On the Use and Performance of Explicit Communication Primitives in Cache-coherent Multiprocessor Systems

Xiaohan Qin and Jean-Loup Baer
Department of Computer Science and Engineering, Box 352360
University of Washington, Seattle, Wa 98195-2350
{baer,xqin}@cs.washington.edu

July 16, 1996

Abstract

Recent developments in shared-memory multiprocessor systems advocate using off-the-shelf hardware to provide basic communication mechanisms and using software to implement cache coherence policies. The exposure of communication mechanisms to software opens many opportunities for enhancing application performance.

In this paper we propose a set of communication primitives that are absent from pure cache coherent schemes. The communication primitives, implemented on a communication co-processor, introduce a flavor of message passing and permit protocol optimization, without sacrificing the simplicity of the shared memory systems.

To assess the overhead of the software implementation of the primitives and protocols, we compare, via simulation, the execution of three programs from the SPLASH-2 suite on four environments: a PRAM model, a hardware cache coherence scheme, a software scheme implementing only the basic cache coherence protocol, and an optimized software solution supporting the additional communication primitives and running with applications annotated with those primitives. With the parameters we chose for the communication processor, the overall memory system overhead of the basic software scheme is at least 50% higher than that of the hardware implementation. With the adequate insertion of the communication primitives, the optimized software solution has a performance comparable to that of the hardware scheme.

These results show that the trend towards software-controlled cache coherence is justified since there is no loss in performance and the software solution is more flexible and scalable.

Keywords: communication primitives, software-controlled cache coherence, performance evaluation, shared-memory multiprocessors

1 Introduction

The performance of parallel processing systems is highly dependent on the communication to computation ratio. Some of the communication costs are intrinsic to the algorithm, for example barrier synchronization or producer-consumer relationships. Other communication costs are directly linked to technology, e.g., memory latency or interconnect bandwidth. In addition, there are some components of the communication costs that depend on the policies chosen for data exchange between processes, and on the mechanisms used to implement these policies.

Our focus is on cache-coherent shared-memory multiprocessor systems (CC-NUMA) and on the communication costs incurred in such systems. The vast majority of shared-memory multiprocessors use a cache coherence policy implemented in hardware. Most optimizations geared at reducing communication costs, such as improvements in the protocols, weak memory models, and stream buffer prefetching among others, require non-trivial hardware modifications. The flexibility and the scalability of both the hardware cache coherence policies and of their optimizations are open to question.

All cache coherence policies rely on a basic communication primitive, namely, a high performance message passing substrate optimized for small messages. These messages are either control messages e.g., invalidate, or messages to transmit a small amount of data, a cache line. The optimizations that we listed above either reduce the number of messages (improvements in protocols) or schedule their occurrences so that they can overlap with computation (weak memory models, prefetching). Nonetheless, the rigidity of the hardware implementations prevents the policies and optimizations to be adapted to the needs of a given application. A separation of the coherence policy and of the communication primitives to support it requires a set of communication and memory-system mechanisms that can be used at run-time to implement application-specific coherence policies. The usage of such primitives can be directed by the user or the compiler [7], or can be detected by some monitoring device. The implementation of these primitives requires some programmable network interface [22].

In this paper, we propose a set of communication primitives that will allow the user to take advantage of features common in message-passing systems. More precisely, we will provide the user with primitives such as transfer of multiple cache lines, multicast of cache lines, prefetch, and post-store in either cache or memory that remove some of the drawbacks of the cache coherence mechanisms. In addition some primitives can tailor the cache coherence policy to the application. The difference with message-passing systems is that we retain the shared-memory global addressing paradigm and the underlying cache coherency support. Implementation of these primitives affects only the performance of the system, not the correctness of the computations. The software implementation of these communication primitives will result in an additional overhead. The goal of this paper is to quantify this overhead and to show that the resulting performance is competitive with that of a straightforward, but rigid, hardware implementation. At equal performance, we will have gained flexibility, ease of scalability, and enhanced opportunities for further optimizations.

The remainder of the paper is organized as follows: In Section 2, we introduce the communication primitives that we propose and their semantics. In Section 3, we discuss the architectural model of the network interface and highlight a possible software implementation of the communication primitives and cache coherence protocols. Section 4 describes the experimental methodology: selec-

tion of benchmarks, baseline architecture, and simulation environment. In Section 5, we show how the benchmarks were modified with the introduction of the communication primitives and present results of simulation of the proposed scheme and the baseline. Finally, we summarize the related work in Section 6, and conclude the paper in Section 7.

2 Communication Primitives

In this section, we present the communication primitives that can be used by the programmer or compiler in order to enhance the performance of CC-NUMA systems. Under usual cache coherence policies, fetching of data is performed on demand only, i.e., when a cache miss occurs. Similarly, storing of data in memory is done only when a replacement is needed (most of the lower level caches in the memory hierarchy follow a write-back policy). The primitives that we propose extend, under programmer or compiler control, these basic data movement operations. Each primitive is a non-blocking operation and multiple requests can be outstanding.

Primitives	Semantics
$get(addr, size)$	fetch the data into the requesting processor cache
$getex(addr, size)$	fetch the data with ownership
$put(pid, addr, size)$	place the data in the cache of the processor pid
$putex(pid, addr, size)$	transfer the data with ownership to processor pid
$multicast(pids, addr, size)$	disseminate the data to a set of processors
$putmem(addr, size)$	return data to memory
$writemem(mode, addr, size)$	set the write policy for the data

Table 1: Communication primitives. $addr$ is the starting address of a block of data. $size$ is its size in bytes. pid is a processor number. $pids$ is a mask indicating a list of processors. $mode$ is either write-back (wb) or write-through (wt)

The primitives and their semantics are shown in Table 1. The first two, $get(addr, size)$ and $getex(addr, size)$, allow the programmer to fetch, or rather prefetch, a set of consecutive cache lines and to request that these lines be either in *shared* or *exclusive* state. Granting ownership to the receiving processor can significantly reduce write latency in subsequent accesses to that data. Note that the *get* operations are not equivalent to page migration or DMA transfers since the data is transferred in the cache of the requester and cache coherency is maintained.

While the *get* primitives are consumer oriented, the next two primitives, $put(pid, addr, size)$ and $putex(pid, addr, size)$, are producer oriented. These operations, akin to an asynchronous send in message-passing, can be used when the producer knows the identity of the consumer. By using a *put* operation, the producer process can store a set of cache lines, in a given state, in the cache of the processor running the consumer process. When there is more than one consumer and their identities are known, the $multicast(pids, addr, size)$ primitive can be used. When the producer knows that it will not use the data any longer but does not know what process will consume it, the data can be stored in memory with the $putmem(addr, size)$ primitive. This may save half of

the request-reply bandwidth in the network when the data will be used next. A similar effect can be achieved on a word per word basis by changing the default write-back policy to a write-through one. The *writemem* primitives can restrict this policy to a range of addresses, or if the *addr* and *size* are both null, it can be applied to all write misses. By selecting the mode, the user has the choice of alternating between the two write policies.

