Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs[1]

David Koufaty[2] and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
dkoufaty@ichips.intel.com torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

As the difference in speed between processor and memory system continues to increase, it is becoming crucial to develop and refine techniques that enhance the effectiveness of cache hierarchies. Two such techniques are data prefetching and data forwarding. With prefetching, a processor hides the latency of cache misses by requesting the data before it actually needs it. With forwarding, a producer processor hides the latency of communication-induced cache misses in the consumer processors by sending the data to the caches of the latter. These two techniques are complementary approaches to hiding the latency of communication-induced misses.

This paper compares the effectiveness of data forwarding and data prefetching to hide communication-induced misses. Although both techniques require comparable hardware support, forwarding usually has a lower instruction overhead. We evaluate prefetching and forwarding algorithms in a parallelizing compiler using execution-driven simulations of a shared-memory multiprocessor. Both data forwarding and prefetching reduce the execution time of applications significantly (30-40% on average). Forwarding performs better on average, while prefetching is more robust to changes in cache and memory parameters. Finally, we propose two ways of integrating the two techniques. The integration of the two techniques reduces the execution time even more (43-48% on average) and is very robust.

1 Introduction

As increases in processor speed continue to outstrip increases in memory system speed, it becomes crucial to develop and refine techniques that enhance the effectiveness of cache hierarchies. An important reason why cache hierarchies are sometimes ineffective in multiprocessors is the intrinsic interprocessor communication required by applications. To tolerate this communication, popularly known as data sharing, techniques that overlap processor computation with the communication have been proposed. Two of these techniques are data forwarding and data prefetching.

In data forwarding [1, 8, 10], producer processors send the data to the caches of the processors that will use the data in the near future. Data forwarding is appealing for three reasons. The first one is that we can forward as soon as the data is generated, therefore maximizing the chances of hiding the communication latency. Secondly, if we forward when the data is produced, we only need to generate the address of the data once. This is true even if multiple processors will consume the data. Finally, forwarding minimizes the use of network and directory resources, since the producer simply transfers the data to the consumer in one transaction.

In data prefetching [4, 9, 11], processors request in advance data that they will use later. This triggers a fetch to the memory system that brings the data to the local caches. Previous work by many researchers has conclusively shown that prefetching is highly effective. However, data prefetching has several disadvantages when compared to data forwarding. First, prefetching requests may be issued too late to hide the access latency completely. Second, if the data is needed by multiple processors, each one must generate a prefetch request, possibly resulting in a large instruction overhead in software-controlled data prefetching. Finally, prefetching increases network traffic and resource utilization. Indeed, in a prefetch-based system, two transactions are usually involved in the communication between a producer and a consumer: when the producer writes, it sends an invalidation to current sharers; later, the consumers issue a prefetch to get the data.

In this paper, we compare the effectiveness of data forwarding and software-controlled data prefetching to hide communication-induced misses. We study full-scale implementations of both techniques in the Polaris parallelizing compiler [3]. We evaluate scenarios with different cache and memory parameters. Based on this comparison, we propose integrated approaches that combine data forwarding and data prefetching to deliver the highest performance and robustness.

The rest of this paper is organized as follows: Sections 2 and 3 present the architectural and compiler support required; Section 4 presents a qualitative comparison of forwarding and prefetching; Section 5 discusses the evaluation environment; Section 6 studies the performance results and, finally, Section 7 presents the integrated algorithms.

2 Architectural Support

Building on past work, we use a type of data forwarding and data prefetching that is embedded in the directory-based cache coherence protocol of a scalable shared-memory multiprocessor. In forwarding, when a processor writes a datum, it updates its cache and sends the corresponding memory line to the caches of the predicted consumer processors via the directory. In prefetching, when a processor issues a prefetch, it checks its cache hierarchy. If it misses, it requests the datum from the directory.

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613 and gifts from Intel and IBM.

[2]Now at Microprocessors Group, Intel Corporation.

2.1 Primitives

Software-controlled data prefetching is supported with a prefetch primitive. Data forwarding is supported with a four-primitive framework. The choice of the latter is based on the insight that we gained from doing a complete data forwarding compiler implementation [7]. We now describe each of the primitives.

Prefetch. It is a non-blocking data read. If the data is found in the local caches, the instruction completes. Otherwise, the data is requested from the right directory and is finally deposited in the local caches.

Write-Forward. The producer processor updates its cache, the memory, and the cache of a single consumer processor specified as part of the primitive. At the same time, all other processors that shared the updated line are invalidated.

Write-Broadcast. The producer processor updates its cache, the memory, and the caches of all processors.

