: the invalidation first and the forward later. This situation would cause the consumer cache to wrongly store the forwarded data.

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 2-hop forwarding algorithm.

As indicated before, in the 2-hop forwarding algorithm, the directory-memory serializes the state changes for a memory line. This means that, for example, if a forward reaches a directory and the line that it wants is at that time changing states, the directory can bounce the forward. In this case, the originating processor will be forced to retry the transaction. A similar approach is used in the transactions of the DASH cache coherence protocol [10].

Even if, in the end, the forward instruction safely deposits the data 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 data transfer latency will be hidden. A second problem occurs when the data arrives and is displaced from the consumer cache by other data before it is used. To avoid this situation, the compiler should not replace a write by a write-and-forward instruction if the consumer processor is likely to access much data between the reception and the use of the forwarded line. 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 this problem, the compiler could identify the conflict and disable the forward. Alternatively, the compiler can reduce the likelihood of this problem by only forwarding data that will be used by the consumers soon. 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 and induce conflict misses. These cases are illustrated in Figure 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)) and a cold miss (Chart (b)), and induce a conflict miss (Chart (c)). For multiple-word cache lines, there are several positive and negative effects.

2.1 Optimizations

The base 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 some other processor, only the last write will be forwarded. An extension to this is to combine several forward messages into one if they are induced by the same processor updating different words in the same cache line.

Table 1: Comparing forwarding to software-based prefetching.

Issue	Comparison
Effectiveness	Prefetching targets all misses: cold, conflict, and coherence. Forwarding can only target coherence and some classes of cold misses. However, for these 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.

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 buffers each forward for a few cycles before sending it. If, in the meantime, the processor generates another forward to the same line, the first one is discarded. This approach is used by Glasco *et al.* [7] 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 fails to predict 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, we can forward the data to memory. As a result, the consumer processor 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 [10]. These forward-only instructions can then be software-pipelined like it is done with prefetches [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 [14]. 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.

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 a processor to memory and a memory to processor hop. 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. Furthermore, prefetch transactions generally have a higher latency than forwarding transactions. This is because they involve three steps: from consumer to directory to find out the location of the data, from directory to producer, and from producer to consumer[2]. 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 [11]. In this case, the producer sends 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 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, and a deeper write buffer that keeps an entry for each pending forward until the forward transaction is completed.

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. This suggests that both techniques should be combined. In this paper, however, we focus on data forwarding only.

[2]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.

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. 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 Figure 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
```

Figure 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:

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

where, from the loop bounds:

$$3 \leq I^P \leq 16 \quad \text{and} \quad 0 \leq I^C \leq 9 \qquad (2)$$

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 to 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**(Figure 3-(a)):
$$P(i) = i \; mod \; P$$
2. **Chunk Scheduling** (Figure 3-(b)). 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:
$$P(i) = \begin{cases} \left\lfloor \dfrac{i}{\lceil \frac{K}{P} \rceil} \right\rfloor & \text{if } r = 0 \text{ or } i < r(q+1) \\ \left\lfloor \dfrac{i-r}{\lfloor \frac{K}{P} \rfloor} \right\rfloor & \text{otherwise} \end{cases}$$
3. **Block Scheduling** (Figure 3-(c)). 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 processors. This produces a simpler expression:
$$P(i) = \left\lfloor \frac{i}{\lceil \frac{K}{P} \rceil} \right\rfloor$$

In our experiments, we use block scheduling. If, for this example where $K = 10$ we use 4 processors, we have:

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

Therefore, from Equation (1), an array element produced in iteration I^P of the producer loop will be consumed by processor $\lfloor \frac{2I^P - 1}{3} \rfloor$ in the consumer loop. Therefore, in Figure 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 as arguments the address to be written to and the IDs of the processors that will receive the forwards.

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

Figure 3: Scheduling algorithms for 10 iterations executed by 4 processors.

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. Therefore, to eliminate useless forwards, we need to examine the bounds of the consumer loop and make sure that all the forwarded data is consumed. From Equations (1) and (2), we have:

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

Looking at the second expression, it is clear that the leftmost inequality is always satisfied. The rightmost one is not. Therefore, it needs to be enforced with an *if* statement enclosing the forward. The resulting code is shown in Figure 4. To eliminate the *if* statement, we could split the loop.