When used appropriately, these primitives will benefit the application since they allow:

1. Overlap of communication with computation; this overlap can be extensive since the operations dictated by the communication primitives are non-blocking and multiple requests can be outstanding.
2. Bulk data transfers; the network can be better utilized by pipelining transmissions.
3. Tailoring the cache coherence protocols; for example, superfluous data transfers, present in write-update protocols, can be avoided, or early requests for ownership, in write-invalidate protocols, can reduce the number of control messages in migratory-like patterns.

However, there are dangers in using these primitives unwisely. These dangers are the same as those that exist when using prefetching or post-storing too aggressively, namely cache pollution, increase in coherence traffic, and saturation of the network. It is up to the programmer/compiler to make sure that the use of the primitives will be beneficial. Similarly, it is up to the system implementation to determine whether data transfers should actually happen or not. For example, if the system detects that the processor issuing a *get* already has the data in its local cache, it can simply acknowledge the request without sending the data. The system may also further require that the processor issuing *put*, *putex*, or *multicast* have the ownership of the data, otherwise no data will be transferred upon such requests.

In summary, the communication primitives give to the user some advantageous features of asynchronous message-passing while keeping the simplicity and correctness qualities of the cache coherent global address space paradigm.

3 Architectural Model

Figure 1 shows the general architecture of a Flash-like [22] CC-NUMA where each node consists of a compute processor, its cache hierarchy and associated controllers, a communication processor, and memory. The memory is used to store not only private and shared variables but also the coherence directory for the memory blocks that it contains and the data structures of the communication processor that do not fit in the latter's cache. In this section we expand on the design of the communication processor and on the implementation of the communication primitives of the previous section.

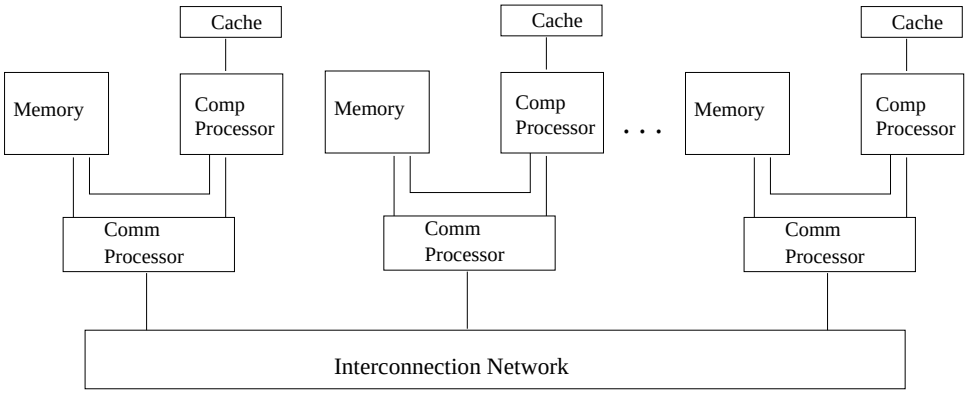

Figure 1: The architectural model

3.1 Communication Processor

Figure 2 depicts the model of the communication processor and its interfaces with the compute processor, the memory, and the network. The function of the communication processor is to handle network and memory related transactions. The communication processor integrates a processor/cache interface (PI), a network interface (NI), and a programmable protocol processor with instruction and data caches. In contrast with the MAGIC processor in the Flash system [22], which uses hardware to schedule and dispatch messages to the protocol processor, in our model both tasks are performed in software. The motivation for this choice is that a software implementation adds flexibility. The types and the formats of the messages may change as the system evolves; software scheduling and message dispatching allow the system to adjust to such changes more easily.

Communication between the communication processor and the compute processor is through the PI interface. Similarly, the communication processor communicates with the network through the NI interface. There are two queues in each of PI and NI. The input queue of PI receives cache miss requests, user-initiated communication requests, and writes to memory, either write-through or replacements. The output queue of PI is used to forward miss data to the compute processor's cache, and to issue cache invalidation/purge requests. The input and output queues of NI receive the same types of messages intended for or coming from other nodes in the CC-NUMA system. Additional buffers, to save data in transit, are included in the communication processor.

Since message processing is one of the most time critical functions of the communication processor, the processor architecture is optimized to handle it efficiently. When a message arrives, it interrupts the protocol processor. The interrupt handler dispatches the message to an appropriate message handler based on the message type (see below). The message handler either executes the message directly or moves the message from PI or NI to the software message queues, just as in Active Messages [31]. However, on a conventional processor, an active message handler still results in too high an overhead for a communication processor maintaining cache coherency at the granularity of a cache line. In order to achieve high-performance message passing, we choose to use a processor architecture that supports rapid interrupt handling and context-switch. The processor needs to have only two hardware contexts, one of which is dedicated to the message interrupt handler which dispatches the messages to the corresponding message handlers. In an optimized design, the

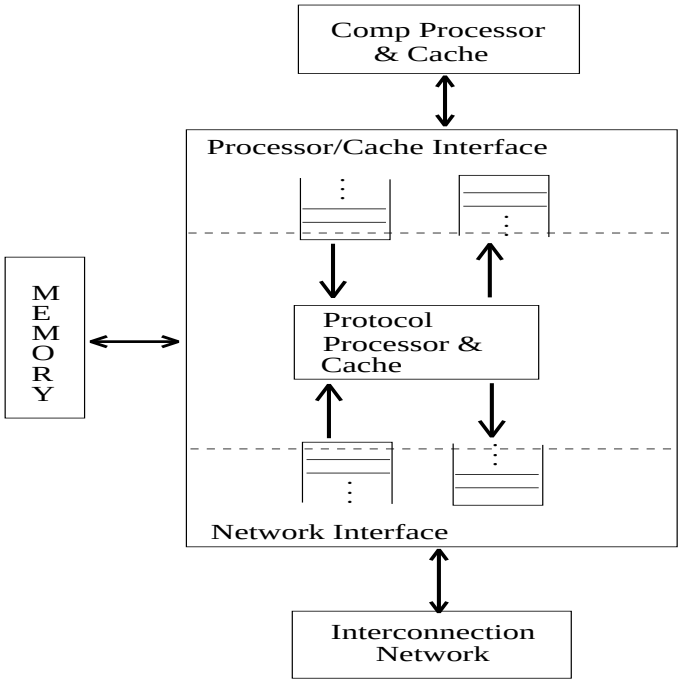

Figure 2: The communication processor

overhead of interrupt handling and context switch can be minimized to just a few instructions [1].

Finally, in our architectural model, we let the private references bypass the communication processor to avoid unnecessary processing overhead. To that effect, the compute processor is directly connected to the memory module. In our current model, both private and shared read misses block the compute processor. We operate under a sequentially consistent model and therefore shared write misses will also block the compute processor. We will use a write buffer for private write misses.

3.2 Message Handlers

As mentioned above, message processing in the communication processor must be performed efficiently. Moreover, the messages whose incompletion blocks the progress of a compute processor should be given highest priority. The messages corresponding to the communication primitives generated by the user will be given low priority and their corresponding message handlers will be interruptible by messages of high priority.

