Compiler Support for Data Forwarding in Scalable Shared-Memory Multiprocessors[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. One promising technique in the context of scalable shared-memory multiprocessors is data forwarding. Forwarding hides the latency of communication-induced misses by having producer processors send data to the caches of potential consumer processors in advance. Forwarding can hide the latency effectively, has low instruction overhead, and uses few machine resources.

This paper presents a complete implementation of a data forwarding pass in an industrial-strength parallelizing compiler. Complete Fortran applications are analyzed for dependences and, based on the analysis, automatically annotated with forwarding directives. We propose a forwarding framework that includes 4 new instructions: *write-forward*, *write-broadcast*, *write-update*, and *write-through*. New microarchitectural support is proposed. In our analysis, we assume that the assignment of loop iterations to processors is known. We perform simulations of multiprocessors with different cache, memory, machine sharing, and process migration parameters. We conclude that data forwarding delivers large speedups (six 32-processor applications ran an average of 40% faster), gets close to the upper bound in performance, and needs compiler support of only medium complexity.

1 Introduction

As increases in processor speed continue to outstrip increases in memory speed, it becomes crucial to develop and refine techniques that enhance the effectiveness of memory hierarchies. An important reason why memory hierarchies are sometimes not very effective in multiprocessors is the intrinsic interprocessor communication required by parallel applications. To cope with this communication, popularly known as data sharing, researchers have proposed techniques that overlap the communication latency with processor computation. Perhaps the most well-studied of these techniques is data prefetching. Work by many researchers has shown that prefetching is highly effective [5, 15, 19].

Another way of hiding communication latency is for producer processors to forward the data to the caches of the potential consumer processors in advance. This technique is known as data forwarding [1, 12, 17, 18, 20]. Data forwarding is appealing for three reasons. The first one is that the producer can forward the data as soon as it generates it, 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, possibly saving instructions over prefetching. Finally, forwarding minimizes the use of network and directory resources by reducing the number of transactions needed to transfer the data over prefetching.

Of course, data forwarding has shortcomings. Unlike prefetching, forwarding is only effective for communication-induced cache misses. In addition, the compiler and hardware support for forwarding may be more complicated.

Because of all these issues, Poulsen and Yew [17], Koufaty et al [12], Ramachandran et al [18], Abdel-Shafi et al [1], and Sivasubramaniam [20] among others have examined data forwarding. The work of all these researchers is largely based on analysis of traces of applications that have been instrumented with forward statements. These researchers uniformly conclude that forwarding can deliver large benefits. In addition, Ohara also showed good performance gains in a hardware-only approach [16].

The contribution of this paper is a complete, fully-automated implementation of a data forwarding pass in the Polaris parallelizing compiler [4]. Only with a complete implementation we can get a true picture of forwarding. We also propose a new framework with four flavors of forwarding and new microarchitectural support. Our analysis assumes that we know the assignment of loop iterations to processors.

Techniques similar to ours have been applied to compilers for message-passing multiprocessors [3, 10, 23]. Besides the obvious differences in the required hardware support, the key difference with our approach is that we do not use the *owner-computes* paradigm to insert the required communication. In the owner-computes paradigm, a processor owns the data allocated in its memory and is required to send the current copy of the data. In our approach, the last processor to write the data sends it to potential consumers. Additionally, our approach addresses the tradeoff between compiler analysis complexity and performance differently because we are not concerned with maintaining program correctness – the underlying cache coherence protocol guarantees it.

Other researchers have proposed similar techniques for page-based, software DSMs. Dwarkadas et al [7] proposed the insertion of primitives to inform the run-time system of data access patterns. Although part of their analysis is similar to ours, their approach is intended for the large overheads of software DSM communication, uses a page-sized grain, and makes the run-time system responsible for sending the data from producers to consumers.

In the second part of this paper, we evaluate data forwarding under many simulated scenarios, including favorable and unfavorable values of cache size, memory size, machine sharing, and process migration parameters. We conclude that fully-automated data forwarding delivers large speedups (six 32-processor applications ran an average of 40% faster), gets

[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 Performance Microprocessors Division, Intel Corporation.

close to the upper bound in performance, and needs compiler support of only medium complexity. Due to space constrains, we do not discuss results on the integration of prefetching with forwarding, which gets an extra 15% increase in performance [13].

This paper is organized as follows: Section 2 presents the architectural support required; Section 3 presents the compiler algorithm; Section 4 discusses the evaluation environment; and Section 5 presents the evaluation.

2 Architecture

Building on past work, we use a type of forwarding that is embedded in the directory-based cache coherence protocol of a scalable shared-memory multiprocessor. This implies that, when a processor forwards a datum, it updates its cache and sends the corresponding memory line to the caches of the predicted consumer processors via the directory. The message goes through the directory to avoid protocol races [12]. The directory also sends invalidations to the caches of the non-consumer processors that have a copy of the line.

In the most favorable case, when the consumer processor reads the datum, it finds it in its cache and a miss is avoided. There are, however, three less favorable cases: the consumer processor requests the data before the forward reaches its cache, the forwarded data is displaced from the consumer cache before being used, and the forwarded data displaces useful data from the consumer cache. In practice, the first case cannot occur in our applications because a producer and a consumer access are always separated by a synchronization. However, forwarding can slow down synchronization because a release cannot complete until all pending transactions, including forwards, are completed. Overall, therefore, forwarding can speed up applications by removing coherence misses and cold misses on data that have been written by other processors. However, forwarding can slow down applications by adding extra instructions, inducing extra traffic and misses, and slowing down synchronization.

In this section, we describe the architectural support for this type of forwarding: the primitives (Section 2.1), the base (Section 2.2) and advanced (Section 2.3) microarchitecture support, and the application to physical processors (Section 2.4).

2.1 Forwarding Primitives

The insight that we gained from doing a complete compiler implementation suggested the use of a four-primitive forwarding framework.

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. We do not support multicast to a subset of processors because it was found that most producer writes have only one consumer [12].

Write-Broadcast. The producer processor updates its cache, the memory, and the caches of all processors. This primitive should be used when all the processors will consume the data.