Write-Update. The producer processor updates its cache, the memory, and the caches of all the processors that, as indicated in the directory, currently share the line.

Write-Through. The producer processor updates its cache and the memory, while invalidating all processors that currently cache the updated line. Although this primitive does not eliminate any coherence misses, it reduces the latency of the next consumer read to the line because the memory has an up-to-date copy.

2.2 Data Forwarding

To support these forwarding primitives in a directory-based scalable shared-memory multiprocessor with multi-word cache lines we need some microarchitecture changes. Specifically, each of the primitives can be implemented with a different assembly instruction that is inserted in lieu of the producer write instruction. These instructions do not stall the processor. The write-forward instruction takes as an implicit argument a register with the ID of the single consumer processor.

The protocol changes are as follows. Consider the write-forward instruction. If the write-forward hits a line that is dirty in the producer's cache, the line is marked shared and the whole line is forwarded. If, instead, the write-forward 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 returns the complete line to the producer processor. In all cases, the producer, the directory and the memory are updated, the updated line is sent to the consumer processor, and the producer and the consumer processors are marked as sharers in the directory. Furthermore, all the other processors that currently have the line are sent invalidations and are removed from the sharer list in the directory. When all these changes are completed, an acknowledgment message is sent to the producer processor. At that point, the producer processor removes the write-forward's entry from its write buffer. The other primitives need similar protocol support. Notice that the write buffer needs to be deeper than usual. This is because it needs to keep an entry for each pending forward.

Based on previous work, we also add extra hardware support that helps forwarding perform better. First, forwards are delayed and combined. When a forward is deposited into the write buffer, it is not launched into the network immediately. Instead, it is delayed for a certain *delay* period. If, during that period, another forward of any word of the same line to the same consumer is deposited in the buffer, the two are combined. Then, the combined forward waits again for a *delay* period before being launched. This optimization reduces the number of network messages and the possibility of buffer overflow. The second optimization is to multicast the forwards in the directory by adding a *U* bit (for update) to each directory entry. When a forward reaches a directory entry, if *U* is zero, it sets it. Otherwise, the forward is part of a stream of forwards and, instead of invalidating all processors other that the consumer, the directory sends the forward to the consumer and updates all current sharers. This is done to allow different consumer processors to consume different words of the same cache line at the same time without invalidating all existing consumers in every forward. Once a regular read or write reaches the directory entry, the *U* bit is reset.

2.3 Data Prefetching

The microarchitecture supports non-binding data prefetching [9]. The prefetched data remains in the processor's cache until it is referenced. If, in the meantime, another processor writes to the data, the line is invalidated. This frees the compiler from the task of preserving correctness. Binding prefetches are not considered is this study.

To support data prefetching, the processor has an instruction that initiates a non-blocking load and a prefetch buffer. The prefetch buffer does not actually hold data, but the state of pending prefetch transactions. Its purpose is to keep track of all pending prefetches. After a prefetch misses in the cache, the prefetch buffer is checked to avoid issuing multiple requests for the same line. If a prefetch has not been issued for the same cache line, a prefetch is issued and an entry inserted in the prefetch buffer. Similarly, when a plain memory access misses in the cache, the prefetch buffer is checked to avoid generating multiple requests for the same cache line. Data prefetches that would overflow the prefetch buffer are discarded.

3 Compiler Support

To insert the forwarding and prefetching primitives automatically in realistic codes, we have taken the *Polaris* parallelizing compiler [3] and added a pass. Polaris takes a Fortran 77 program as input and generates a transformed Fortran program annotated with directives. These directives specify the parallelism in the code and are used to run the code efficiently on a parallel computer.

The model of parallelism in Polaris is loop-based. A program consists of a sequence of parallel and serial sections, and processors synchronize after each section. Polaris includes advanced algorithms that perform array privatization, data dependence testing, induction variable recognition and symbolic program analysis. These passes expose much parallelism. Figure 1 shows a code segment from SPEC95's *tomcatv* after Polaris has parallelized it. *Doall* indicates a fully-parallel loop.

```
DO j = 1, n-3, 1
  DOALL i = 2, n-1, 1
    rx(i,n-j-1) = ...        1
  ENDDO
ENDDO
DOALL j = 2, n-1, 1
  DO i = 2, n-1, 1
    ... = rx(i,j) ...        2
  ENDDO
ENDDO
```

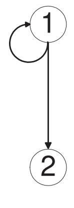


Figure 1: A code segment and its data flow graph.