High priority messages are executed as soon as the communication processor is not processing another high priority message. The message handler can either process the message to completion if all actions can be taken in the node (e.g., invalidation or when receiving data that was requested) or process partially the message if other nodes of the system are needed (e.g., cache miss on a shared variable). In that latter case, the message handler will create a thread to be awakened upon receiving the right acknowledgment.

Low priority messages are removed from the PI and NI interfaces and stored in message queues in memory, most often the communication processor's cache. When the communication processor becomes idle, it polls the (software) message queues to see if there is any pending message. If there is, the message is handled but it can be interrupted for the processing of high priority messages, hence the need for a fast context-switch.

One of the major issues in implementing the message handlers for the communication primitives is to filter out wasteful data transfers. One plausible implementation would be to filter the user's call at the processor on which the call is being initiated. For example, if the data is already present in the local cache in the case of a *get* operation, then the operation would be ignored. This approach has the advantage of filtering out useless operations at the earliest possible stage. However, if the operation is useful it will incur double overhead since there is a need to consult again the directory status at the home node in order to determine the appropriate action to take. This is necessary because (1) the local cache status checking cannot guarantee completely the legality of the operation and (2) cache state may have changed from the time the local checking is performed to the time the forwarded request is processed by the home node.

We chose the alternative, namely to always forward the user generated calls to the home node. If no action is to be taken, then the only price we pay is a waste of communication processor time and of network bandwidth. On the other hand, we do not disturb the progress of the compute processor with local checking. If the action is useful, then it will be completed faster. In the case of transfers larger than a single cache line, the directory information is consulted line by line to determine whether the requested data transfer is necessary. Finally, since communication primitives are only performance hints, there is no need to retry a data transfer if it had to be aborted (e.g., if the cache line is being processed by another message handler). This is in contrast with cache miss requests that need to be performed to completion.

4 Experimental Methodology

4.1 Applications and Experiments

For our experiments, we selected three kernel applications, FFT, LU factorization, and RADIX sort from the SPLASH-2 benchmark suite [32, 33]. These applications have been coded with a CC-NUMA system in mind, thus they already have some communication optimizations embedded in them. They also exhibit coarse grain regular communication patterns that can be exploited by the proposed communication primitives. Table 2 summarizes some pertinent statistics about the applications: problem size, number of instructions, number of read and write references to shared data.

The execution time[1] of an application running on a parallel system can be divided into four components:

- Computation time (labeled *Processor busy* in the figures in Section 5)

[1] In our measurements, we exclude the initialization time from the execution time because the way initialization is performed is not always realistic.

Program	Problem size	Total instr(M)	Shared read(M)	Shared write(M)
FFT	64k points	33.32	6.00	5.73
LU	256x256 matrix, 16x16 block	64.20	23.11	11.19
RADIX	0.5M integers, 1024 radix	29.55	5.64	3.46

Table 2: Applications: problem sizes, number of instructions executed (in millions), number of shared read references (in millions), number of shared write references (in millions).

- Synchronization due to the intrinsic properties of the algorithm, e.g., producer-consumer relationships, serial sections of the program, load imbalance (labeled *Synchro-Algorithm*)
- Communication time due to the memory latency (*Memory latency*)
- Extra synchronization time due to the effects of the memory latency (*Synchro-Memory*)

These four components are not independent. For example, a mismatch between the partitioning of computation and partitioning of data can easily cause excessive communication overhead. Likewise, an inefficient memory system exacerbates load imbalance. Since our goal is to test the efficiency of the communication primitives with respect to a hardware implementation, we will isolate the contributions of each component and compare the performance of the applications, via simulation, in four experimental environments.

- Case 1: A machine with a perfect memory system (PRAM model).
- Case 2: A hardware-based cache-coherent (full directory [5]) system.
- Case 3: A system that uses a communication processor/node with the communication processor implementing the coherence protocol (software implementation).
- Case 4: A system as in case 3, with, in addition, the communication processor being able to handle the user-based communication primitives (optimized software implementation).

The difference between case 1, where only the first two components are taken into account, and case 2 displays how much of the parallel efficiency is lost due to memory latency and cache coherence minimized as much as possible by the best hardware implementation. The difference between cases 2 and 3 shows the additional overhead brought by a software implementation. Case 4 reveals the potential performance gains obtained by supporting various communication optimization schemes on top of a flexible software infrastructure.

The systems we simulated had 16 processors and, except for the PRAM case, we considered 5 combinations of cache size and associativity: infinite cache, large (256 KB) direct-mapped and 2-way set-associative caches, and small (32 KB) direct-mapped and 2-way set-associative caches. The cache line size was set at 32 bytes.

4.2 Simulation Parameters

We use Mint [30] as our simulation tool since our interest is primarily in the performance aspects of the memory system. Mint is a software package that emulates multiprocessing execution environments and generates memory reference events which drive a memory system simulator. In addition to memory references, Mint provides an interface to trigger any user specified event. We make use of this interface for specifying the communication primitives in the applications. When encountering such primitives in the execution, Mint will generate special types of events that invoke the simulation back-end.

We assume perfect pipelining in the compute processor. This assumption affects only the *Processor busy* time, which remains essentially the same for the four environments, and thus does not bias the results. The memory system simulator models the operations of the communication processor in detail. This includes the interrupt overhead of message reception with the context switch, if needed, and moving the message from the PI and NI input queues either to be executed directly or to be stored in (software) queues, the time to package and write messages to the PI and NI output queues, and the manipulation of the directory data structure. Table 3 shows a list of major architectural parameters specified to the simulator.

Parameters	Value
Interrupt and context switch	8 cycles
Processor interface (inbound)	2 cycles
Processor interface (outbound)	4 cycles
Network latency (one way)	24 cycles
Network interface (in & out)	12 cycles
Retrieving data from memory (first word)	15 cycles
Retrieving data from compute proc. cache (first word)	12 cycles
Size of buffers (PI, NI)	infinite
Max. number of pending requests	10

Table 3: Principal parameters of the communication processor and associated values.

To illustrate the overhead of the software implementation, we show in Table 4 the timing differences between the hardware and software implementations of a shared read miss. The data is assumed to be in a non-exclusive state (i.e., one hop is sufficient). As can be seen, in ideal conditions (no contention) the software implementation is 54% slower. While this is the slow-down effect that we will use in our simulations, it is in fact quite favorable to the hardware implementation because we have not tried to optimize the communication processor as much as we could; we want to keep it a flexible, programmable resource. Furthermore, with current technological trends, network and memory latencies and bandwidths will not progress as fast as processor speed; the software implementation will suffer less from this widening gap. For example doubling only the network latency would reduce the 54% factor above to 39%. Finally, the software implementation latency will suffer less from expansion of the system; there will be no need to change the full directory structure into a partial one since all the directory look-ups are handled in software.

