Exploiting the Capabilities of Communications Co-processors

Klaus E. Schauser, Chris J. Scheiman, J. Mitchell Ferguson, and Paul Z. Kolano

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{schauser,chriss, ferguson,kolano}@cs.ucsb.edu

Abstract

Communications co-processors (CCPs) have become commonplace in modern MPPs and networks of workstations. These co-processors provide dedicated hardware support for fast communication. In this paper we study how to exploit the capabilities of CCPs for executing user level message handlers. We show, in the context of Active Messages and Split-C, that we can move message handling code to the co-processor thus freeing the main processor for computational work. We address the important issues that arise, such as synchronization, and the limited computational power and flexibility of CCPs. We have implemented co-processor versions of both Active Messages and Split-C. These implementations, developed on the Meiko CS-2, provide us with an excellent experimental platform to evaluate the benefits of a communications co-processor architecture.

1 Introduction

Many modern parallel architectures, including networks of workstations, contain dedicated communications co-processors (CCPs) to support fast communication. Examples of architectures with CCPs include the Intel Paragon [PR94], Manna [BGSP94], Meiko CS-2 [HM93], Flash [Kea94], Typhoon [RLW94], and cluster of workstations connected via ATM networks [vEBBV95]. These co-processors provide the protection, reliability, and protocol handling needed for communication, thus freeing the main processor for computational work.

In many forms of communication, when a message arrives, there is some action that must be performed to incorporate the data into the on-going computation. For example, this is the case with Active Messages, RPC, and tagged send & receive [TM94]. A message may match a tag, insert an item on a queue, increment a variable, or store something in memory. Executing these handler functions on the main processor incurs a high overhead, as the main processor must either use expensive interrupts or poll frequently for incoming messages. Since some communication co-processors can execute arbitrary code, it may be desirable for these co-processors to perform the message handling.

In this paper we study how to exploit the capabilities of CCPs for executing user level message handlers. Executing message handlers on the co-processor provides a number of benefits [SS95, KNW95]. Since the co-processor is fully dedicated to communication, it is likely to serve incoming messages faster than the main processor. Furthermore, if all messages are handled by the co-processor, there is no need for the main processor to be interrupted or for polls to be inserted into the computation. While the computational overhead of the poll usually is not significant, the real cost of polling occurs when a request message is not processed promptly, causing the requester to wait. If messages are received and processed by a co-processor, this delay can prevented.

Executing message handlers on a communications co-processor, however, raises a number of issues that must be addressed. Frequently, CCPs are special purpose hardware which are limited in computational power and flexibility. This constrains the kinds of computations that can be performed by the co-processor. For example, some co-processors do not support floating point operations, or can not run arbitrary user code. Another important issue is synchronization between the co-processor and the main processor. Sometimes, code on the main processor must execute atomically with respect to certain message handlers. When not taking advantage of a co-processor, explicit polling on the main processor provides an easy solution. When a co-processor can execute message handlers independently of the main processor, however, explicit synchronization is required.

In this paper, we evaluate the handling of messages by the co-processor in the context of Active Messages and Split-C. Active Messages associate with each message a handler function, which is executed on the receiving processor upon arrival of the message [vECGS92]. In order to evaluate our approach under large parallel applications, we also implemented Split-C, a simple parallel extension of the C programming language. Using Split-C gives us access to a large set of parallel programs. The programs serve as a useful benchmark set that allows us to evaluate the absolute performance of the implementation based on the CCP.

The structure of the remainder of the paper is as follows. Section 2 gives an overview of Active Messages and Split-C. Section 3 presents the implementation strategy for running the Split-C library on a co-processor architecture. Section 4 presents in more detail our hardware platform, the Meiko CS-2 architecture. Section 5 evaluates the performance of our Split-C library implementation. Section 6 discusses related work. Finally, Section 7 summarizes our experiences and concludes.