3.1 Finding Producer-Consumer Pairs

The first phase of our algorithm identifies the pairs of statements that produce and consume the same datum. In our analysis, we focus only on accesses to shared arrays. This is because scalars have very little weight in the codes considered. In addition, since it has been suggested that it is necessary to analyze the codes interprocedurally [8], and Polaris lacks good interprocedural analysis, we inline the programs. First, we build a data flow graph (Figure 1). Each node is a serial or a parallel section of the program and edges correspond to the flow between one section and another. Second, a Polaris pass is used to identify the array sections that are read and those that are written in each node. Finally, data flow analysis is used to compute the set of definitions that reach each node of the graph. The details of this algorithm are described in [7].

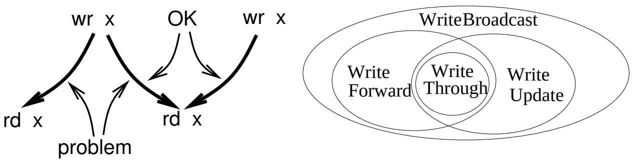
The reaching definitions are then intersected with the array sections that each node reads to generated a set of potential producer-consumer pairs. Each pair is composed of a write and a read reference in the source code that produce and consume a given array section, respectively. In the example of Figure 1, the pair is composed of references "`rx(i, n-j-1) =`" and "`= rx(i, j)`".

3.2 Inserting Forwarding Instructions

If data forwarding is requested, the second phase of our algorithm selects one of the data forwarding primitives for each producer-consumer pair based on the characteristics of the producer and consumer expressions.

The primitive is selected based on the complexity of the expressions and the potential overhead and benefits in each case. The criteria used includes the location of the consumer read (parallel or serial section), complexity of the consumer expression, number of consumer expressions, independence of the consumer expression from the enclosing parallel loop and ability of the compiler to analyze both expressions [7].

Before generating the final code, the compiler must handle the case where an access belongs to several producer-consumer pairs. Figure 2-(a) shows the two possible cases. A consumer read that belongs to several producer-consumer pairs poses no problem: each producer write is handled separately. However, a producer write that belongs to several producer-consumer pairs may cause a problem: the separate analysis of each producer-consumer pair may lead to different forwarding primitives for the same write.



(a) (b)

Figure 2: Merging of forwarding primitives.

When this occurs, we select the most appropriate primitive based on the hierarchy of Figure 2-(b). In the figure, the primitives are represented as sets. If one set includes a second one, the first primitive is preferred over the second one in case of conflict. For example, the write-through set is included in all sets, which means that any other primitive is preferred over write-through. The reason is that the action of a write-through, namely, updating memory, is also performed by all the other primitives. When we have a write-forward and another write-forward with different target processor or a write-update, we choose the one that is most likely to be useful.

```
      DO j = 1, n-3, 1
       DOALL i = 2, n-1, 1
DIR$   FORWARD (n-j-3)/((n-3)/P+1)
        rx(i, n-j-1) = ...
       ENDDO
      ENDDO

      DOALL j = 2, n-1, 1
       DO i = 2, n-1, 1
        ...  = rx(i, j) ...
       ENDDO
      ENDDO
```

(a)

```
      DO j = 1, n-3, 1
       DOALL i = 2, n-1, 1
        rx(i, n-j-1) = ...
       ENDDO
      ENDDO

      DOALL j = 2, n-1, 1
       DO k = 0, 7
DIR$    PREFETCH (rx(2+k,j))
       ENDDO
       DO i = 2, n-1, 1
DIR$    PREFETCH (rx(i+8,j))
        ...  = rx(i, j) ...
       ENDDO
      ENDDO
```

(b)

Figure 3: Code after adding data forwarding (a) and data prefetching (b).

The compiler algorithms are described in detail in [7]. Figure 3-(a) shows the example code segment after data forwarding has been added. The *Forward* directive specifies the ID of the consumer processor for the *rx(i, n-j-1)* location as a function of the loop indices, loop bounds, and the number of processors P.

3.3 Inserting Prefetching Instructions

When data prefetching is requested, our implementation is based on the same set of producer-consumer pairs found in the first phase of the algorithm. For each producer-consumer pair, we mark the corresponding read to be prefetched. After all the reads have been marked, we proceed to schedule the prefetches for each reference.

We schedule the prefetches using software-pipelining [9]. Prefetches must be issued early enough to hide the latency of the accesses. We use an estimate of the number of cycles needed to execute one loop iteration, *time*. Conservatively, we assume that all operations take one cycle, all memory accesses result in a cache hit, and loops are executed at least once. If *latency* is the average latency for a remote miss in cycles, then the prefetch instruction needs to be scheduled $\lceil latency/time \rceil$ iterations ahead.