Action	Resource	Software latency	Hardware latency
Miss detection	Comp. processor	6	6
PI processing (inbound)	PI	2	2
Receiving and forwarding request	Comm. processor or hard logic	17	4
Network latency (including NI)	network & NI	36	36
Home node handling	Comm. processor or hard logic and memory	55	24
Network latency (including NI)	network & NI	36	36
Processing data	Comm. processor or hard logic	26	8
PI processing (outbound)	PI	4	2
Totals		182	118

Table 4: Difference in timing between hardware and software implementations of a shared variable read miss. The directory for the block is in a home node different from the node requesting the data.

5 Performance Results

In this section, we first describe briefly each application with an emphasis on the communication aspects, i.e., the massaging of the major data structures. We then present the results of the simulation of the four architectures described in Section 4 and assess the effectiveness of the software implementation using the communication primitives.

5.1 FFT

The application
The SPLASH-2 FFT algorithm is optimized for distributed or hierarchical memory systems [2]. The input data of FFT consists of a $\sqrt{n} \times \sqrt{n}$ matrix of complex numbers. The major data structures are the input matrix A, its transpose B, and another matrix of same dimension for the "roots of unity". In the beginning, one processor performs initialization for the entire input matrix and the roots of unity. The post-initialization algorithm consists of six phases. (1) $B \leftarrow A^T$; (2) and (3) Compute and update B; (4) $A \leftarrow B^T$; (5) Compute and update A; (6) $B \leftarrow A^T$. There are three synchronizing barriers, after phases 3, 5 and 6. Each processor is assigned to compute a set of consecutive rows. In the three transpose phases, each processor is responsible for reading a subblock of the source matrix and writing it into the corresponding rows of the destination matrix. During a computation phase, each processor operates on the set of rows set up in the prior transpose phase.

In a hardware invalidation-based protocol, communication occurs in the transpose phases. For example, in phase 4, when a processor reads a subblock of B, it will generate read misses since the data for that source matrix was generated on a subblock by subblock basis by other processors

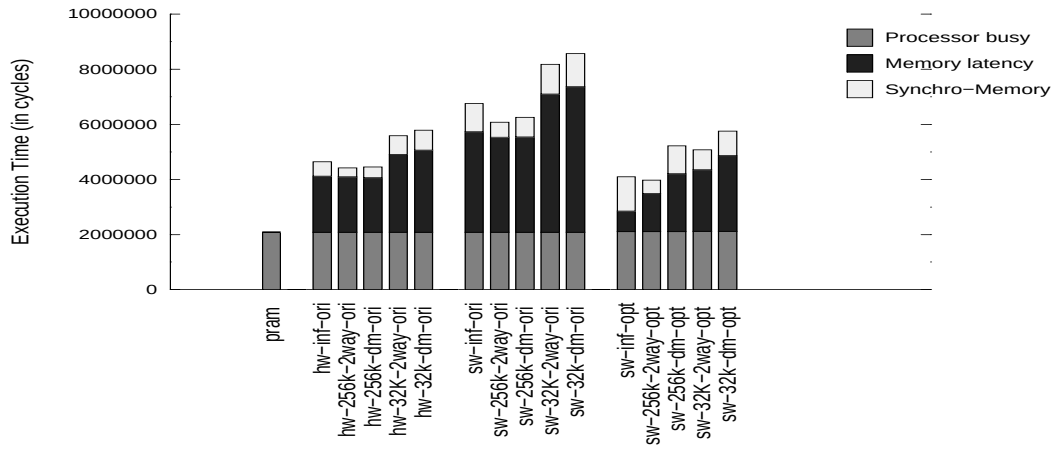

Figure 3: Execution times for FFT. Processor busy is the compute processor execution time. Memory latency is the amount of time the compute processor waits because of memory latency and cache coherence effects. Synchro-Memory is the synchronization time, including load imbalance, due to the effect of memory latency.

in phases 2 and 3. When a processor writes the data in A, it will need to generate invalidation messages since the first transpose phase left the corresponding cache lines in *shared* state.

We use the communication primitives to interleave computation with communication. More specifically, in phases 2 and 3 (likewise in phase 5), we insert *put* operations as soon as a processor (producer) has finished computing a row of B to disseminate the data to the other (consumer) processors. In addition, right after the *put*s we insert *getex* calls to place in the correct state the cache lines corresponding to the rows of A that the processor will overwrite in phases 4 (and 6). Thus, when the processor finishes the computing phase it has (or at least has requested) in its cache the needed data in the desired state to perform the transpose. Either the data has been *put* there by another processor or it has been prefetched via a *getex*. There is one condition, however, for this aggressive communication latency hiding scheme to be successful. Namely, the cache must be large enough to hold the working set of the computation phase and the working set of the transpose phase. If the cache is too small to hold the two working sets, cache pollution will occur because the *put* or the *get* operations will have been performed too early. To that effect, in the cases of small caches, the prefetching is scheduled just before the matrices are accessed in the transpose phases.

Simulation results

Figure 3 displays the simulation results of FFT running on the four model architectures and the combinations of cache size/associativity described in Section 4. We show, from left to right, the PRAM model (of course without cache), the hardware full-directory implementation, the software implementation, and the optimized software implementation, each with the five cache configurations (cf. Section 4.1). As mentioned above, the use of communication primitives was dependent on the size of the caches.

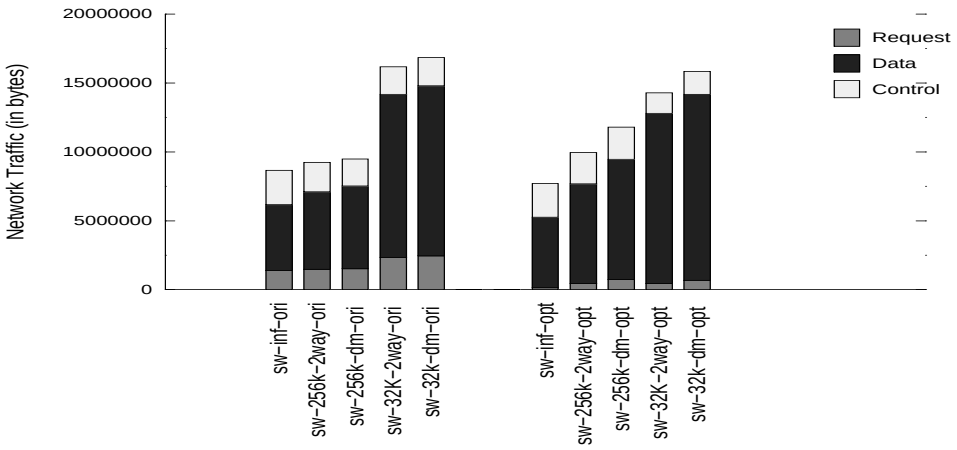

Figure 4: Network traffic for FFT. "Request" "Data", and "Control" are the network traffic for forwarding cache miss requests or communication primitives, transferring of data, and sending control messages (e.g., invalidation and acknowledgment) respectively. The network traffic of the hardware implementation is omitted since it is equivalent to that of the software implementation.