```
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
```

Figure 4: Resulting code. The arguments of the write-and-forward macro are the location to be written to and the IDs of the processors that will receive the forwards.

3.1.1 Examples of Forwarding Expressions

Before finishing this section, we show two examples of the resulting forwarding expressions for loops in hand-parallelized versions of the Perfect Club codes [4, 5]. The examples show 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 from Serial to Parallel Section. One important parallel loop in *QCD* [5] consumes data produced in a serial section. The serial section and the loop are shown in Figure 5-(a). Constant propagation is used to determine all the values that appear in the loop bounds. Using our approach from Section 3.1 and extending the formulas to support two index variables I^P, J^P and I^C, K^C, we produce the code in Figure 5-(b). Note that, in Figure 5-(b), 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.

```
C   producer loop: it is filling an array with 0's,
C   except that for 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 consuming chunks
C   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)

Figure 5: Example of forwarding from serial to parallel section. Code (a) is the old code, while code (b) is the instrumented code for the producer.

Forward from Parallel to Parallel Section. This common case is illustrated by two parallel loops from *QCD* shown in Figure 6-(a). This time, when we apply our analysis, we have to insert a branch (Figure 6-(b)). In this case, 4-5 assembly instructions are required: 2-3 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 (1-5) 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.

```
C   producer: each processor produces 8 chunks
C   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 have work.
C   Each processor reads 16 groups of elements from
C   a 1152-element long subarray
    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)

Figure 6: Example of forwarding from parallel to parallel section. Code (a) is the old code, while code (b) is the instrumented code for the producer.

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 [9, 17], 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.* [12] suggest a threshold approximately equal to 1/16th of the size of the cache.

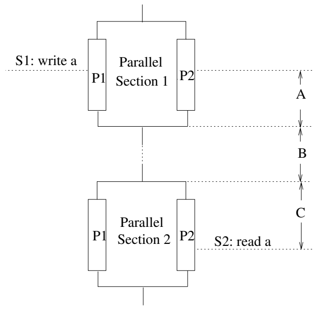

Figure 7: Computing the amount of data that the consumer processor needs to keep in its cache.

To understand our approach, consider the example in Figure 7. 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

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 2-D transonic airflow	4.0	29.3
ARC2D	Solves a 2-D fluid dynamics problem	5.5	32.1

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.

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 [15]. We start by compiling our applications with the Parafrase2 compiler [13]. 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 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 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. Caches are 4-way set-associative, have 8-byte lines and vary in size from 16 to 256 Kbytes. 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 [8]. In our evaluation, we assume that each forward requires 4 assembly instructions in the producer code.

4.2 Applications