2 Active Messages and Split-C

Active Messages provide a universal communications architecture which is frequently used by compiler and library writers [vECGS92]. Split-C is a C-based parallel language which makes use of Active Messages. Our goal is to execute much of the message handling on the co-processor in order to free up the main processor.

2.1 Active Messages

Each Active Message has a handler function associated with it which is executed on the destination processor when the message arrives. Active Message handlers are intended to be short functions which execute quickly and are not allowed to suspend. Handlers are divided into two classes: requests and replies. A processor can send a request message to an arbitrary processor. When it arrives, the specified request handler is invoked. Request handlers may answer by sending a single reply message. However, to simplify deadlock considerations reply handlers are prohibited from additional communication. Under the Active Message model, messages travel from user space (the send instruction) directly to user space (the message handler), avoiding any form of buffer management and synchronization usually encountered in traditional send & receive.

Although they are quite primitive, Active Messages have become an important communication layer because of their efficiency. The small overhead and low latency facilitates building more complicated communication layers [TM94] and makes it a desirable target for high-level language compilers [CGSvE93, CDG$^+$93]. Over the past several years Active Messages have been implemented on many different hardware platforms. Several of these architectures have a communications co-processor, which is used to support reliable and protected communication. So far only the Meiko CS-2 and the Intel Paragon have been used to run user level message handlers on the co-processor [KNW95, SS95].

2.2 Split-C

Split-C is a simple parallel extension of the C programming language [CDG$^+$93]. Split-C follows the SPMD model of computation: a single thread of computation is started on each processor. Both the parallelism and data layout is explicit and is specified by the programmer. Split-C provides a global address space in the form of distributed arrays and global pointers, and supports several efficient split-phase operations (including bulk transfers) to access remote data. Split-phase operations separate the initiation of a memory operation (the request) from the response. Split-phase operations in Split-C are *put* and *store*, for transferring data to another location, and *get*, for retrieving data from another location. (Put and get are acknowledged transactions, while store is a one-way operation.)

The split-phase nature of Split-C requires that the basic memory operations (put, get, and store) keep track of outstanding memory transactions. Split-C keeps track of these outstanding data transfers with counters, using Active Messages to transfer the data and increment or decrement the counters. Every put or get request increments a counter to indicate that an operation has begun but not yet completed. When the data has been successfully transferred, the counter is decremented. The user can wait for all outstanding gets and puts with the Split-C function sync(), which waits for the counter to equal zero. Unlike the get and put operations, a store operation is one-way and uses two counters: one is incremented when data is sent, and the other is incremented when data is received.

Since get, put, and store all work in a similar fashion, we examine only the get in more detail. As shown in Figure 1, a get operation consists of 3 steps: First, the requesting processor (1a) sends the “get” request to the remote processor and (1b) increments the counter tracking outstanding requests. Next, the remote processor receives the request message, (2a) reads the requested data, and (2b) sends it back. Finally, the requesting processor receives the reply, (3a) stores the data, and (3b) decrements the counter.

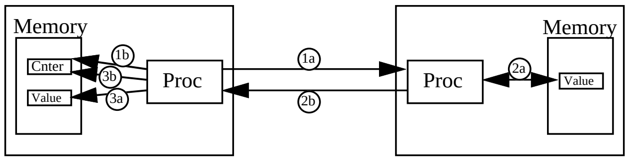

Figure 1: *Steps required for a Split-C get operation.*

Steps 2 and 3 correspond to Active Message handlers (each processor servicing the request or reply message). As described in the next section, we implement Split-C by running the Active Message handlers directly on the communications co-processor.

3 Implementation of Split-C on Co-processor Architectures

This section shows how we implement Split-C on a co-processor architecture. Using the co-processor to handle messages has the advantage that the main processor on the sending side only needs to initiate the communication process, while the main processor on the receiving side can proceed uninterrupted. The co-processors keep track of the outstanding requests, send the requested data, and move the incoming data into the specified memory locations.