First, looking at the PRAM results, we see that the load balance is close to ideal. Second, a comparison between the PRAM and hardware implementation shows that communication, which occurs only during the transpose phases, is intensive. Execution time in the hardware implementation is more than double that of the PRAM when we include a realistic memory latency and a fast but inflexible coherence protocol. Third, as already noticed in other studies [11], infinite caches can be worse than large finite caches because while there is less data traffic there can be more coherence traffic or contention for data in a single cache. This latter phenomenon is the case here during phase 4 of the original program (phases 2 and 3 of the optimized program) since A is initialized by one processor.

Comparing the software and hardware implementations shows the price that is paid by adding (software) flexibility. In Section 4, we gave one example (cache read miss) where the latency in the software implementation was 54% higher than that of the hardware implementation. This ratio is close to the 70-80% and 70-96% ratios of the *Memory latency* and *Synchronization-Memory* components of the execution time. The slightly higher ratio in the execution times can be explained by the facts that messages that can be processed locally take relatively longer in the software implementation than in the hardware one and that there exists contention in the communication processor.

The insertion of communication primitives pays off handsomely. Now the optimized method is much faster than the original software implementation and, unless conflict misses occur frequently (direct-mapped caches), is even faster than the hardware implementation. In Table 5, we break down the execution times phase by phase of the two software schemes for the infinite cache case.

Phase	Application	Proc. Busy	Memory	Synchronization
1	Original	53925	792100	0
	Optimized	61119	386036	0
2,3	Original	1004490	3691	140073
	Optimized	1010790	271857	1128018
4	Original	53910	1570010	0
	Optimized	53908	8482	0
5	Original	915967	2382	736366
	Optimized	922303	150690	51664
6	Original	53958	1271600	122772
	Optimized	53955	2670	16036
Total	Original	2082305	3641568	1026376
	Optimized	2102136	821539	1222850

Table 5: Breakdown of execution times of FFT for original and optimized software implementation (infinite cache).

Notice first that the memory latency in the optimized case is only 23% of the original and this gain overwhelms, in absolute numbers, the 22% loss in synchronization time. The introduction of the communication primitives has a negligible effect on the computation time (*Processor Busy*). Overall, the optimized implementation is 40% faster than the original implementation.

In the original implementation, all the memory latency overhead is incurred in the transpose phases 1[2], 4, and 6. This overhead is almost twice as much as the processor's busy time. During the transpose phases, no computation takes place and conversely during the computation phases there is no memory traffic (recall that we have an infinite cache).

After the optimization, communication is performed while computation is in progress, except during phase 1. In phase 1, the reduction of memory latency stems from the possibility of transfer of several cache lines with a single request. This eliminates the need for sending and processing multiple small messages, thus reducing network traffic (cf. Figure 4 which shows an appreciable decrease in the number of requests). This bulk transmission mode also enables the home node to pipeline the data transfers, hiding the communication latency even further. In phases 2 and 3[3], and phase 5, the large amount of computation should be sufficient to overlap with the entire memory latency of phase 4 and 6 respectively. The residual memory latencies in those phases are caused by the fact that the number of pending communication operations is limited to 10, which is less than what is required, namely $(number\ of\ processors - 1)$ *put* and one *getex* for a complete overlap with the

[2]The first transpose phase (phase 1) incurs shorter memory latency because when a processor first writes the transposed data into its own cache no other processor has touched the data yet. Therefore writes proceed almost twice as fast as in the other two transpose phases.

[3]The large synchronization time at the end of phase 3 of the optimized software implementation is caused by initializing the input matrix A on a single processor. A comparable corresponding synchronization time is found at the end of phase 5 of the original implementation.

computations.

The infinite cache case illustrates the performance gain potential of an aggressive communication and computation overlapping scheme. When the cache size is limited, the use of aggressive communication primitives raises the danger of polluting the cache. In the FFT case, the pollution will be caused by conflicts between the current working set (e.g., a set of consecutive *rows* of B in phases 2 and 3), and future working set (e.g., a set of consecutive *columns* of B and corresponding rows of A in phase 4). In the case of the large 256 KB cache, we see that the pollution effect is indeed present since the memory latency time (in the optimized case) is twice as much as that of the infinite cache for 2-way set associativity and 3 times as much for direct-mapped. Even so, though, the performance is comparable to that of the hardware scheme where there are almost no conflict misses: slightly better for the 2-way case, slightly worse for the direct-mapped. For the two small caches, data is prefetched into the caches just before its use. The results show that the conservative communication primitives also reduce the memory overhead significantly without causing much of the cache pollution problem. Again the results are comparable with those of the hardware scheme.

Figure 4 shows the network traffic before and after the optimization. The network traffic of the hardware implementation is omitted since it is equivalent to that of the software implementation. As mentioned earlier, the optimized version will result in fewer requests. Cache pollution on the other hand will result in an increase not only in requests but also in data to be transferred since data displaced by prematurely prefetched data must be fetched again. This is particularly visible in the 256 KB direct-mapped cache. Finally, the amount of control messages slightly decreases because of the *getex* requests that place the data in the right state.

Summary
In the case of FFT, the use of communication primitives to prefetch data and to put cache lines in the correct state in advance yields execution times for the software optimization comparable, in fact even slightly lower, to those of the hardware implementation.

5.2 LU Factorization

The application
The SPLASH-2 parallel implementation of the LU factorization of a dense matrix has been optimized to exploit data locality [33]. Nonetheless, the serial sections of the application produce a fair amount of load imbalance. The input matrix is divided into submatrices, or blocks, which are assigned to processors in a 2D scatter decomposition fashion. In essence, every processor is responsible for factorizing the same number of blocks and this number diminishes almost uniformly for all processors during the computation. The algorithm iterates over the number of blocks along the diagonal. In the k-th iteration, the processor responsible for the block A_{kk} factorizes it. This is by necessity a serial part of the algorithm. Then the perimeter blocks A_{ik} and A_{kj} are computed using A_{kk}, a phase that requires communication. This can be performed in parallel but the processors owning the perimeter blocks all need to access data in the processor owning A_{kk}. Finally the remaining interior blocks A_{ij} are updated by the processors responsible for them. This phase also requires communication between the updating processors and the processors to which the perimeter blocks have been assigned.

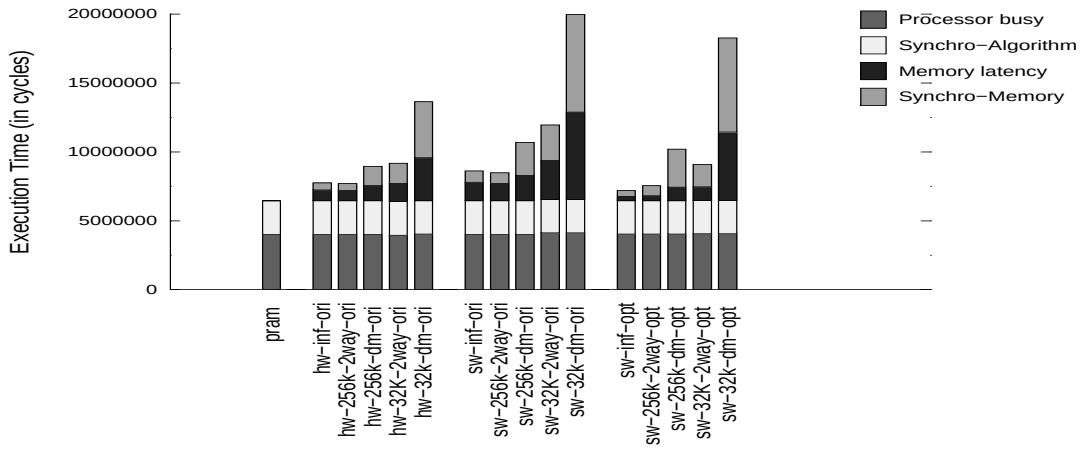

Figure 5: Execution times for LU. Processor busy is the compute processor execution time. Synchro-Algorithm is the synchronization time due to load imbalance intrinsic to the algorithm. Memory latency is the amount of time the compute processor waits because of memory latency and cache coherence effects. Synchro-Memory is the synchronization time, including load imbalance, due to the effect of memory latency.