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. This primitive should be used when there is a strong indication that the consumers are the past sharers.

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. This primitive should be used when the others are not suitable.

2.2 Base Microarchitecture

To support these primitives in a directory-based scalable shared-memory multiprocessor with multi-word cache lines we need four new assembly instructions and some modest changes to the cache coherence protocol, write buffers, and caches. Specifically, each of the primitives can be implemented with a different assembly instruction that is inserted in lieu of the producer write instruction. These forwarding instructions behave like an ordinary write with respect to the memory consistency model. Since, in this paper, we consider a release-consistent memory system, the forwarding instructions do not stall the processor. Like ordinary writes, forwarding instructions can only be performed after all previous acquires are performed, and must be performed before any subsequent release is allowed to perform. These forwarding instructions take as argument the address of the data to forward. In addition, 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. The consumer processor will allocate the forwarded line in its cache in the same way that it accommodates any requested line. If, however, it has to evict a dirty line and the write back buffer is full, the forwarded line is not allocated to avoid a potential deadlock. Finally, 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 only other protocol change needed to support the other primitives is that the directory needs to know the type of forwarding primitive and send the appropriate combination of invalidations and updates.

The support in the write buffer is that it needs to be deeper. This is because it needs to keep an entry for each pending forward. Finally, the second level caches need to be designed to accept cache lines that were not requested and may not even be there, namely the forwarded lines. Note that we push the data to the second-level caches only. The updates are not propagated into the first-level cache unless the line is already there, for fear of displacing good data.

2.3 Advanced Microarchitecture

Based on our experiments, we also add the following extra hardware to make forwarding work better.

Forward Delay & Local Combining. Often, a producer forwards different words of the same cache line to the same consumer in a short interval of time. This may occur, for example, when the producer walks down an array of data (Figure 1-(a)). If we combine these forwarding messages into one in a coalescing write buffer, we will reduce the utilization of the directory, network, and caches. More crucially, we will reduce the number of write buffer entries used and, therefore, the chances that the processor stalls due to write-buffer overflow.

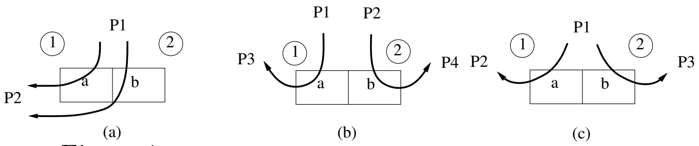


Figure 1: Combining forwards. In the figure, arrows represent forwards, and the encircled numbers are the order of the operations.

To enable this optimization, 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 is similar to the approach used by Glasco et al [8] for update-based coherence protocols. If the other forward is to a different consumer, the forwards cannot be combined and the second forward will be completed after the first one. In practice, a single delay value might not deliver the best performance for all applications. We select a delay that maximizes the average performance gain.

Directory Combining. Another common pattern is what we call *Consumer False Sharing*. This occurs when two consumer processors are forwarded different parts of a cache line. The forwarded data can come from different producers (Figure 1-(b)) or from a single producer (Figure 1-(c)). The problem is that the second forward may invalidate the line in the cache of the first consumer processor before the latter has consumed it.

To solve this problem, we need to detect, in the directory, a stream of forwards of the same line to different consumers. Then, we need to ensure that these forwards do not invalidate the current sharers but, instead, update them. This can be easily done 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 accessing the same line as a previous forward. Therefore, the forward is part of a stream of forwards as indicated before and, instead of invalidating all processors other that the consumer one, the directory will update all current sharers too. This will eliminate any misses resulting from consumer false sharing. The U bit will remain set as long as the accesses to that directory entry are initiated by any of the four forwarding primitives. Once a regular read or write reaches the directory entry, the U bit is reset. This is because such an access is often an indication that the clean producer-consumer behavior is over.

Memory Forwarding. For some applications, the second level cache may be too small and many of the forwards may be displaced before they are used. Worse yet, the forwards may displace the current working set of the application. Hence, we consider the effect of pushing the data to the processor's memory module only. This is possible in COMA machines, where memory modules are organized as caches. Of course, the forward propagates to the caches if they cache the line.

No Home-Processor Updating. In our protocol description, the memory in the home node gets a copy of the forwarded data every time that a line is forwarded. This will help reduce latencies in cases of consumer misprediction or displacement from the consumer's memory hierarchy. However, in COMA machines, we have the option of not leaving a copy of the forwarded line in the home memory. Not leaving a copy there may avoid the displacement of useful data.

2.4 Forwarding to Physical Processors

The identity of the consumer processor in a write-forward instruction is a *logical* processor number between 0 and the number of threads in the parallel application minus one. This logical processor number needs to be translated at run-time into the number of the physical processor executing the consumer thread. The translation needs to be totally hidden from the application. It can be done either in the producer node or in the node that owns the directory entry for the forwarded data. This is because, in a forwarding transaction, the data is first delivered to the directory before being sent to the consumer processor. In this discussion, however, we focus on the case where the directory performs the translation.

Figure 2 shows the microarchitecture required: a register in each processor called the *forwarding program identifier* (*fpid*) register. The *fpid* register can only be set in supervisor mode. It holds a unique number assigned by the operating system to each parallel program that uses data forwarding. This number is the same for all the processes in the same program. The *fpid* register is saved and restored by the operating system during context switches. The number in the *fpid* register is automatically appended to each forwarding message issued by a processor.

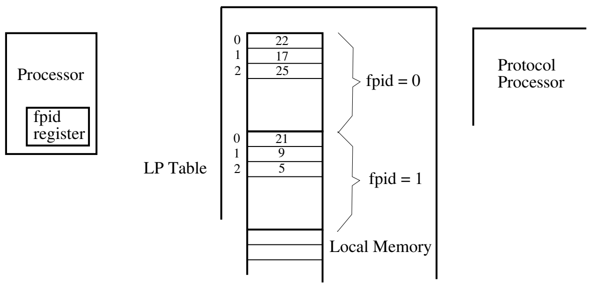


Figure 2: Per-node support needed to apply forwarding to physical processors.