For a co-processor implementation of Split-C, the steps originally done by message handlers on the main processor are now executed by the co-processor. For the get operation, as shown in Figure 2, the co-processor reads and stores the data, as well as decrements a counter. The steps are almost exactly the same as in Figure 1, except that the co-processors on both nodes are utilized. In Step 1, the message is sent to the receiving co-processor, and the counter is incremented as before; in Step 2, the receiving co-processor reads the data and sends the reply; and in Step 3, the requesting co-processor stores this data in memory and decrements the counter. In most architectures, the co-processor is also involved on the sending side, to ensure protection. But this is transparent to the application, which is just provided with a user-level communication interface.

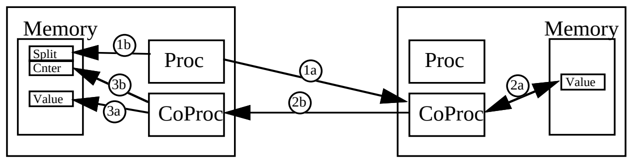

Figure 2: *Steps required for a Split-C get operation on a co-processor architecture.*

There are several synchronization issues that must be addressed. For example, a problem arises if the co-processor decrements the counter just as the main processor issues another get request: the counter value could be corrupted. We have to ensure mutually exclusive access if we allow both the main and co-processor to update the same counter. Implementing this with locks can be quite expensive. As proposed by [KNW95], a more efficient solution is to split the counter in two counters, using one counter for the main processor increments and the other for the co-processor decrements. This solves the synchronization problem. No race condition can occur since each processor can only write to one location, and can read the other. The sum of the two counters produces the desired counter value. The counter for the put operation is treated similarly. (The counter for store is already split into sending and receiving counters in the basic Split-C implementation.)

Another important synchronization issue, when executing code on the co-processor, involves atomic operations. Split-C allows the user to specify message handler functions which are atomic with respect to other message handlers as well as the main computation. (These atomic functions are subject to the same constraints as Active Messages: they cannot block and can only send a single reply.) These functions are intended only for very simple operations; for example, enqueuing or dequeuing an object. If the co-processor is allowed to execute these atomic functions, some mechanism has to be applied to ensure mutually exclusive execution of the function. The simplest solution is to allow only the main processor to execute such functions. The main processor will have to poll for atomic functions it must execute.

A final issue is that frequently a communications co-processor only provides limited functionality and cannot run arbitrary message handlers. For instance, many co-processors cannot execute floating point operations. Additionally, most communication co-processors have less design effort invested compared to the commercial main processors, thus are not as fast. It can be more efficient to run resource intensive tasks on the main processor instead. Not only will the code run faster, but it will not burden the communications co-processor and prevent new messages from being processed.

In summary, communication in Split-C consists of a number of split-phase memory primitives, plus a collection of less frequently used operations such as atomic functions. To keep track of outstanding memory transfers, Split-C uses counters. The memory transfer and counter operations can easily be performed by the co-processor, and, if split counters are used, no synchronization is needed with the main processor. Atomic operations, on the other hand, have stringent synchronization requirements and may therefore execute more efficiently on the main processor. In the case of the Meiko CS-2, the co-processor provides features which we can exploit to further optimize this basic implementation. This is described in the next two sections.

4 The Meiko CS-2 Architecture

The Meiko CS-2 consists of Sparc based nodes connected via a fat tree communication network [HM93]. It runs a slightly enhanced version of the Solaris 2.3 operating system on every node, and thus closely resembles a cluster of workstations connected by a fast network. Each node in a CS-2 contains a 40 MHz SuperSparc processor with 1 MB external cache and 32 MB of main memory. The SuperSparc is a three-way superscalar processor, and achieves a peak rate of 120 MIPS.

Each node in a CS-2 contains a special CCP, the Elan processor, which connects the node to the fat tree. The CCP enables direct user-level communication. For every message, the co-processor ensures protection by a series of translations and checks at both the sending and receiving side. This achieves protected user-level communication without the intervention of the operating system for every message.