We trace the parallel versions of the five Perfect Club codes [4] presented in Table 2. These versions run with 32 processors and were parallelized using a parallelizing compiler and later by hand [5]. 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. Then, we measure the speedups achieved by forwarding in the simulated architecture in two scenarios: when the caches are larger than the working set size of the applications and when they are smaller. 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 curve is approximately equal to the working set size [14]. 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

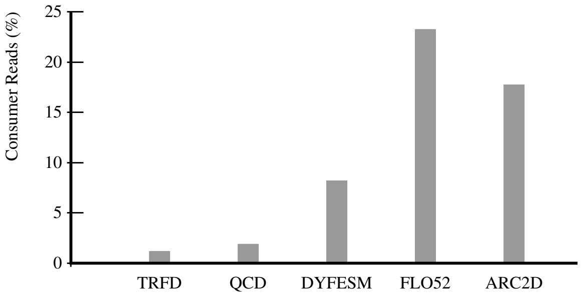

Figure 8: Fraction of reads that are consumer reads.

reads we call consumer reads. A consumer read is, therefore, the first read issued by a processor to a data that was written by a different processor. Figure 8 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%. Recall that, without forwarding and with single-word cache lines, these reads will all 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. Figure 9 plots the distribution of the distance between 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. We do not use the number of references because it lacks information about the locality of the data reference stream in the consumer. Using misses is not completely accurate 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 and 8-byte cache lines.

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. We measured this distribution for other cache sizes and found similar results, although some percentages varied a little.

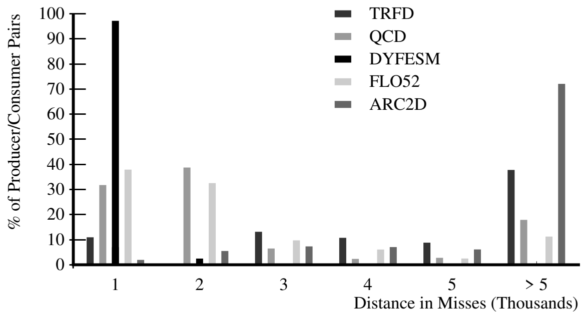

Figure 9: Distribution of distances between producer writes and consumer reads. Distances are measured in misses in the simulated architecture with 256-Kbyte caches and 8-byte cache lines.

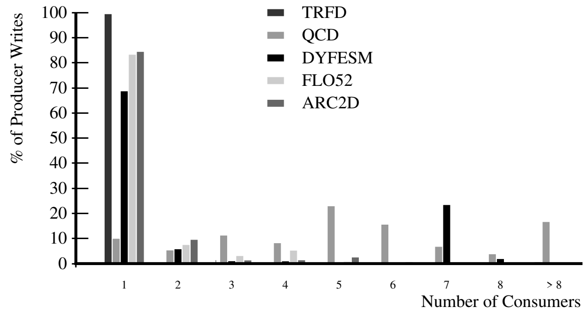

Figure 10: Number of consumer reads per producer write.

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. Figure 10 presents the distribution of such number for the applications. Clearly, the number of consumers per write can range from 1 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 in Caches Larger than the Working Set

We now measure the execution time speedups achieved by forwarding. In this section, we use 256-Kbyte caches. This size is larger that the working set size of the applications. This case is a favorable one for forwarding: forwarding is less likely to induce conflict misses. To gain insight into the effectiveness of forwarding, we compare five different levels of 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

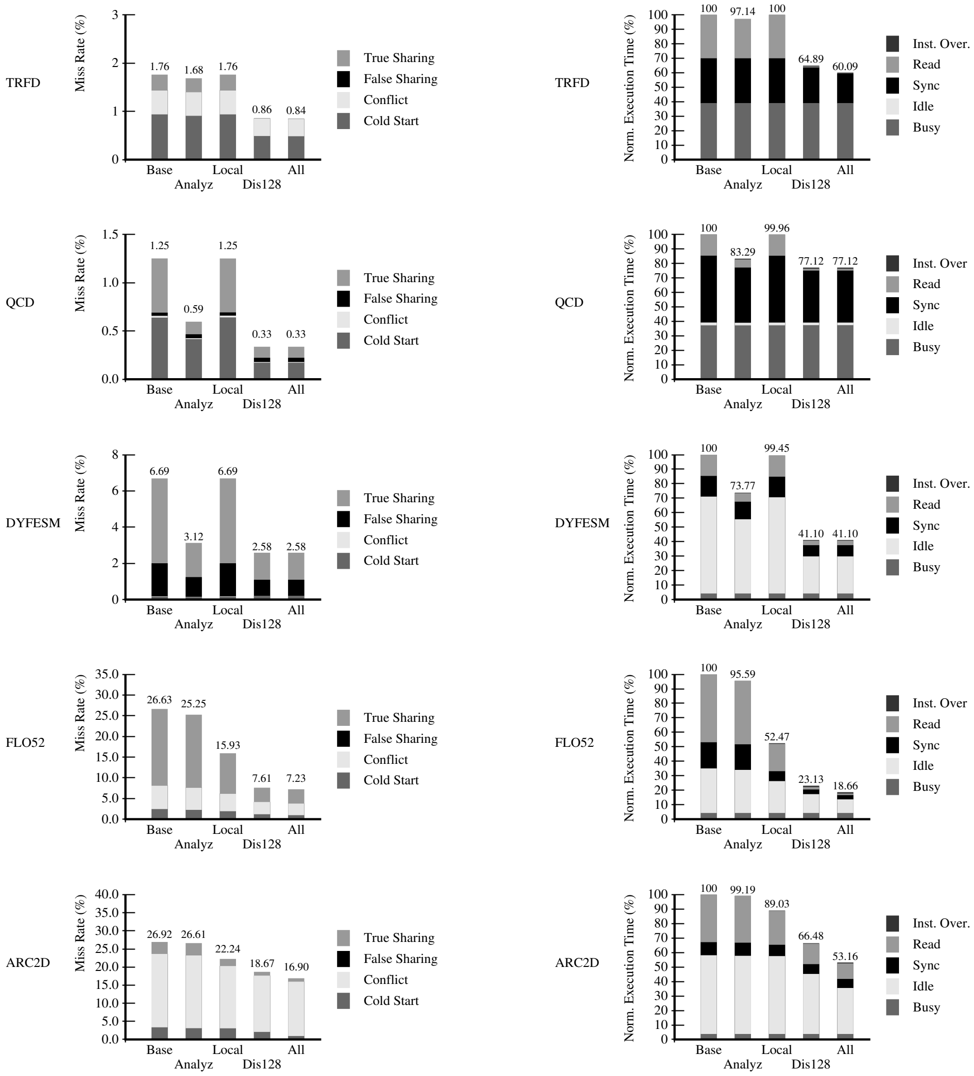

Figure 11: Miss rates and normalized execution times with 256-Kbyte caches. The miss rates include both reads and writes.

perform constant propagation, range 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.
- *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 128 Kbytes of data via misses. This corresponds to half the cache size.

Figure 11 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, 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.*). 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 includes time wasted 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 little forwarding takes place and, therefore, as seen in the leftmost charts, the cold and sharing 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 by speeding-up execution: processors spin less time in locks because locks are passed from processor to processor faster; in addition, 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 in *ARC2D*,