The operating system keeps a software data structure called the logical-to-physical (*LP*) processor translation table. A copy of this table exists in the memory of each node so that the protocol processor in each node can perform a fast local lookup. The table contains a subtable for each possible value of the *fpid*. Each subtable records the mapping of logical to physical processors for a particular program (Figure 2). The operating system modifies the *LP* table on program creation and termination, and on process creation, *migration*, and termination. In a CC-NUMA machine, the copies of this table must be kept consistent by the operating system; in a COMA machine, the hardware keeps all the copies up to date.

When a directory receives a write-forward message, the protocol processor uses the *fpid* in the message and the logical consumer processor number in the message to index its *LP* table. The number retrieved from the table is the ID of the physical processor to forward the data to. This approach transparently solves the problem of mapping logical to physical processors. It can handle process migration and programs with more or fewer processes than physical processors. In the remainder of this paper, all references to processors refer to logical processors.

3 Compiler Algorithm

To insert the forwarding primitives automatically in realistic codes, we have added a pass to the *Polaris* parallelizing compiler [4]. Our pass has two phases. The first one identifies write-read pairs that have a producer-consumer relationship. For each of these pairs, the second phase chooses and inserts in the code the best forwarding primitive. In the following, we describe Polaris, the two phases, and then briefly compare our algorithm to message-passing systems.

3.1 The Polaris Parallelizing Compiler

Polaris is a parallelizing compiler developed at the University of Illinois. It 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. Examples of directives are `DOALL`, `SHARED`, `PRIVATE`, `FIRSTVALUE`, and `RANGEWRITTEN`. `DOALL` means that the loop is parallel. In each loop, `SHARED` and `PRIVATE` identify shared and private data respectively, while `RANGEWRITTEN` and `FIRSTVALUE` identify the data being written and read respectively.

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.

In this paper, we use data forwarding to hide communication *between* different serial sections or fully-parallel loops. We do not consider forwarding within *doacross* loops because they are relatively less frequent. Nevertheless, our algorithms could be easily extended to support doacross loops too.

3.2 Finding Producer-Consumer Pairs

The first phase of our forwarding pass identifies the pairs of statements that produce and consume the same datum. In our analysis, we focus only on accesses to arrays because scalars have little weight in the codes considered. In addition, since it has been suggested that it is necessary to analyze the codes interprocedurally [12], and Polaris lacks good interprocedural analysis, we inline the programs. The steps in this phase are as follows:

1. Building the Data Flow Graph. We first build a data structure called the data flow graph (DFG). Each node in the DFG corresponds to a serial or a parallel section of the program. Consecutive serial statements get merged into one serial section. Each edge in the graph corresponds to the flow between one section and another. Figure 3 shows the DFG for a code segment from SPEC95's *tomcatv*.

```
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
```

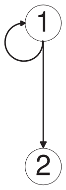


Figure 3: A code segment and its data flow graph.

2. Array Sections Written and Read. We now apply a Polaris pass that returns the array sections that are read and those that are written in each DFG node. The array sections that are read appear in the `FIRSTVALUE` directive, while those that are written appear in the `RANGEWRITTEN` one. For example, the code in Figure 3 now becomes that in Figure 4. The array sections are specified with the lower bound, upper bound, and stride of the accesses. For instance, `RANGEWRITTEN rx(a:b:c, d)` means that the `d`th column of array `rx` is written with stride `c` from row `a` to row `b`. Note that, while the figure shows the directives for the parallel loops only, serial sections also have them.

```
      DO j = 1, n-3, 1
        DOALL i = 2, n-1, 1
CSRD$ FIRSTVALUE ...
CSRD$ RANGEWRITTEN rx(2:n-1:1, n-j-1), ...
          rx(i, n-j-1) = ...
        ENDDO
      ENDDO

      DOALL j = 2, n-1, 1
CSRD$ FIRSTVALUE rx(2:n-1:1, 2:n-1:1), ...
CSRD$ RANGEWRITTEN ...
        DO i = 2, n-1, 1
          ...  = rx(i, j) ...
        ENDDO
      ENDDO
```

Figure 4: Annotations with the sections read and written.

In this step, the effectiveness of array section analysis in Polaris is of vital importance. The analysis is enhanced with new techniques to handle conditionally defined and used variables and to analyze the symbolic expressions in the array sections. These techniques use the gated single assignment form of a program to perform conditional data flow analysis and demand-driven symbolic analysis [22].

3. Data flow Analysis. This step uses standard data flow algorithms to compute the set of definitions that reach each node in our DFG. For each node, we intersect the reaching definitions with the array sections that the node uses (as given in the `FIRSTVALUE` for the node). If such an intersection is not empty, we have found some 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 our example, a pair is composed of reference "`rx(i, n-j-1) =`" and reference "`= rx(i,j)`".

3.3 Instruction Selection

For each producer-consumer pair that we identify, the second phase of our pass selects and inserts the best forwarding primitive. In our analysis, we assume that serial sections are always executed by the same processor, namely logical processor 0. Our algorithm proceeds as follows.

1. Write-forward to processor 0. If the producer write is in a parallel section and the consumer read in a serial section, we use write-forward to processor 0.

2. Write-broadcast. If, instead, the consumer read is in a parallel section, we examine its index expression ("i, j" in Figure 4). If the expression is independent of the consumer loop's index variable ("j" in Figure 4), then all processors will read the same data. In that case, we use write-broadcast in the producer point.

3. Write-forward. If the index expression in the consumer read depends on the index variable of the consumer loop, it is possible that different processors read different data. In that case, we try to use write-forward. We want to determine the consumer processor for each iteration of the producer loop.

We proceed in two steps. First, we try to determine, for each iteration in the producer loop, what iteration in the consumer loop consumes its data. This is done by solving the equation:

Index expression in the consumer read = Index expression in the producer write

If the equation is beyond the capabilities of our compiler, as explained in Section 3.3.1, we give up on write-forward. The second step is to determine, for each iteration in the consumer loop, what processor executes it. This is possible only if we know the schedule of the consumer loop. With these two steps, we can determine the consumer processor for each iteration in the producer loop and, therefore, can generate the forward expression. Section 3.3.1 gives an example of the complete process.