The Elan co-processor consists of a number of functional units, which include the DMA engine (for sending data), the thread engine (for running arbitrary code, including user code), and the input engines (for receiving messages). The *DMA engine* performs DMA transfers of arbitrary size from a virtual address on the source node to a virtual address on the destination node. The *thread engine* implements a Sparc-like instruction set enhanced with special instructions for opening and closing connections, sending network transactions, and transferring control to other functional units. Since the Elan co-processor contains its own TLB, it can access virtual memory and run user-level code.

Meiko has a unique synchronization mechanism called an *event*. When an event is triggered, it can, among other things, cause a co-processor thread to resume execution. They allow for synchronization since the main processor and most of the Elan engines can setup and trigger events. A thread waiting on an event uses no resources except memory, so there is no penalty for having many suspended threads.

5 Performance of the Co-processor Implementation

We now give the details and discuss the performance measurements of our co-processor implementations for Active Messages and Split-C. First, we examine the raw communication performance of the Meiko CS-2 and compare our co-processor implementation of Active Messages to the version that runs handlers on the main processor. Next, we discuss our optimized implementation of Split-C, which specializes the message handlers for execution on the co-processor. Finally, we examine applications written in Split-C.

5.1 Performance of Active Messages

In this section we compare the performance of two implementations of Active Messages with the raw performance measurements of a basic DMA transfer. The first implementation uses the CCP only on the sending side. The second implementation goes a step further and also uses the co-processor on the receiving side to execute the message handlers.

The one-way latency of a DMA transfer establishes the best performance an ideal implementation could achieve. We compute this latency (9.5 μs as shown in Table 1) by halving the round-trip latency involved in transferring a flag between two processors. This roundtrip operation is performed repeatedly so that overhead incurred by starting and stopping the timer can be ignored.

Sending an Active Message involves the following steps: (1) The sending processor assembles the message in the outgoing buffer and sets a flag to indicate that the message is ready. (2) The sending CCP polls on this flag. Once it sees that the flag is set, it checks whether the receiving buffer on the remote processor is empty, and if so deposits the message into the receiving buffer. (3) The main processor on the receiving node checks this buffer on a poll; when it encounters a message, it dispatches to the message handler and frees the buffer. This scheme, which is described in more detail in [SSFK95], essentially implements a remote queue [BCL$^+$95] of depth one.

In the second implementation of Active Messages, the CCP is used both on the sending and the receiving side. In this implementation, the main processor is no longer involved in the reception of messages. When a message arrives, it triggers a thread (using Meiko's event mechanism). This thread calls the appropriate Active Message handler.[2]

As shown in Table 1, the one way and roundtrip latencies are more than twice as large as the main processor implementation. This is because the co-processor is an order of magnitude slower (runs at only 3 MIPS versus up to 120 MIPS on the main processor) and has only an 8 word instruction cache, no data cache, and no register windows.

Implementation	one-way	roundtrip
Basic DMA	9.5 μs	19 μs
AM main processor	12.5 μs	23 μs
AM co-processor	30 μs	52 μs

Table 1: *Comparison of DMA and Active Message implementations for a simple ping-pong program.*

Our implementation involves a dispatch thread on the receiving co-processor, which takes incoming messages and calls the proper handler function. If a program or library uses only a few message handlers, one optimization is to avoid this dispatch thread and directly invoke the proper handler function. This approach is ideal for the Split-C library, which only contains a few important message handlers. This forms the basis for the optimized Split-C implementation described next.

5.2 Performance of Split-C Primitives

As we saw in the previous section, a direct implementation of the Split-C primitives using Active Messages on the co-processor is about twice as slow as the main processor implementation. To improve the Split-C primitives, we exploit the fact that Split-C requires only a small number of Active Message handlers. While the generic Active Message implementation requires a dispatch thread to call the appropriate message handler, the optimized version can avoid this dispatch and directly invoke the appropriate handler. Furthermore, many Split-C operations only require the functionality that is already built-in to the Elan hardware. We can use special communication primitives such as DMAs and remote atomic adds to avoid running thread code on the receiver.