Figure 3-(b) shows the scheduling for the example of Figure 1. Assume that the *latency* is such that we need to schedule the prefetches 8 iterations in advance. Consequently, during the loop prolog, we issue the first eight prefetches for as many loop iterations. During the execution of iteration i we prefetch the element needed for iteration $i + 8$. For simplicity, we do not peel the last few iterations of the loop, although that would eliminate unnecessary prefetches. Finally, although not shown in the figure, we unroll the loop to issue only one prefetch per cache line.

4 Qualitative Comparison

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 could eliminate any kind of read misses, namely cold, conflict and coherence misses. Forwarding, on the other hand, can only eliminate communication-induced misses, namely 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

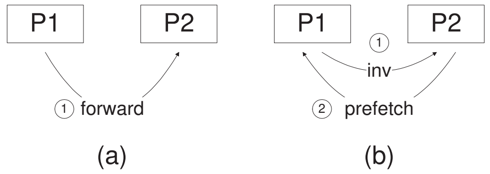


Figure 4: Transactions needed in data forwarding (a) and data prefetching (b).

the consumer as soon as it is produced. Secondly, the data is transferred with high efficiency: one message from the producer to the directory and from the directory to the consumer (Figure 4-(a)). Finally, a producer can forward the same data to all 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, data transfer between producer and consumer is now less efficient, involving two transactions: an invalidation from producer to consumer when the former writes the data, and the prefetch from the consumer to the producer (Figure 4-(b)). The latter often involves three network hops. Finally, if several consumers want the data, all of them have to execute prefetch instructions.

The compiler support required for data forwarding is more complex. Indeed, if a processor issues prefetches to eliminate non-coherence misses only, the compiler analysis can focus exclusively on that processor's code. Furthermore, while in both forwarding and prefetching for coherence misses the compiler needs to analyze the code executed by different processors, prefetching requires a simpler analysis. This is because, to use the write-forward primitive, the producer processor needs to compute the identity of the consumer processor.

Both techniques require roughly comparable hardware support. Our 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 prefetch buffer. Forwarding, on the other hand, requires one or several write-and-forward instructions that write the cache through, lock-up free caches, a deeper write buffer to keep pending forwards until completion, and the ability for a cache to accept data that it has not requested.

Finally, both techniques can cause cache conflicts by bringing too much data to the consumer cache before it is actually used. This problem is slightly less acute in prefetching than in the type of forwarding presented because the compiler can control the time when the prefetch is issued. In the type of forwarding that we presented, however, the forward is always issued when the data is written. Similarly, both schemes have 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. In practice, data forwarding often has an edge because of three reasons. First, a write-broadcast can transfer data to several consumers with one single instruction, while in prefetching the same data transfer requires instructions in all consumers. Second, none of the forward instructions beyond the write-forward needs to compute a destination processor. Third, even for write-forwards, the forwarding expression is often loop invariant and, therefore, can be moved out of the loop. All this discussion is summarized in Table 1.

Table 1: Comparing forwarding to software-based prefetching.

Issue	Comparison
Effectiveness	Prefetching targets all misses: cold, conflict and coherence. Forwarding only targets coherence and some classes of cold misses. However, for these misses, forwarding is more efficient.
Compiler Support	Forwarding requires more sophisticated compiler support.
Hardware Support	Both strategies require comparable hardware support.
Overhead	Both strategies can cause cache conflicts and have instruction overhead. Data forwarding often has less instruction overhead.

5 Simulation Environment

To evaluate our compiler algorithm, we use TangoLite [6] execution-driven simulations of applications parallelized by Polaris and annotated by our forwarding and prefetching passes. The architecture model that we simulate is a scalable Flat COMA multiprocessor [12] with 32 processors cycling at 300 MHz. Each processor has a two-level cache hierarchy. Table 2 shows the latencies for a round trip to different levels of the memory hierarchy. To schedule prefetches, the memory latency is taken to be 500 cycles. The first-level caches are direct-mapped, have a size of 16 Kbytes, and have a line size of 64 bytes. The second-level caches are 4-way set-associative, vary in size from 32 to 512 Kbytes, and have a line size of 64 bytes. The depth of the write buffers is 20 entries and their width is one cache line. The prefetch buffer also has 20 entries. The attraction memory in each node is 4-way set-associative and has a line size of 64 bytes. The ratio of the application's total data size to the total memory in the machine is called the memory pressure. In our experiments, for each application, we perform two experiments, one with a low memory pressure (40%), and one with a higher one (60%). Finally, in our simulations, we accurately model contention in all parts of the machine except in the global network that connects the nodes. The latter is a 2D mesh with a hop latency of 150 cycles.