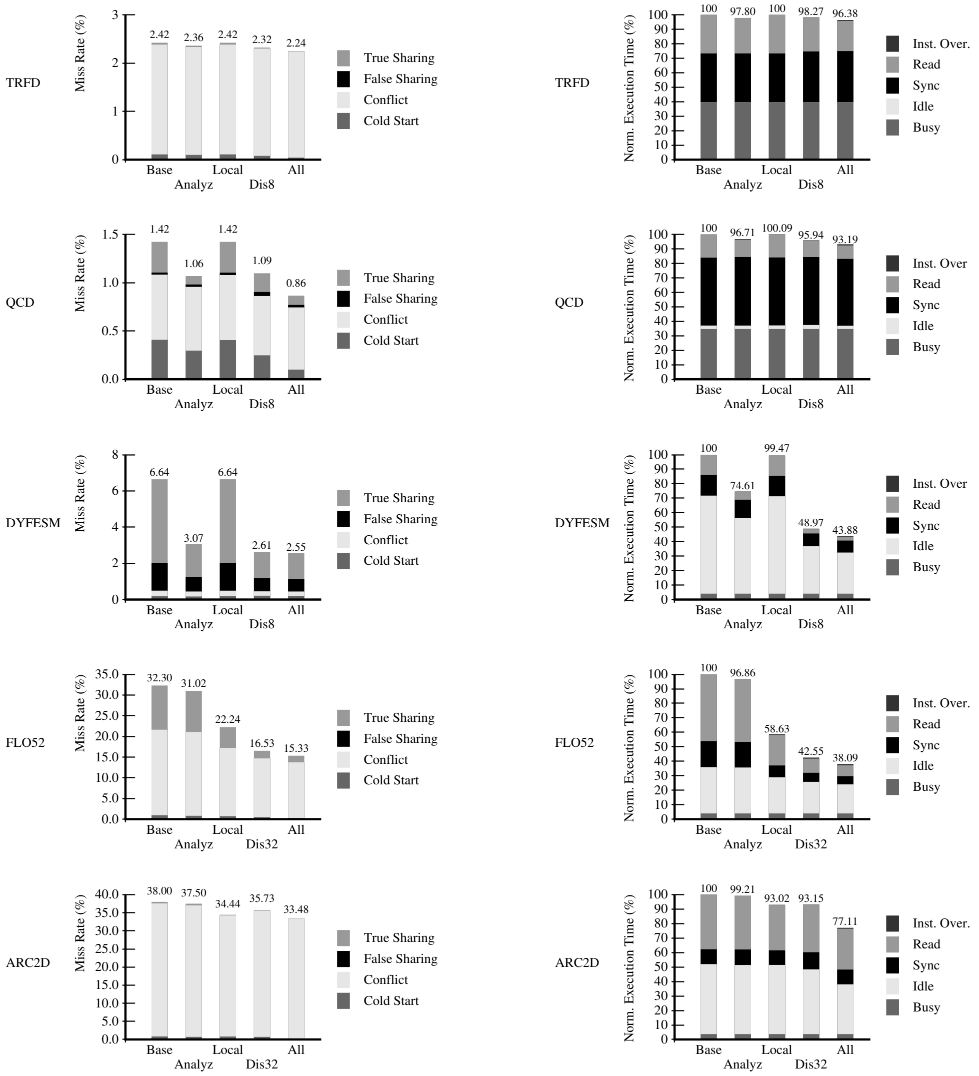

Figure 12: Miss rates and normalized execution times with 16-Kbyte caches for *TRFD*, *QCD* and *DYFESM*, and 64-Kbyte caches for *FLO52* and *ARC2D*. The miss rates include both reads and writes.

the rightmost charts show that the read miss stall time practically disappears. Remarkably, this is because the large majority of the cold, sharing, and conflict read misses are eliminated. The miss rates remaining in the leftmost charts are mostly caused by write misses. These misses are not targeted by our scheme. Forwarding can eliminate conflict misses because of the way we count them. Indeed, consider a word x that is invalidated and later the cache line where it was invalidated is replaced by another word y. If the processor then references x, we count the miss as a conflict miss even though the reference would have also missed without the conflict. Some of these conflict misses hide sharing misses that can be eliminated with 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 handle loop invariant terms and some kinds of non-affine functions in their analyses. 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 speed up by 11 and 48%. The reason for the overall low impact is the small number of forwards that can be issued locally. Therefore, we conclude 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* case. 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. In fact, the miss rates are higher, specially for *ARC2D*, which has long producer-consumer distances (Section 5.1.2). 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 in caches larger than the working set in the *All* case achieves substantial 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.

5.3 Forwarding in Caches Smaller than the Working Set

We now measure the execution time speedups achieved by forwarding in caches smaller than the working set of the applications. We use 16-Kbyte caches for *TRFD*, *QCD* and *DYFESM*, and 64-Kbyte caches for *FLO52* and *ARC2D*. Figure 12 presents the miss rates and normalized execution times of the five applications. The figure is organized like Figure 11. For space reasons, we only present a summary of the results.

As shown in the miss rate charts, conflict misses dominate in all applications except one. As expected, forwarding is generally not very successful at removing conflict misses. Remarkably, however, it does not increase the total number of conflicts. In fact, in *FLO52*, it even decreases them as explained in Section 5.2. Furthermore, it usually decreases the other types of misses, namely cold and sharing misses.

Focusing on *All*, we see that, even with small caches, forwarding speeds up three applications. Furthermore, no application suffers slowdowns. On average, *All* speeds up the five applications by 31%. For *Analyz* and *Local*, we reach the same conclusions as in Section 5.2. It is interesting, however, to examine the *Dis8/32* bars. While the execution times of *All* are only slightly smaller than those of *Dis8/32*, the numbers of forwards in *All* are up to 10 times larger than in *Dis8/32*. Therefore, most of the extra forwards in *All* do not bring any benefit. The forwarded data gets displaced from the consumer cache before it is used. The reason why the performance of *All* is not worse is two-fold. First, the data