Before addressing the optimizations, we present the timings for basic Split-C primitives. Figure 3 gives an overview of the implementation results for the get, put, and store

[2] Our implementation is optimized by using $P-1$ threads per processor for the sending and receiving of messages. Each thread is responsible for receiving messages from a single remote processor, and each is triggered by its own event. Note that multiple events/threads only take up space; threads do not run unless triggered by an event. This scheme assures that two sending processors do not set the same event at the same time. Furthermore, in our implementation, a processor waits for an acknowledgment before sending a second message to the same destination; therefore, no buffer management is needed.

operations. It shows the latencies for the optimized co-processor implementation and compares them to the versions of Split-C that run handler functions on the main processor and co-processor. We see that get and store are still much slower in the co-processor library, compared to the main processor version: 34 vs. 24μs for get, and 22 vs. 13μs for store. However, the times are much better than those under the Active Message co-processor implementation (54μs for get and 30μs for store). Furthermore, put and bulk store are actually faster under the optimized co-processor implementation: 22 vs. 24μs for put, and 23 vs. 27μs for bulk store. This is because we were able to specialize these functions in the co-processor implementation, while the main processor implementation relies on generic Active Messages. The co-processor implementation also out-performs the main processor implementation for long messages:[1] it achieves 38 MB/s compared to 32 MB/s.

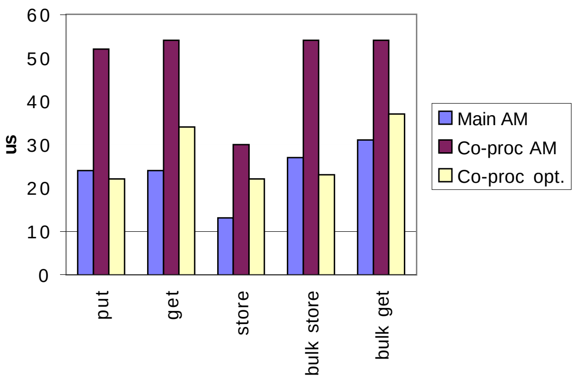

Figure 3: *Latencies for basic Split-C operations, for three versions of Split-C. Main AM is the version of Split-C which executes the Active Message handlers on the main processor; co-proc AM is the version running Active Messages on the co-processor; co-proc optimized is the version optimized for the co-processor. (The bulk store is for 32 bytes.)*

We now discuss the optimization of the three Split-C operations— put, get, and store— in more detail. The *put* operation increments a counter, transfers data to a remote processor, then decrements the counter when the transfer is complete. The optimized put operation is faster because it executes no code on the receiver; we can use a highly optimized DMA to transfer the data and avoid any sort of Active Message altogether. On the Meiko, the completion of a DMA can trigger a thread on the sending co-processor, and this thread can decrement the counter.

[1]This occurs because the main processor version trades off bandwidth for latency: The co-processor continuously schedules a thread which checks whether the main processor wants to send a message. While this reduces the latency, it takes resources from the co-processor and thus reduces the bandwidth for DMAs used in bulk transfers. The co-processor implementation, on the other hand, uses the event mechanism so that none of its many threads require resources until needed. Consequently, these threads do not affect the bandwidth.

The *get* operation increments a counter, requests data from a remote processor, then decrements a counter when the data has arrived. The get operation is also optimized using DMAs. The requesting processor first sends a DMA descriptor to the remote processor (using a second DMA), then triggers the DMA descriptor to run on the remote processor. This DMA transfers the required data and triggers thread code on the requesting co-processor which decrements the counter.

The *store* operation increments a local counter, transfers data to a remote processor, and increments a second counter on the remote processor. Unlike get and put, the store involves a computation on the remote processor. The Meiko supports remote atomic adds. Thus, the sending processor sends the data, then performs an atomic add of the counter on the receiving co-processor. This avoids the high cost of triggering and running thread code.