In our simulations, we use the release memory consistency model. Data forwards deposit the data in the second-level cache of the consumer processor. If the first-level cache of the consumer has an old copy of the memory line, the forward pushes the data to the first-level cache. Prefetching, however, always brings the data to the first-level cache.

To drive the simulations, we use six Polaris-parallelized Fortran applications from a wide variety of sources to cover as many cases as possible: *emit* and *gmtry* from the NAS benchmark set [2], *swim* and *tomcatv* from SPEC95, *fft* from the CMU suite [5], and *lu*, a custom-developed LU factorization. For the SPEC95 applications, we use the test data set. The test and the reference data sets differ only in the number of iterations executed, not in the data set size. Consequently,

Table 2: Latencies of contention-free round-trip accesses to different levels of the memory hierarchy.

Level	Latency (Cycles)
First-level cache	1
Second-level cache	10
Local attraction memory	70
Remote attraction memory (2-hops)	400
Remote attraction memory (3-hops)	600

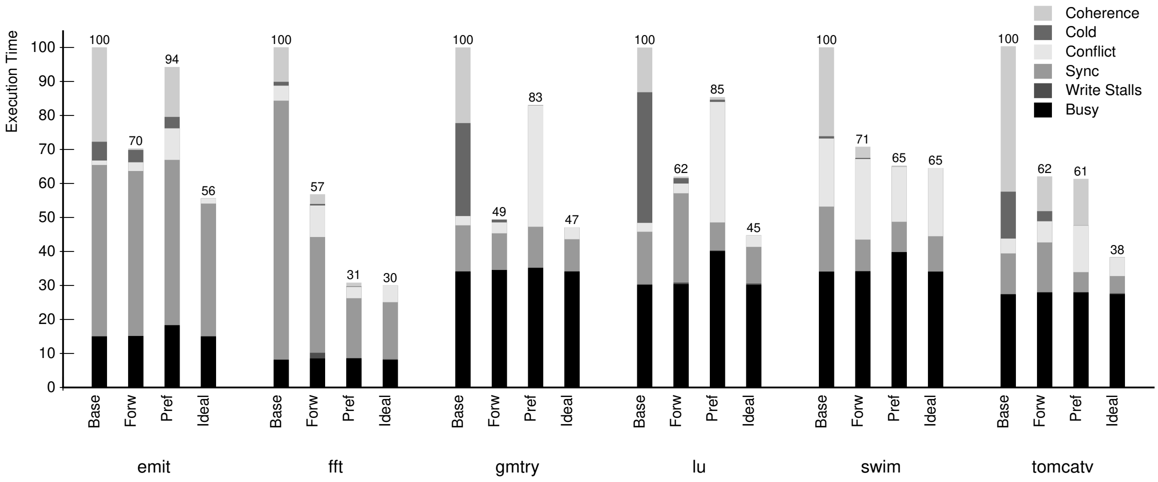


Figure 5: Impact of forwarding and prefetching on the execution time for the baseline system.

the results of our experiments are largely the same in both cases.

We use small caches in the simulations because we run small applications. For the second-level caches, we use two different sizes: one that is large enough to hold the working set of the application and one that is not. These two cache sizes and their miss rates are shown in Table 3. For a given application, we find these sizes by measuring the change in the miss rate as we vary the size of the cache [7].

Table 3: Applications studied.

Name	Small L2		Large L2	
	Cache (Kbytes)	Miss Rate (%)	Cache (Kbytes)	Miss Rate (%)
emit	32	3.7	128	2.3
fft	64	6.3	256	5.4
gmtry	32	1.6	128	1.0
lu	32	2.2	256	1.3
swim	64	4.1	512	1.2
tomcatv	64	4.5	512	2.6

6 Performance Results

In our evaluation, we first consider a baseline environment with the large second-level caches and the low memory pressure. Then, we consider environments where these conditions change. In all experiments, we assume that the schedule of the loops that contain the consumer reads is known at compile time. In these loops, the iterations are scheduled in as many blocks as the number of processors.

6.1 Baseline Environment

Figure 5 shows the execution time of each application under four scenarios. *Base* is the baseline architecture. *Forw* and *Pref* are *Base* plus forwarding and prefetching, respectively. Finally, *Ideal* is an environment where all reads that consume data produced by another processor (including those not identified by our compiler pass) and miss in the first-level cache, magically find the data requested in the second-level cache. Therefore, it is an upper bound on the performance of data forwarding.