The communication required in the first and second phases of each iteration provides some opportunities for using the communication primitives. Thus, we modified the algorithm in three ways. First when a (producer) processor is factorizing a diagonal block A_{kk}, it sends (*multicast*) a row of data as soon as the row is computed to the (consumer) processors in charge of the perimeter blocks A_{ik} and A_{kj}. Second, after a (producer) processor finishes updating one row of a perimeter block, it sends (*multicast*) the data to the (consumer) processors that use the block for processing the interior blocks. Finally, since an interior block is exclusively accessed by the processor in charge of its computation, we let the processor prefetch the data with ownership (*getex*) before updating the block. In the cases of the infinite and large caches, prefetching is scheduled once for all at the first instance of phase 3. For the small caches, prefetching is issued in each iteration.

Simulation results

Figure 5 displays the execution times for the five environments. The PRAM simulation shows the effects of the serial sections. Processors are idle one third of the time, on average. This synchronization is exacerbated on a realistic system by the additional memory overhead because busy processors take longer to complete the assigned tasks, thus keeping idle processors waiting longer. On the other hand, the memory overhead accounts for only 10% to 15% of the total execution time in the hardware implementation and 15% to 30% in the original software implementation. This leaves less room for optimization but reducing the memory overhead is still desirable because it helps reduce the synchronization overhead as well.

In LU, the 256KB caches are large enough to accommodate the whole working set of LU for the problem size we simulated. When the caches are 2-way set-associative, the optimized solution is

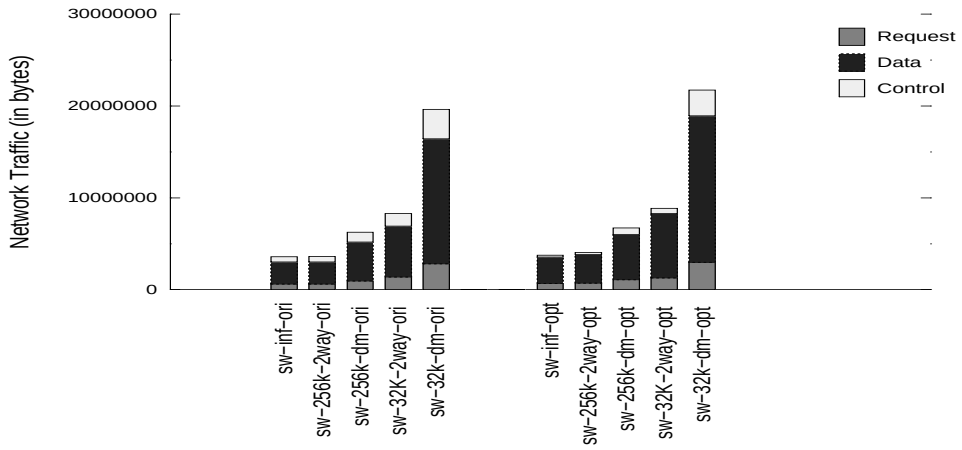

Figure 6: Network traffic for LU. "Request" "Data", and "Control" are the network traffic for forwarding cache miss requests or communication primitives, transferring of data, and sending control messages (e.g., invalidation and acknowledgment) respectively. The network traffic of the hardware implementation is omitted since it is equivalent to that of the software implementation.

as efficient as the hardware implementation for both large and small caches. When the cache is direct-mapped, the large number of conflicts takes its toll and the effect is magnified in the software implementations because of the increased latencies. Those conflicts that occurred within the working set of phase 3, when each processor factorizes interior blocks using perimeter blocks, cannot be avoided by the optimized software implementation. Thus we see less significant improvement, especially in the case of the 32K direct-mapped cache.

The network traffic does not vary much between the three implementations (cf. Figure 6). The optimizations do not introduce severe cache pollution.

Summary
In LU, there is less opportunity to perform optimizations. However, the use of *multicast* and prefetching is worthwhile, mostly if the caches are not direct-mapped. In that case, the performance of the optimized software is as good as that of the hardware implementation which, itself, is within 20% to 45% of the PRAM lower bound.

5.3 RADIX Sort

The application
RADIX sort is part of the NAS parallel benchmark [3]. It sorts k-bit integers by examining r bits ($r \leq k$) of the keys per iteration. In the parallel implementation, each processor is assigned an equal fraction of the keys. An iteration consists of three phases. In phase 1, each processor scans through its assigned keys and computes the local histogram and density of each radix value. In

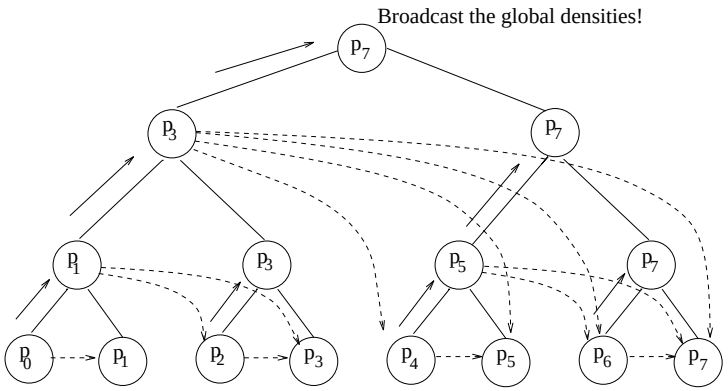

Figure 7: Communication patterns for computing the global densities and prefix sums of the local histograms. Each processor initially has local histograms and local densities in its cache. The solid arrows indicate the data transfer needed for both global densities and prefix sums. The dashed curves are for the prefix sums only. The label in each node is the processor that computes the intermediate or final results.

phase 2, prefix sums of the local histograms and the global densities (sums of local densities of all processors) are computed. These results are used to compute the new position of each key in the output order. In phase 3, the keys are permuted based on the new position computed in the prior step.

The major data structures are two arrays that are used alternately for the input and sorted keys, and a binary tree of arrays to store the prefix sums of the local histograms and the (partial) results of the global densities. Communication occurs in every phase at each iteration but with different patterns. In phase 1, each processor reads a portion of the keys, sorted and written in phase 3 of the previous iteration. Although the reading is sequential, the order of the keys cannot be determined until phase 3 completes. In phase 2, the communication patterns for computing the global densities and the prefix sums are illustrated in Figure 7 using eight processors. In phase 3, each processor determines the new order of the keys assigned to it and the pattern is random. False sharing will occur frequently in this phase since neighboring keys might be assigned to different processors.