5.3 Performance of Split-C Applications

In this section we compare the performance of full Split-C applications under the two implementations. As the measurements in the previous section show, gets and stores using the co-processor are about 30–40% slower than those using the main processor. Nevertheless, executing message handlers on the slower co-processor can be beneficial, because the communications co-processor provides more responsive service for requests. Furthermore, because the main processor spends less time servicing messages, it has more time for the computation and can overlap communication and computation to a larger degree. To evaluate this scheme, we compare the performance of the two Split-C implementations for a number of benchmark programs.

Table 2 lists our benchmark programs, along with the corresponding running times for both versions of Split-C. All programs were run on a 16 node Meiko CS-2 partition. Bitonic is a bitonic sort program, which uses bulk store operations for communication. Cannon codes two versions of Cannon's matrix multiply algorithm (which vary on data placement); both use bulk store operations. FFT performs a fast Fourier transform computation involving 2 computation phases separated by a cyclic-to-blocked remap phase (done with store operations). FFTB is the same FFT program, except communication uses bulk stores. MM codes two versions of a blocked matrix multiply algorithm (which differ with respect to data placement); both use bulk gets for data movement. P-ray is a parallel ray-tracer program, which uses bulk put operations. Radix is a radix sort algorithm which uses store operations. Radixb is the same radix sort algorithm, except it uses bulk store operations. Sample is a sample sort program which uses get, put, and bulk get operations. Sampleb is the sample sort algorithm, but optimized with bulk store operations. Shell is the shell sort algorithm using bulk reads. Wator is a simulation program that models fish moving in a current; it uses a number of operations, including reads. For more information regarding our benchmarks, see [SSFK95].

In examining Table 2, we find that many of the bench-

Program	Description	Time (in sec)	
		Main proc	Co-proc
bitonic	Bitonic sort	17.0	16.0
cannon	Cannon matrix multiply		
	global blocked algorithm	13.9	12.4
	local blocked algorithm	8.5	8.5
fft	FFT using small transfers		
	phase 1 (computation)	4.0	3.7
	remap (communication)	12.5	8.8
	phase 2 (computation)	1.0	0.9
fftb	FFT using bulk transfers		
	phase 1 (computation)	9.2	8.5
	remap (communication)	2.4	2.2
	phase 2 (computation)	2.0	1.8
mm	matrix multiply		
	global blocked algorithm	29.0	23.2
	local blocked algorithm	11.2	12.4
p-ray	Ray-tracer	37.5	38.0
radix	Radix sort	35.2	55.4
radixb	Radix sort, bulk transfers	15.3	14.4
sample	Sample sort	20.9	25.3
sampleb	Sample sort, bulk transfers	7.40	7.4
shell	Shell sort	6.9	6.5
wator	N-body simulation	139	34

Table 2: *Run times for various Split-C programs. Run times on the left correspond to the main processor version of Split-C, while times on the right correspond to the version which exploits the co-processor. Measurements are for 16 processors on a Meiko CS-2.*

mark programs generate similar timings for both the co-processor and main processor Split-C libraries. This occurs for two reasons: First, much of the communication is done with bulk operations, and there is only a 20% difference in bandwidth between the two Split-C implementations. Second, most of these programs do not overlap communication and computation to a great degree. Instead, they run in computation phases, separated by barriers. These programs do not provide the co-processor implementation with an opportunity to exploit its ability to compute and handle communication at the same time. There are some programs that show a significant variation in run times; these are mm, radix, sample, and wator. Radix and sample run slower with the co-processor library because of the raw performance of the communication operations (as shown in Figure 3). Radix uses store operations which are much faster in the main processor library (13 vs. 22 μs), while sample uses on a number of operations, including gets, which are faster in the main processor library. The global blocked matrix multiply program runs faster in the co-processor version for two reasons: First, the main processor implementation does not queue more than one get request at a time, forcing the main processor to wait for the previous one to complete (this accounts for roughly half of the difference in run times). Second, mm uses bulk get operations, which have a slightly higher bandwidth in the co-processor library.