In all applications, the bars are normalized to *Base*, and divided into 6 categories: stall time due to coherence (*Coherence*), cold (*Cold*), and conflict (*Conflict*) misses, synchronization time (*Sync*), stall time due to write buffer overflow (*Write Stalls*) and useful busy time (*Busy*). To simplify the chart, for prefetching, we have lumped the stall time due to partially-hidden prefetches into *Conflict*.

Focusing on the *Forw* bars, we see that the execution time of the applications is reduced by an average of 40% relative to *Base*. In all applications, *Coherence* and *Cold* times decrease significantly and the instruction overhead of forwarding is negligible. The synchronization time often decreases, which the result of speeding up the serial sections of the applications. Overall, however, there is still some room for improvement to get close to *Ideal*, which has about 50% less execution time than *Base*.

Focusing on the *Pref* bars, we note that in three applications, namely *emit*, *gmtry* and *lu*, prefetching is not very effective. One of the reasons for this, especially in *gmtry* and *lu*, is that many of the prefetches are not able to hide all the latency of the memory access. This is because, in these applications, the parallel loops often have small iteration counts. As a result, the processors stall. As indicated before, this stall time is lumped into the *Conflict* category. Forwarding, instead, is much more effective at hiding the latency of communication in these applications.

A second reason for the small prefetching speedups, especially in *lu* and, to a lesser extent in *emit*, is the increase in the *Busy* time due to the instruction overhead. This overhead accounts for over 10% of the execution time in *lu*. Forwarding has much less instruction overhead because it uses the low-overhead *write-broadcast* instruction and *write-forward* instructions in loop-invariant form. *lu* also has the problem of hot spots, since all processors try to get the pivot column right after a synchronization point. This increases the contention in the system, resulting in increased miss latencies. Forwarding avoids the problem of hot spots by broadcasting data in *lu*.

Prefetching performs much better than forwarding in *fft*. This application has a large serial section, as shown by the weight of the *Sync* time. Prefetching is able to speed up this serial section much more than forwarding. There are two reasons for this. One reason is that our prefetching algorithm, in addition to hiding communication misses as in forwarding, may also hide conflict misses. This occurs when a given read instruction may, in some iterations, consume data produced by another processor and, in other iterations, simply read data produced by the same processor. In the latter case, the prefetch may actually eliminate a conflict miss. The second reason is that forwarding from the serial section can cause the bunching-up of forward messages right before the end of the serial section. This slows down execution because the barrier that synchronizes all the processors at the end of the serial section cannot complete until all these forwards are completed.

In the other two applications, namely *swim* and *tomcatv*, the speedup of prefetching is similar to that of forwarding. Most prefetches are on time and the consumer reads find the data in the caches. Also, the instruction overhead is small, except in *swim*. Overall, prefetching reduces the execution time of the applications by an average of 30%.

6.2 Effect of Cache and Memory Size

We now repeat the previous experiments with the small second-level caches listed in Table 3. While conflict misses accounted for 22% of the misses in the large caches, they now account for 50%. As a result, the *Conflict* time is relatively more important, although it still accounts for only a very modest part of the total execution time. In this case, forwarding increases the number of conflict misses, and its absolute performance impact is reduced. *Forw* now reduces the execution time of the applications by an average of 35%. *Ideal* does it by 50%.

Prefetching, however, performs relatively better than before. The reason is that, with the higher *Conflict* time induced by the smaller caches, the base system runs slower. Recall that, as indicated before, *Pref* can also hide some conflict misses at the same time as it hides communication misses. Consequently, prefetching has a higher beneficial effect. Overall, *Pref* now reduces the execution time of the applications by an average of 32%. Consequently, prefetching shows more robustness than forwarding with respect to cache size.

We also repeat the experiments with the default cache size and the higher memory pressure of 60%. The result is an increase in the relative contribution of the *Conflict* time in the applications due to more conflicts in the attraction memories. While data prefetching is able to decrease the number of conflict misses in the attraction memory, data forwarding is not. Overall, *Pref* and *Forw* now reduce the execution time of the applications by an average of 31% and 30% respectively. *Pref* is again more robust.

7 Integrated Techniques

Given that both data forwarding and data prefetching have good aspects, we now consider the integration of both techniques.

7.1 Issues Affecting the Performance

To decide whether data forwarding or data prefetching is the best scheme for a given producer-consumer pair, we consider several factors that determine the performance of these primitives.

Serial sections. Serial sections limit an application's performance. Unfortunately, serial sections are slowed down by data forwarding or data prefetching instructions. Consider for example, a producer-consumer pair where the producer is sitting in a parallel section while the consumer is in a serial section. Instrumenting the consumer read with prefetches would result in a larger slowdown than instrumenting the producer write with forwards. The opposite is true if the producer is in a serial section and consumer is in a parallel section.