4. Write-update. If we could not use write-forward, we try to use write-update. We use it when the compiler finds a repeated consumer pattern: a serial loop enclosing a producer and a consumer loop, such that the index expression in the consumer read is independent of the index variable of the enclosing serial loop. In this case, the same processor consumes the same array location across iterations of the serial loop. If, in addition, the index expression in the producer write is also independent of the serial loop's index variable, the update messages flow between the same pair of producer-consumer processors in every iteration of the serial loop. Otherwise, this scheme may effectively degenerate into a situation close to broadcasting.

5. Write-through. Finally, if none of the previous conditions is satisfied, we use write-through to write the data to memory immediately. This will reduce the penalty of the next consumer read.

In our analysis, we introduce an optimization to reduce unnecessary forwarding overhead. Since we know the schedule of producer and consumer loops, if a processor is producing data for only itself to consume, we cancel the producer-consumer pair.

Overall, we can compare our primitives in terms of compiler support complexity and instruction overhead. For write-forward, the compiler needs to determine the consumer identity. This is harder than to determine that all processors consume (write-broadcast) or that there is data reuse (write-update). Write-through needs no extra compiler support. Regarding instruction overhead, only write-forward involves adding extra instructions.

3.3.1 Generating the ID of the Consumer Processor

A hard part of the algorithm is to determine the ID of the consumer processor in the write-forward primitive. We now show how to do it for the example in Figure 4. In the example, the consumer read induces parallel accesses along the second dimension of array `rx`. Consequently, in the first step of our algorithm, we build an equation based on the index expressions in the second dimension of array `rx` for the producer and consumer accesses. If we denote the indices in the producer and consumer loops as j^p and j^c respectively, the producer and consumer iterations that access the same array entry are related by:

$$j^c = n - j^p - 1 \tag{1}$$

We need to express the index of the consumer loop as a function of the index of the producer loop. In the case of Equation 1, this is already done. The equation says that iteration j^p in the producer loop produces data consumed by iteration $n-j^p-1$ in the consumer loop. In general, however, to get to this point, we need to perform several transformations. In our analysis, we give up on write-forwarding if the transformations are too complicated. Specifically, we only handle the cases where the left side of Equation 1 is a linear function of the consumer loop index, with the linear coefficient and the independent term being invariant expressions (a function of constant and loop-invariant variables). This analysis is performed on all the dimensions of the shared array that are dependent on the consumer loop index. If the equations do not all have the same solution, then we give up on write-forward.

The second step is to determine, for each iteration in the consumer loop, what processor executes it. This is possible only if we know the schedule of the consumer loop. How we proceed depends on the type of schedule used. There are several policies, including round-robin, chunk, and block scheduling [11]. For each type of scheduling, the appropriate mapping function is used. To simplify our discussion, let us use block scheduling. In that case, given a loop with N iterations, P processors, and with an index normalized to start at l and to have a step of 1, the *pid* of the processor executing iteration i is

$$pid(i) = \left\lfloor \frac{i-l}{\lceil \frac{N}{P} \rceil} \right\rfloor$$

In the example, the number of iterations in the consumer `DOALL` is $n-2$, the index starts at 2 and the step is 1. Therefore, the scheduling function $pid(j^c)$ is:

$$pid(j^c) = \left\lfloor \frac{j^c - 2}{\lceil (n-2)/P \rceil} \right\rfloor \tag{2}$$

From Equations 1 and 2, the data produced in iteration j^p of the producer loop is consumed by processor

$$\left\lfloor \frac{n - j^p - 3}{\lceil (n-2)/P \rceil} \right\rfloor \tag{3}$$

Therefore, in Figure 3, we replace the regular write to `rx(i, n-j-1)` by a write-forward whose target processor is given by Equation 3. We indicate this by adding the `FORWARD` directive to the source code. Figure 5 shows the resulting code. The floor function is not needed in integer division, while the ceiling of `(n-2)/P` is `(n-3)/P+1` with integer division. Since the forwarding expression is loop invariant with respect to the innermost loop, we can move the code that evaluates the expression out of the innermost loop.

```
      DO j = 1, n-3, 1
        DOALL i = 2, n-1, 1
CSRD$     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
```

Figure 5: Code after adding forwarding.

In general, depending on the producer and consumer loop bounds, we may be forwarding data that no processor will consume. If, after examining the loop bounds, we conclude that this is happening, we can do two things to avoid unnecessary forwards. We can either enclose the write-forward in a conditional statement or, alternatively, we can split or peel the producer loop to separate the forwarding from the non-forwarding sections of the code.

3.3.2 Merging of Forwarding Primitives

Before generating the final code, the compiler must handle the case where an access belongs to several producer-consumer pairs. Figure 6-(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 write.

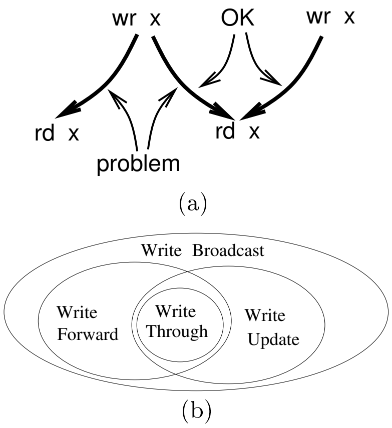


Figure 6: Merging of forwarding primitives.

When this occurs, we select the most appropriate primitive based on the hierarchy of Figure 6-(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. As another example, the write-broadcast is preferred over any other primitive. This is because its action, namely updating memory and all caches, includes the actions performed by the other primitives.

When we have a write-forward and either another write-forward with different target processor or a write-update, we choose one of the two based on which one is more likely to be useful. To determine the most useful one, we assign frequency weights to the different edges of the DFG graph (Section 3.2). The weights are assigned using the 90/50 rule, which assumes that backward branches are taken 90% of the time and forward branches 50%. We then select the primitive assigned to the highest-weighted edge. In the future, we plan to add more complex static or dynamic profiling information. Note that the type of dynamic profiling needed would be limited to the frequency of transitions between sections, not the taken frequency of all branches. Overall, however, even if we mispredict, the cost will only be a miss satisfied by the home memory. The reason is that all primitives update memory.