The communication primitives can be used in each phase in different ways. In phase 1, the only possibility is to prefetch data as soon as phase 3 terminates. We insert *get* operations for prefetching keys before computing the local histograms and density functions. In phase 2, we insert *put* or *multicast* operations in the (producer) processors to send the data to the (consumer) processors according to the regular pattern of communication flow shown in Figure 7. Neither *get* nor *put* primitives can be used in phase 3 because the data locations are not known until run-time. A certain amount of false-sharing can however be avoided by having processors writing memory directly (write-through) instead of obtaining the ownership of a line and caching it locally. Thus a *writemem(wt)* call is inserted at the beginning of phase 3 and a corresponding *writemem(wb)* at the end.

Simulation results

Figure 8 shows the performance results of RADIX sort. As can be seen when comparing the PRAM and hardware implementation execution times, the memory latency effects are the most important

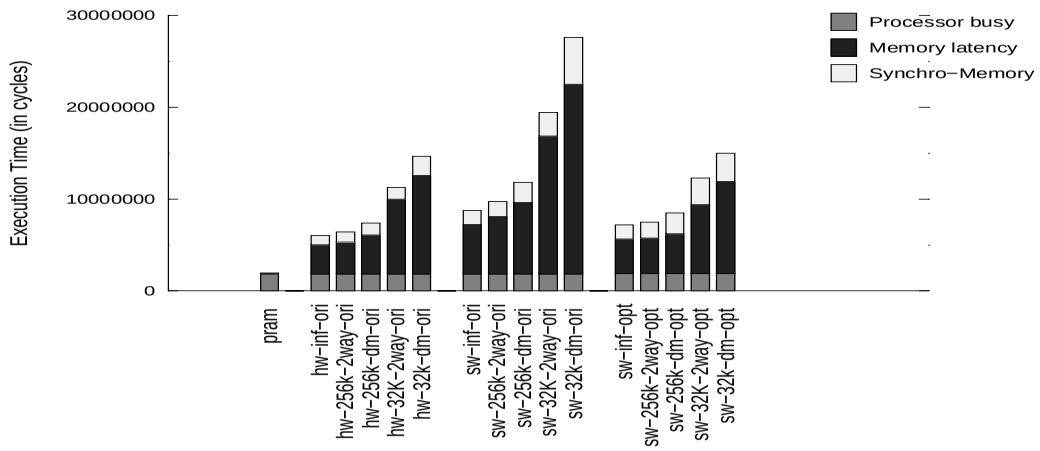

Figure 8: Execution times for RADIX. Processor busy is the compute processor execution time. Memory latency is the amount of time the compute processor waits because of memory latency and cache coherence effects. Synchro-Memory is the synchronization time, including load imbalance, due to the effect of memory latency.

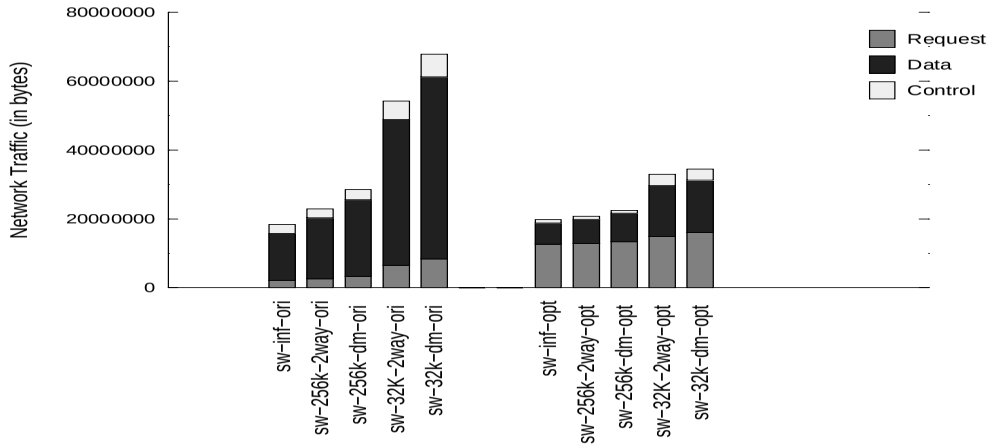

Figure 9: Network traffic for RADIX. "Request" "Data", and "Control" are the network traffic for forwarding cache miss requests or communication primitives, transferring of data, and sending control messages (e.g., invalidation and acknowledgment) respectively. The network traffic of the hardware implementation is omitted since it is equivalent to that of the software implementation.

Phase	Application	Instr	Memory	Sync
1	Original	704778	1783595	113057
	Optimized	704834	515701	310923
2	Original	139419	707990	1248630
	Optimized	139682	435627	1156961
3	Original	999610	3733430	248395
	Optimized	1048766	2905070	240463
Total	Original	1843872	6226998	1638077
	Optimized	1893347	3858511	1737689

Table 6: Breakdown of execution times of RADIX for the original and optimized software implementations, 256K 2-way set associative cache.

of the three applications. A very small load imbalance in the PRAM case occurs during phase 2. But since phase 2 is short compared to phases 1 and 3 (cf. the instruction count in Table 6), the effect is not significant and cannot be seen at the scale of the figure.

Table 6 shows the breakdown of the execution times for the case of the 256 KB, 2-way set-associative cache for the original and optimized software implementations. In the original implementation, and similarly in the hardware implementation, more than half of the memory latency effects arise in phase 3. Unfortunately this is where the communication primitives are least applicable since the communication patterns are random. Nonetheless, the change of write policy in that phase yields a 20% improvement over the original implementation. The prefetching effects in phase 1 are quite significant, decreasing the memory latency and associated load imbalance by a factor of two for that phase. The communication primitives also reduce the memory latency in phase 2 but here the effect is less important since that phase does not take much time.

The gains of the optimized implementation are even more significant for the small caches. The optimized software implementation is almost as good as the hardware one.

Figure 9 displays the network traffic before and after the optimizations. While in FFT and LU the amount of network traffic was about the same (but both the timing of when it took place and the amount of time to process the messages were different), here we have a large reduction due to the change in write policy. It is particularly striking in the case of small caches where not only false sharing but also conflict misses are avoided by writing through to memory in phase 3.

Summary
Radix is the most memory intensive of the three applications but also the least amenable to optimization. Nonetheless a combination of prefetching and of write policy change allows the optimized software implementation to be competitive with the hardware implementation.

6 Related Work

Cache coherence controlled by a communication processor and the integration of the shared-memory and message-passing paradigms have been extensively studied in the past three years.