Instrumentation. Data forwarding tends to have less instruction overhead than data prefetching in many cases. First, none of the data forwarding primitives beyond write-forward needs to compute the consumer processor, resulting in a zero instruction overhead. Second, a write-broadcast instruction needs to be matched by $P-1$ prefetches, where P is the number of processors. These prefetches have a significantly higher instruction overhead. Finally, many forwarding expressions are loop-invariant and can be placed outside of the loop.

Conflict misses. If a consumer read can also cause a conflict miss, data prefetching can be used to hide both the coherence and the conflict misses. Forwarding, on the other hand, can not hide conflict misses.

Data size. Consider a processor that will read a large data structure written by other processors. If forwarding is used, we run the risk of wiping out a significant fraction of the consumer processor's working set from its cache. With software-pipelined prefetching, instead, we may avoid this problem because we bring the data into the consumer cache incrementally.

Distance. When the time distance between the producer write and the consumer read is large, prefetching is more attractive. The reason is that, with prefetching, the data is brought into the memory hierarchy of the consumer processor much closer to the time when the data will be used. This reduces the likelihood of the data being displaced from the cache.

Multicast limits. In the case of multiple consumer reads for a single producer write, data forwarding is limited by our lack of support for multicasting. Our current approach is to select one consumer over all the others and forward the data only to it. Clearly, it is more effective to insert prefetch instructions in every consumer read.

7.2 Combining Forwards with Prefetches

We now present two ways of integrating data forwarding and data prefetching. In our schemes, for each producer-consumer pair we have three choices: use only forwarding, use only prefetching or use both. Using both forwarding and prefetching for a given producer-consumer pair does not cause cache coherence protocol races because all transactions directed to a given memory line are serialized in the corresponding directory entry.

Naive combining. In this scheme, a forwarding write and a prefetching read are inserted for every producer-consumer pair.

Smart combining. In this scheme, we take into account the factors discussed in Section 7.1. For a given producer-consumer pair, we will use both forwarding and prefetching unless any of those factors apply. In the latter case, depending on the factor that applies, we will remove either the forward or the prefetch. We qualitatively assign priorities to the factors in the following order: serial sections, instrumentation, conflict misses, data size, distance and multicast limits. The exact details of the scheme used can be found in [7].

7.3 Performance Results

Figure 6 presents the results of the experiments for the baseline environment. The figure is organized in the same way as Figure 5. Two bars are added: the base architecture with naive combining (*C-Naive*) and the base architecture with smart combining (*C-Smart*).

Let us focus first on *C-Naive*. Naive combining is able to reduce the execution time of four applications by 2%-8% over the previous best case, either prefetching or forwarding alone. This indicates that prefetching and forwarding complement each other reasonably well. For example, data forwarding can

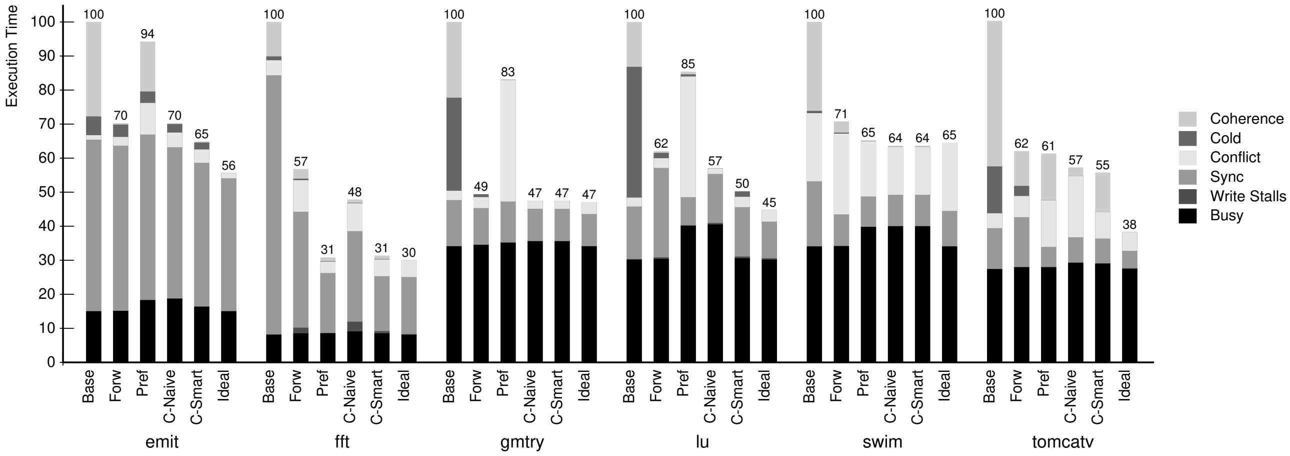


Figure 6: Performance of the integrated techniques for the baseline system.

reduce the number of late prefetches. Forwarding sends the data to the second-level cache of the consumer processor, and prefetching gets the data from there and deposits it into the first-level cache in a few cycles. Only *fft* runs slower under *C-Naive* than under *Pref*. The reason is that, in this case, the negative effects of forwarding, namely the increase in write buffer stall time and the delay of release points, dominate.