3.4 Comparison to Message-Passing

Most compilers for message-passing multiprocessors are based on the *owner-computes* rule [3, 10, 23]. In the owner-computes paradigm, a processor owns the fraction of the data that is allocated in its local memory. Each processor executes the operations that write to data it owns. The operands of these operations, if they are not owned by the processor, are sent-in by the owners using messages.

One difference between compiler support for forwarding and for message-passing is in the algorithm used to compute the places where communication is needed. In a message-passing compiler, communication is needed in a statement if, given the program data distribution, a processor does not own the data it needs. Regardless of when or where the data was last written, the owner processor has to send the current copy of the data. In forwarding, instead, communication is given by the data flow in the program. We identify places where the data is written and read, and use data flow techniques and loop scheduling information to determine when data needs to be transferred.

A second difference is that forwarding addresses the tradeoff between compiler analysis complexity and performance differently. The reason is that forwarding does not need to maintain the correctness of the program; the underlying cache coherence protocol guarantees it.

4 Simulation Environment

To evaluate our compiler algorithm, we use TangoLite [9] execution-driven simulations of applications parallelized by Polaris and annotated by our forwarding pass. The architecture model that we simulate is a scalable Flat COMA multiprocessor [21] with 32 processors cycling at 300 MHz under the release consistency model. Each processor has a two-level cache hierarchy. The first-level cache is direct-mapped, has a size of 16 Kbytes, and has a line size of 32 bytes. The second-level cache is 4-way set-associative, varies in size from 32 to 512 Kbytes depending on the application, and has a line size of 32 bytes. We justify the choice of these small cache sizes in the next few paragraphs. The write buffer between the caches and between the second-level cache and memory has a depth of 20 entries and a width of a cache line. The attraction memory in each node is 4-way set-associative and has a line size of 32 bytes. Peak memory bandwidth is approximately 400 MBytes/sec. The network is a 2D mesh topology. Table 1 shows the latencies for a round trip from the processor to different levels of the memory hierarchy without contention.

Level	Latency (Cycles)
First-level cache	1
Second-level cache	10
Local attraction memory (AM)	70
Remote AM (Data in home memory)	$\approx$ 400
Remote AM (Data in third node)	$\approx$ 600

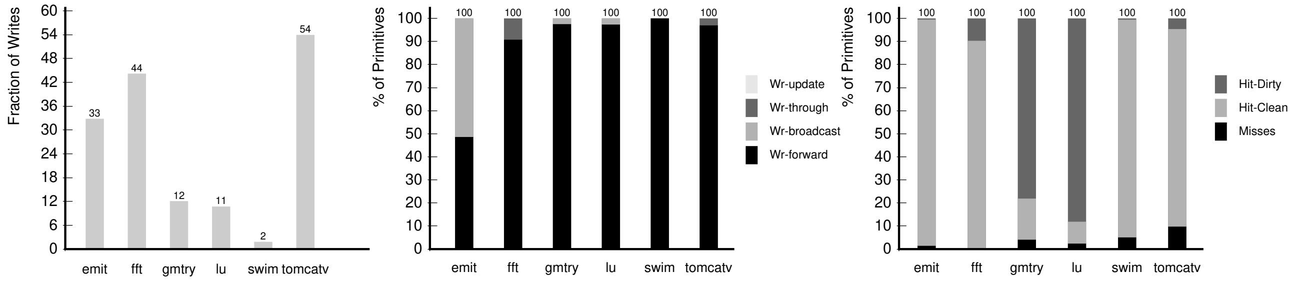
Table 1: Latencies of contention-free round-trip accesses to different levels of the memory hierarchy.

The ratio of the application's total data size to the total memory in the machine is called the memory pressure. In our experiments, we perform two sets of experiments, namely one with a low memory pressure (40%), and one with a higher one (60%). Cache coherence is maintained using a DASH-like [14] directory-based protocol. The protocol is extended with the new data forwarding transactions described in Section 2. All transactions are modeled in detail. We model a simple protocol processor per node to handle the coherence transactions. It has an occupancy of 30 cycles and a latency of 20 cycles per transaction.

To drive the simulations, we use six Polaris-parallelized Fortran applications from a wide variety of sources to cover as many cases as possible (Table 2). The applications are *emit* and *gmtry* from the NAS benchmark set [2], *swim* and *tomcatv* from SPEC95, *fft* from the CMU suite [6], and *lu*, a custom-developed LU factorization. For the SPEC95 applications, we

Name	Small L2			Large L2			
	Cache (Kbytes)	Miss Rate (%)	Conflict Misses (%)	Cache (Kbytes)	Miss Rate (%)	Conflict Misses (%)	Speedup for 32 Proc.
emit	32	3.7	39	128	2.3	5	5.0
fft	64	6.3	61	256	5.4	54	3.9
gmtry	32	1.6	48	128	1.0	6	14.1
lu	32	2.2	10	256	1.3	2	12.9
swim	64	4.1	94	512	1.2	60	16.8
tomcatv	64	4.5	46	512	2.6	6	12.3

Table 2: Applications studied.



(a) (b) (c)

Figure 7: Some dynamic characteristics of forwarding.

use the test data set. For the applications considered, the test and the reference data sets differ only in the number of iterations executed, not in the data set size. Consequently, the results of our experiments are largely the same for both data sets. As explained in Section 3.2, Polaris limitations forced us to use inlined programs. For this reason, we did not evaluate the other SPEC95 applications: their code size is so large that the compiler fails to inline them. Finally, we schedule the loops in the applications statically using block scheduling. This is a good schedule because, in these applications, loops tend to be load-balanced and there is substantial reuse of data across same-numbered blocks in different loops.