displaced by the extra forwards in *All* would have been displaced by other data anyway. This is intuitive if we note the large miss rates of *FLO52* and *ARC2D*. Second, the network that we use has a bandwidth large enough to tolerate the extra useless traffic generated. However, in networks with less bandwidth or when the rates of displacement are smaller, *Dis8/32* will likely perform better than *All*. Therefore, it is advisable to limit the forwarding distance in small caches. We are currently investigating the issue further.

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 via data forwarding. In this paper, we have presented a framework for a compiler algorithm for forwarding and, based on address traces, evaluated the performance gains 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 achieves large reductions in execution time. On average, the applications are sped-up by about 50% for large caches and 30% for small caches. For large caches, practically all sharing read misses and many cold read misses disappear. For small caches, forwarding rarely increases the number of conflict misses. For forwarding to be effective, however, the compiler must perform interprocedural analysis. Similarly, symbolic analysis, range analysis, and constant propagation are required for good performance.

We are currently investigating two issues. First, we are attempting the full automation of an algorithm for data forwarding and its integration in a compiler. Second, we are focusing on understanding the interaction between forwarding and prefetching and on using both techniques in the same code.

7 Acknowledgments

We thank the referees for their feedback. David Koufaty is supported by Universidad Simón Bolívar and CONICIT, both of Venezuela. Josep Torrellas is partly supported by an NSF Young Investigator Award.

References

[1] S. Adve and M. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 1–13, May 1990.

[2] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 3, pages 525–539, September 1992.

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

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

[5] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Center for Supercomputing Research and Development, October 1992.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[7] D. Glasco et al. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. Technical Report CSL-TR-93-588, Computer Systems Laboratory, Stanford University, November 1993.

[8] C. Kruskal and M. Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. In *IEEE Trans. on Computers*, pages 1091–98, December 1983.

[9] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[11] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[12] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[13] C. D. Polychronopoulos et al. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[14] D. K. Poulsen. Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. Ph.D. thesis, University of Illinois at Urbana-Champaign. Also Center for Supercomputing Research and Development Report 1377, July 1994.

[15] D. K. Poulsen and P.-C. Yew. Execution Driven Tools for Parallel Simulation of Parallel Architectures and Applications. In *Proceedings of Supercomputing '93*, pages 860–869, November 1993.

[16] D. K. Poulsen and P.-C. Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II, pages 276–280, August 1994.

[17] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.