Now let us study *C-Smart*. In all applications, the performance of smart combining is better than the performance of the best technique alone. On average, *C-Smart* reduces the execution time of the best technique alone by 7%. This is a significant improvement. However, *C-Smart* performs no better than *C-Naive* for two applications, namely *gmtry* and *swim*. This is because, in these applications, smart combining results in exactly the same instrumentation as naive combining. In all the other applications, removing unneeded forwards or prefetches helps reduce the execution time.

To understand these results, Figure 7 shows the fraction of dynamic forwards and prefetches in *C-Naive* that remain after applying *C-Smart*. From the figure, we find changes in four of the six applications. In *emit*, forwards in serial sections and high-overhead prefetches were removed (the later because of loop-invariant forwarding expressions or write-broadcasts). In *fft*, forwards that moved large amounts of data and forwards to references that induced conflict misses were removed. In *lu*, the write-broadcast of the pivot column was preferred over prefetching. Finally, in *tomcatv*, a few prefetches were removed from a serial section, while a significant number of forwards were removed because of a large distance and a large amount of data.

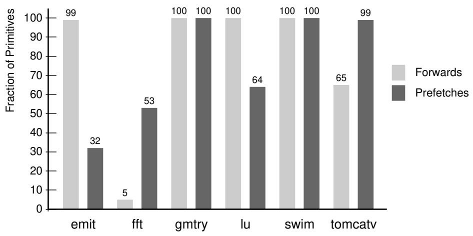


Figure 7: Fraction of primitives left after smart combining.

Overall, naive and smart combining helps increase the performance. Naive combining is good enough for most applications. However, smart combining leads to larger performance improvements. Smart combining performs better than any of the other techniques, including data prefetching and data forwarding alone. On average, naive and smart combining reduce the execution time of the applications by 43% and 48% respectively.

7.4 Effect of Cache and Memory Size

In general, if we reduce the size of the second-level cache, we observe the same trends as for the large caches. One difference is that, since prefetching is more robust with respect to cache size, the combined approach is also more robust than data forwarding. Indeed, all applications get a significant performance benefit, and the smart combining approach performs best. Recall that, for small caches, data forwarding and data prefetching separately reduced the execution time by 35% and 32% respectively. The naive and smart combining approaches deliver execution time reductions of 42% and 47% respectively. This is similar to the experiments with a large cache size.

Finally, we consider high memory pressure and large second-level caches. While increasing the memory pressure impairs the performance of data forwarding, the integrated approach is able to exploit the available resources efficiently, resulting in average execution time reductions of 42% and 47% for the naive and smart combining approaches, respectively. Overall, therefore, the combining schemes are quite robust to changes in cache and memory parameters.

8 Conclusions

The contribution of this paper is two-fold. First, we compared full-scale compiler implementations of a data forwarding and a data prefetching pass for communication-induced misses. Second, we integrated both techniques in a combined scheme.

We evaluated our prefetching and forwarding algorithms using execution-driven simulations of a shared-memory multiprocessor. Data forwarding and prefetching reduced the execution time of the applications by an average of 40% and 30% respectively. Forwarding performed better on average, while prefetching was more robust to changes in cache and memory parameters. We proposed two ways of integrating the two techniques. The integration of the two techniques reduced the execution time even more (43-48% on average) and was very robust.

Acknowledgments

We thank the referees and the members of the I-ACOMA group for their valuable feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on High-Performance Computer Architecture*, pages 204–215, February 1997.

[2] D. H. Bailey and J. T. Barton. The NAS Kernel Benchmark Program. Technical report, Numerial Aerodynamic Simulations Systems Division, NASA Ames Research Center, June 1986.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[4] T. F. Chen and J. L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[5] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, Carnagie Mellon University, March 1994.

[6] S. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[7] D. Koufaty. *Architectural and Compiler Support to Hide Coherence Misses in Distributed Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign, October 1997.

[8] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(12):1250–1264, December 1996.

[9] 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.

[10] 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.

[11] V. Santhanam, E. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[12] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.