We use relatively small caches in the simulations for two reasons. First, because we run small applications, their working sets tend to be small, making larger caches unnecessary. Second, large caches will tend to underestimate any cache pollution caused by forwarding, so it is interesting to look at smaller cache sizes. 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 are shown in Table 2. For a given application, we find these sizes by measuring the miss rate as we vary the size of the cache [11]. The table also shows, for each data cache, the miss rate and the fraction of the misses that are caused by conflicts. Finally, the table shows the speedup of these compiler-parallelized applications for 32 processors and the architecture with large L2 caches. All the data in the table corresponds to the low memory pressure.

5 Experimental Results

We evaluate forwarding under several environments. First, we consider an environment with the large L2 caches and the low memory pressure, where the application runs alone in the machine, and there is no process migration. This environment we call baseline. Then, we consider environments where these conditions change. In our discussion, we provide an upper bound for the performance of any data forwarding algorithm that forwards to second-level caches. In this upper bound, called *Ideal*, all L1-missing reads that consume data produced by another processor, including those not identified by our compiler pass, magically find the data requested in the second-level cache.

5.1 Characterizing Forwarding

To understand the effectiveness of forwarding, we examine three dynamic parameters, namely the fraction of writes that are replaced by forwarding primitives, the type of forwarding primitive used, and the state of the line to be forwarded in the producer cache. The fraction of writes that are replaced by a forwarding primitive is shown in Figure 7-(a). In the figure, we include all the writes in the application. As we can see, this fraction varies from 2% to 54%. However, we find no correlation between this number and the speedup of an application due to forwarding.

Figure 7-(b) shows the type of forwarding primitives used. In all applications except one, the large majority of uses are for write-forwards. Only a few write-broadcasts are used. The exception is *emit*, where all processors often consume the same data. Finally, there are a few write-throughs, and practically no write-updates at all. This data shows that restricting the analysis for write-forwards to linear functions (Section 3.3.1) is not a serious limitation in our applications: in only a few cases we have to give up on write-forwarding and fall back to write-through.

Finally, Figure 7-(c) shows the state of the line to be forwarded in the producer cache. In all the applications, the miss rate of these accesses is very small. However, while in most applications, the line to be forwarded is usually in state clean, in *gmtry* and *lu*, the line is usually in state dirty. If the line is dirty, a plain write would be carried out locally and, unlike a forwarding primitive, not use the write buffer. Consequently, we can expect that, in *gmtry* and *lu*, forwarding will put more pressure on the write buffers.

5.2 Baseline Environment

Figure 8 shows the execution time of each application under the baseline environment. *Base* is the no-forwarding architecture. *Fwd* is *Base* plus forwarding. *Fwd-L* is *Fwd* plus local forward combining with a delay of 100 cycles. This delay delivers the best average performance gain for our applica-

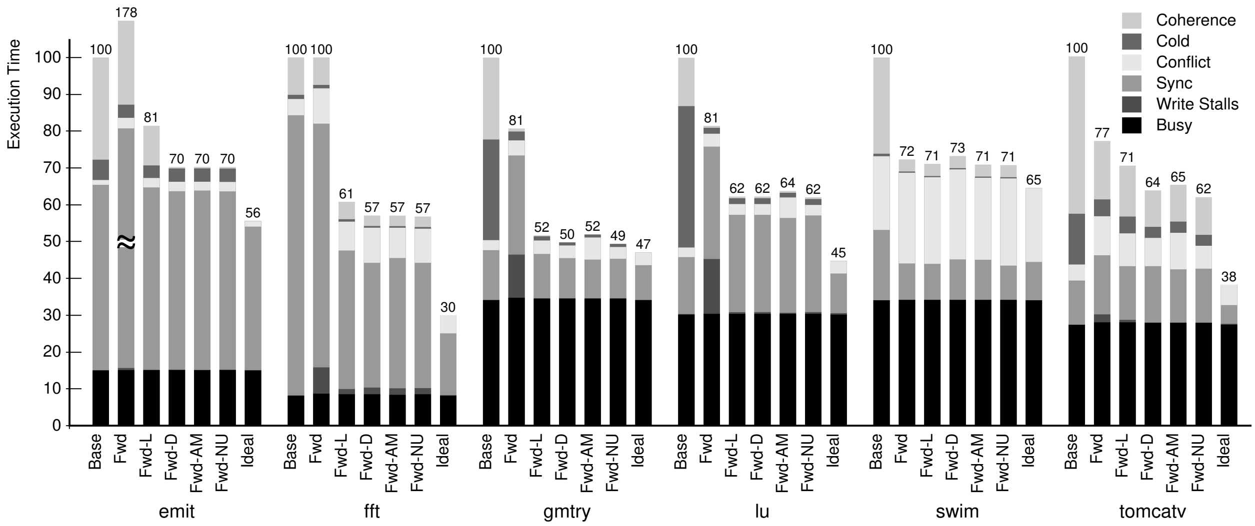


Figure 8: Impact of forwarding and its optimizations on the execution time for the baseline environment.

tions [11]. *Fwd-D* is *Fwd-L* plus directory combining. *Fwd-AM* is *Fwd-D* except that the forwarded data is left in the attraction memories instead of being pushed to the second-level caches. *Fwd-NU* is *Fwd-D* except that the home node does not get a copy of the line being forwarded. Finally, *Ideal* is the upper bound on the performance of data forwarding discussed above.

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*). Forwarding targets directly the *Coherence* time plus, in our applications, all the *Cold* time because all cold reads access data initialized by another processor.

The figure shows that *Fwd* reduces the execution time of four applications by only a modest 20-30%. In addition, *fft* does not change and *emit* runs significantly slower. These disappointing results are despite the fact that the *Coherence* and *Cold* times often decrease significantly and that the instruction overhead of forwarding only induces a negligible 1% increase in the *Busy* time of the applications. There are three reasons for the poor results. The first one is that *Fwd* induces some write stall time. This is especially noticeable in *gmtry* and *lu*, the two applications that, as indicated in Section 5.1, put more extra pressure on the write buffer.