The design of our communication processor is closely related to that of the Flash project [22, 16]. Flash is a tightly coupled CC-NUMA system that uses a programmable processor (MAGIC) and software (running on MAGIC) to maintain cache coherence. The MAGIC chip is highly optimized. It includes special hardware to assist message receiving, scheduling, dispatching, and directing outgoing messages to proper destination interface units. The processing of incoming messages, the protocol handling, and the preparation of outgoing messages are pipelined. In addition to its macropipelining architecture, MAGIC employs speculative memory access to overlap the memory latency with the protocol handling. As a result, the slowdown due to the software overhead is minimized to only 2%-12% over that of an ideal hardware implementation. In contrast, our co-processor is more flexible, allowing more easily the introduction of new primitives or changes in the protocols, and closer to what is available off-the-shelf, but the software overhead is of the order of 50%. The use of dedicated communication hardware can also be found in tightly-coupled message-passing systems such as the Intel Paragon [18] and has been proposed for networks of workstations [27].

Combining message passing with shared memory to overcome some of the inefficiencies of cache coherence mechanisms was first proposed in the context of the Alewife project [20, 21] and further elaborated in Flash [15]. A number of important issues have been raised and discussed, e.g., user-level messaging and protection, and coherence strategy for bulk data transferring. Since our interest was mainly on performance related issues, we have concentrated on the latter, imposing a global coherence strategy for prefetching and bulk data transfers.

The communication primitives that instruct the system to perform efficient data transfers resemble the asynchronous send/receive operations in message passing interfaces [4, 24], prefetching [23, 6, 14] and poststore [19] commands, non-blocking (bulk) read (get) and write (put) operations in the split-phase assignment statement of Split-C [9], and explicit communication mechanisms [26]. The common idea is the overlap of communication with computation. The differences are: first, in policy with respect to message-passing systems, since the global addressing space paradigm and cache coherence is maintained; second, in the (extendible) set of primitives, since the communication primitives include both producer-consumer oriented operations and availability of bulk transfers; and third, in implementation, since the execution of the optional communication primitives are interruptible by mandatory shared memory requests which are assigned a higher priority.

Tailoring the coherence protocol to the application can be done in several ways. Protocol modifications can be specified by the user at a coarse grain level [12, 13] or in incremental fashion [17], can be dictated by the compiler [25, 10], or can be the result of hardware monitoring [8, 29]. In the applications we have studied, we have seen the need for applying difference coherence strategies at the granularity of a computational phase on a data structure per data structure basis.

7 Conclusion

We have proposed a set of communication primitives that can enhance the performance of cache coherent shared-memory multiprocessors. These primitives give the user some of the capabilities of message-passing systems while keeping the correctness and simplicity aspects of the global address space paradigm. The primitives allow prefetch and post-store of blocks of data whose sizes are not limited to single cache lines and permit to tailor the cache coherence protocol to the needs of the application.

We have shown how these primitives could be implemented in a communication processor that handles not only these primitives but also all cache coherence transactions in software. This pure software approach will incur some overhead but will add flexibility since it facilitates the introduction of new primitives as well as the implementation of variations in cache coherence protocols.

We selected three benchmarks from the SPLASH-2 suite for performance evaluation purposes. We then simulated a 16 processor system with five different cache configurations. We simulated four environments: an ideal PRAM model to gauge the effects of memory latency and cache coherence, a system where cache coherence was maintained with a full directory hardware scheme, and two implementations with a communication processor, one using the original benchmarks and one with communication primitives appropriately inserted. With the parameters that we chose for the communication processor, the contributions to the execution times of the memory latency and cache coherence effects in the original software solution were at least 50% higher than those of the hardware implementation. With communication primitives, the optimized software solution gave results comparable to those of the hardware solution.

The evaluation results are encouraging and point to the value of a software approach enhanced by user directives. The quantitative results, which show a comparable performance to the hardware solution, are conservative since an increase in network latency or memory bandwidth would hurt relatively more the hardware implementation. The software solution is also more attractive since it is more scalable, for example there is no difficulty in keeping a full directory since it is a software data structure, and more amenable to change.

There are several directions that can be followed to expand this study. One is to see the effects of integrating the communication processor with either the compute processor or the memory controller. In the first case, the interface between the compute and communication processors will be much faster. This speed advantage could be negated by the competition for off-chip bandwidth and overall complexity of design. The second case would be more in the philosophy of intelligent memory, i.e., the integration of processor-like functions in DRAMs [28]. Another possible study is to see how the communication processor could be shared in cluster-like environments. In a cluster-like architecture, tight-coupling between the communication processor and several compute processors could become too complex. A looser coupling, i.e., another design, might be more appropriate. Whether the looser coupling and potential saturation of the communication processor might be detrimental to overall performance is an interesting issue.

References

[1] A. Agarwal, D. Kranz B.-H. Lim, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 104–114, 1990.

[2] D. H. Bailey. FFT in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[3] D. H. Bailey et al. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[4] V. Bala et al. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4):445–462, April 1994.

[5] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEE Transactions on Computers*, c-27(12):1112–1118, Dec. 1978.

[6] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 223–32, 1994.

[7] T. M. Chilimbi and J. R. Larus. Cachier: A tool for automatically inserting cico annotations. In *Proceedings of International Conference on Parallel Processing*, pages 89–98, 1994.

[8] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 98–108, 1993.

[9] D. E. Culler et al. Parallel programming in Split-C. In *Proceedings Supercomputing '93*, pages 262–73, 1993.

[10] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *Proceedings of International Conference on Parallel Processing*, pages 229–238, 1988.

[11] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 373–382, 1988.

[12] B. Falsafi et al. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–9, 1994.

[13] M. I. Frank and M. K. Vernon. A hybrid shared messory/message passing parallel machine. In *Proceedings of International Conference on Parallel Processing*, pages 232–236, 1993.

[14] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessor with memory hierarchies. In *Proceedings of International Conference on Supercomputing 1990*, pages 354–368, 1990.

[15] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38–50, 1994.

[16] M. Heinrich et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274–285, 1994.

[17] M. Hill, J. Larus, S. Reinhardt, and D. Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. In *Proc. of 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, 1992.

[18] Intel Corporation. *Intel Paragon(tm) Supercomputer Product Brochure.* http://www.ssd.intel.com/paragon.html#system.

[19] Kendall Square Research Corporation. *KSR1 technical summary*, 1992.

[20] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.H. Lim. Integrating message-passing and shared-memory: early experience. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, 1993.

[21] J. Kubiatowicz and A. Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of 7th ACM International Conference on Supercomputing*, 1993.

[22] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, 1994.

[23] D. Lenoski et al. The Standford DASH multiprocessor. *IEEE Transactions on Computer*, 25(3):63–79, March 1992.

[24] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994.

[25] D. K. Poulsend and P.-C. Yew. Integrating fine-grained message passing in cache coherent shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 33(2):172–188, March 1996.

[26] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of Supercomputing '95*, 1995.

[27] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proc. of 24th Int. Symp. on Computer Architecture*, pages 34–43, 1996.

[28] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proc. 23rd Int. Symp. on Computer Architecture*, pages 90–101, 1996.

[29] P. Strenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 109–118, 1993.

[30] J. E. Veenstra and R. J. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–7, 1994.

[31] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. K. Schauser. Active messages: a mechanism for intergrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–66, 1992.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[33] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 219–229, 1994.