The second reason is that *Sync* increases in many applications. What causes this is that the processors that stall because of write buffer overflow force the faster processors to wait longer in barriers. In addition, if many forwards are issued right before a lock release, the latter is likely to get delayed.

The third reason for the poor results is that forwarding fails to eliminate much of the *Coherence* time in applications like *emit* and *tomcatv*. In *emit*, the problem is consumer false sharing (Section 2.3), while in *tomcatv*, the problem is that some data is consumed by a subset of processors. Since we do not have multicast, we use write-forwarding to a single processor.

We now apply the forwarding optimizations. Local forward combining (*Fwd-L*) eliminates nearly all the write stalls. As a result, the synchronization time also decreases. In addition, combining also generally decreases the usage of network and directory. As a result, *Coherence*, *Conflict*, *Cold*, and synchronization time tend to decrease in all applications. The applications now run 33% faster than *Base* on average. In [11], we examine the impact of different combining delay periods. We conclude that the best design point is about 100 cycles.

If we add combining in the directory (*Fwd-D*), we eliminate consumer false sharing. This type of pattern is present in *emit* and, to a lesser extent, in *tomcatv*. As a result, *emit* and *tomcatv* run 14% and 10% faster respectively. Although aside from these two applications the changes are minimal, we recommend this optimization because it is cheap. Finally, the last two optimizations (*Fwd-AM* and *Fwd-NU*) have only a marginal impact. *Fwd-NU* is slightly better than the rest. Overall, with *Fwd-NU*, the applications run an average of 40% faster than in *Base*. The average application speedup with 32 processors has changed from 11 in *Base* (Table 2) to 18.

Even after all our optimizations are applied, there still remains some *Coherence* time, most notably in *tomcatv*. In addition, there is still some room for improvement to get close to the *Ideal* scenario, which is about 50% faster than *Base*. The major limitations of our scheme are the failure to completely eliminate the clustering of forwards before releases and the lack of a multicast forward primitive (*tomcatv*). A lesser problem is the inability to use write-forwards with non-linear expressions.

5.3 Varying Cache Size and Memory Pressure

Figure 9 presents the results of the previous experiments with the small L2 caches listed in Table 2. While conflict misses accounted for 22% of the misses in the large caches, they now account for 50%. As a result, *Conflict* time is relatively more important. Additionally, forwarding sometimes increases the amount of *Conflict* time. However, the impact of the different forwarding optimizations follows similar trends as in Figure 8. Furthermore, since *Conflict* does not account for most of the execution time of the applications, the absolute impact of forwarding does not change much: *Fwd-NU* and *Ideal* speed up the applications by an average of 35% and 50%, respectively.

We also repeat the experiments with the baseline environment with the higher memory pressure of 60%, as discussed in Section 4. The result is an increase in the relative contribution of the *Conflict* time in the applications due to memory conflicts. We omit the results here, however, because the impact of the different forwarding optimizations follows similar trends as in Figure 8. *Fwd-NU* speeds up the applications by an average of 30%, while *Ideal* does it by 50% [11].

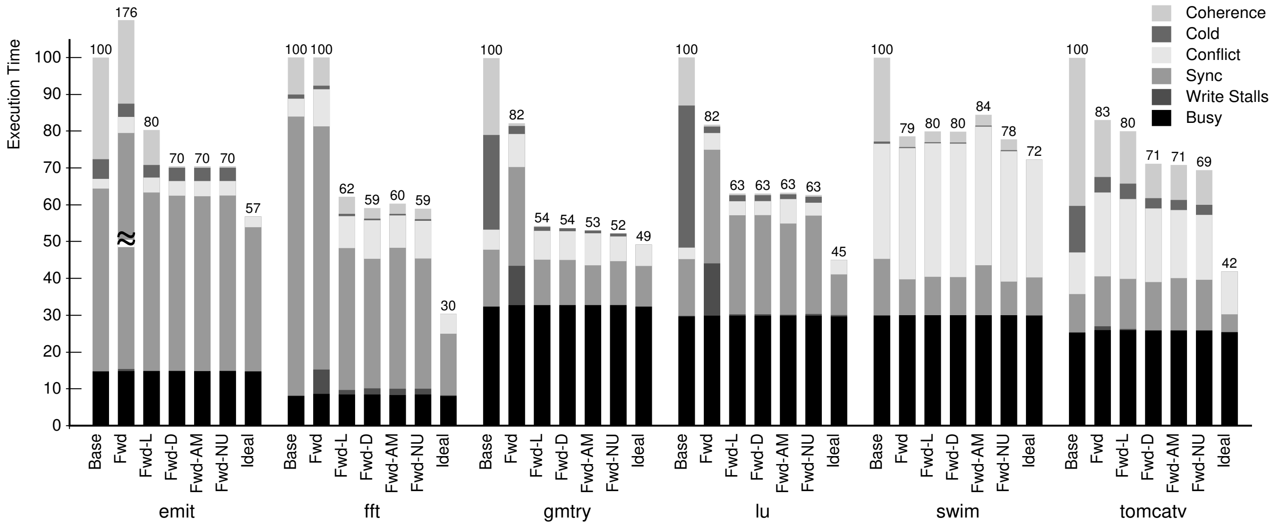


Figure 9: Impact of forwarding and its optimizations on the execution time for the baseline environment with small L2 caches.

5.4 Context Switching and Process Migration

Finally, we estimate the impact of forwarding when an application time-shares the machine with other applications and when the processes of the application migrate between processors. To model time-sharing without migration, we assume that, when a process is preempted, all its cache state is lost. This includes lines that are present in its cache and lines that are being forwarded to the process. We assume, however, that the attraction memory is large enough to maintain the state. Process migration is worse in that the preempted process can reuse neither the state in its caches nor the state in its attraction memory. Note, however, that the hardware described in Section 2.4 ensures the delivery of forwards across process migration.

For these experiments, we simulate the baseline architecture of Section 5.2 and add periodic state invalidations. To emulate context switching (*BaseSwi*), we invalidate the caches every 10 ms. To emulate context switching with process migration (*BaseMig*), we invalidate the caches and reschedule the process on a different node every 10 ms. We repeat these experiments under *Fwd-NU* forwarding, additionally canceling in-flight forwards when the context switches occur. These environments are called *Fwd-NUSwi* and *Fwd-NUMig*. Notice that this model mimics a gang-scheduled system. We choose a small time quantum of 10 ms to compensate for the small caches that we use. This means that, on average, each application suffers 15 context switches during its execution.

Figure 10-(a) shows the execution time of *BaseSwi* and *Fwd-NUSwi*, while Figure 10-(b) does it for *BaseMig* and *Fwd-NUMig*. In the figures, the execution times are normalized to *BaseSwi* and *BaseMig* respectively. Note that, in the environments with process migration (*BaseMig* and *Fwd-NUMig*), the *Cold* time is larger than before. The reason is that, when a process gets rescheduled after a migration, first-time misses on data that it already accessed before the migration should be counted as *Cold* misses. Forwarding, of course, can eliminate these *Cold* misses too.

From the figures, we see that, in both environments, forwarding is still able to speed up the applications significantly. The periodic loss of state affects both the forwarding and the non-forwarding environments, making the relative impact of forwarding only slightly worse than in the baseline environment of Section 5.2. While, in that environment, applications were sped up by an average of 40%, under context switching they are sped up by an average of 36%, and under context switching plus process migration by an average of 31%. More discussion can be found in [11].

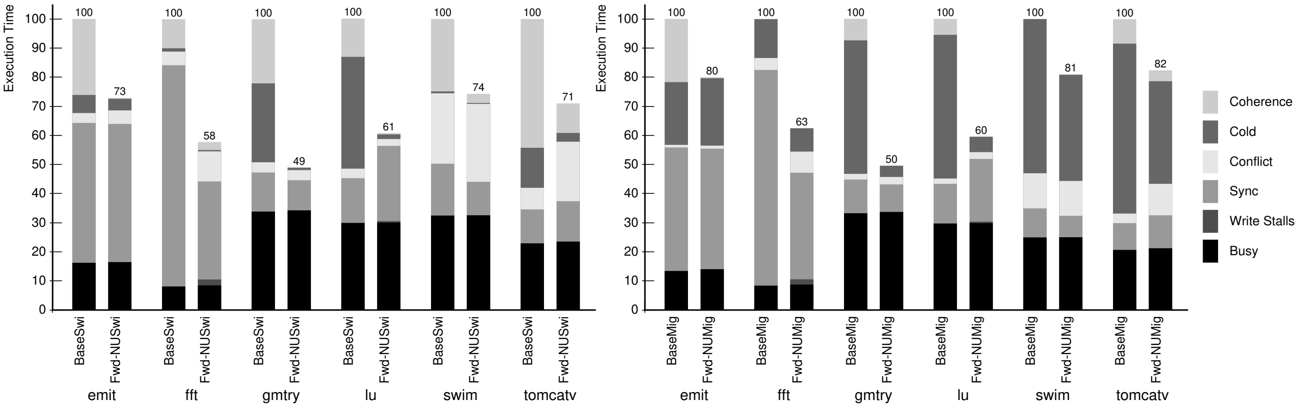
5.5 Limitations of the Algorithm

In the previous sections, we have shown that the forwarding primitives speed up applications by an average of 40%. However, in several applications, the *Ideal* execution time is much smaller than what we can do with forwarding. Some reasons for this difference are that our forwarding techniques suffer from the clustering of forwards before releases and the lack of multicast.

There are, however, some other things that we can do in our compiler algorithm to improve the performance of our data forwarding. First, the computation of array sections that are read and written could be improved and integrated seamlessly with our pass. Much of the information collected by this pass, including the source statement, is lost. Second, data flow analysis produced many uninteresting pairs due to small or overlapping sections being propagated. This can also be improved. Finally, the lack of interprocedural analysis forced us to inline programs. Interprocedural analysis techniques need to be modified and added.

6 Concluding Remarks

The main contribution of this paper has been the complete implementation of a data forwarding pass in the Polaris parallelizing compiler. Our new framework includes four primitives, namely *write-forward*, *write-broadcast*, *write-update*, and *write-through*, and new microarchitectural support. We evaluated forwarding with simulations for different cache size, memory size, machine sharing, and process migration parameters. We concluded that forwarding delivers large speedups: six 32-processor applications ran an average of 40% faster. In addition, forwarding gets close to the upper bound in performance, and needs compiler support of only medium complexity. To extract the remaining performance, it is necessary to improve on two areas: preventing the clustering of forwards before lock releases, and supporting some form of multicast.



(a) (b)

Figure 10: Estimating the impact of forwarding on the execution time of the baseline environment with context switching (Chart (a)), and context switching plus process migration (Chart (b)).

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*, 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] P. Banerjee, J. A. Chandy, M. Gupta, J.G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multiprocessors. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, December 1994.

[4] 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.

[5] 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.

[6] 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, Carnegie-Mellon University, March 1994.

[7] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, October 1996.

[8] D. Glasco, B. Delagi, and M. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the 27st Annual Hawaii International Conference of System Sciences*, pages 543–545, January 1994.

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

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] D. Koufaty. *Hiding Communication-Induced Misses: Data Forwarding and Prefetching*. PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign, October 1997.

[12] 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.

[13] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs. In *Proceedings of the International Conference on Supercomputing*, pages 53–60, July 1998.

[14] 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.

[15] T. C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching. Ph.D. Thesis, Stanford University, March 1994.

[16] M. Ohara. Producer-Oriented versus Consumer-Oriented Prefetching: A Comparison and Analysis of Parallel Application Programs. Technical Report CSL-TR-96-695, Computer Systems Laboratory, Stanford University, June 1996.

[17] 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.

[18] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Sigla, and I. Yanasak. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. In *Proceedings of Supercomputing 1995*, December 1995.

[19] 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.

[20] A. Sivasubramaniam. Reducing the Communication Overhead of Dynamic Applications on Shared Memory Multiprocessors. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 194–203, February 1997.

[21] 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.

[22] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign, May 1995.

[23] H. P. Zima and B. M. Chapman. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.