The final program with a significant difference in run times is wator. In this program each processor has a read/compute loop that reads a data point and then computes on that data. Because the main processor version only polls when it performs communication, any requests it receives while it is computing experience a long delay. (To avoid at least some of this delay, the programmer would have to add polls to the compute routine.) The co-processor implementation, in contrast, can process requests no matter what the main processor is doing. This leads to substantially smaller average wait times: We measured 25-33 μs for read requests for the wator program under the co-processor implementation, compared to 600-800 μs under the main processor implementation.

6 Related Work

In exploiting the co-processor for message handling, our study focused on Active Messages and Split-C. Active Messages have been ported to a wide variety of machines, including the Ncube/2, CM-5 [vECGS92], the Paragon [LC95], Meiko CS-2 [SS95], Alewife [BCL$^+$95], IBM SP-2 [CGvE95], a cluster of HP workstations connected by FDDI [Mar94], and a cluster of SparcStations connected by ATM [vEBBV95]. These last few implementations make use of a co-processor. In most of these implementations, the co-processor is used to implement the Remote Queue abstraction [BCL$^+$95]. The main processors can enqueue an outgoing message or dequeue an incoming message at user-level, while the co-processor manages the queue and protocol handling. The FDDI implementation uses a Medusa interface card, which enqueues and dequeues data, but is not programmable. U-NET, a fast user-level communication layer on ATM networks, is implemented using FORE ATM cards, containing i960 processors that run specialized firmware. Similarly, the Myrinet LANai co-processor is used for both Active Messages [MLMC95] and Illinois Fast Messages [PLC95]. An interesting Active Message implementation from our perspective is a version for the Paragon[LC95], which uses a standard Intel i860 as a CCP. They, too, implement Active Messages and Split-C in order to test the use of the CCP.

Split-C shares its origin with Active Messages, and most implementations of the language are based on them. Split-C has been ported to a number of machines, including the CM-5, Paragon, SP-1, SP-2, and networks of workstations. An interesting implementation is a recent prototype version of Split-C implemented on the Paragon, using the co-processor for message handling [KNW95]. By using the co-processor on the receiving side to handle requests, they provide a speed up of 25% for read/write operations.

7 Conclusions

In this paper we have studied how to exploit the capabilities of communications co-processors present in modern parallel architectures. We have studied this in the context of Active Messages and Split-C, and have implemented new versions of these libraries which execute the message handlers on the co-processor of the Meiko CS-2. We identified and solved the synchronization issues which arise when running message handlers on the co-processor in parallel with the code on the main processor. Ours is the first full implementation of Active Messages and Split-C running on a co-processor.

As our experimental results show, the latencies of Active Messages when executing handlers on the co-processor increase about two-fold. The reason is that the co-processor is is much slower than the main microprocessor. On the other hand, the co-processor has important specialized functionality which we exploit to optimize the split-phase remote memory operations of Split-C. Under this optimized Split-C implementation, the latencies are reduced: put operations are as fast as the main processor implementation, though get and store operations are still slower. However, the real benefit of the new implementation is that the main processor does not need to poll or process interrupts to handle messages, since this is done by the co-processor. This frees the main processor for computation and allows it to overlap communication and computation to a larger degree. Since the co-processor is not doing any other computation, it can handle messages more responsively. This benefit exhibits itself in some applications, such as the benchmark program wator, which achieves a factor of three performance improvement.

This work is applicable to other modern architectures which have a communications co-processor, such as the Paragon, T3D, and future machines. Many of these architectures will, like the Meiko, contain a custom designed chip that may be slower than its main processor counterpart. However, as we have shown, this does not preclude its usefulness for handling messages. Even if the co-processor can only execute kernel code, it is possible to support a fixed set of handlers required by a communication library.

References

[BCL+95] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *7th Annual Symposium on Parallel Algorithms and Architectures*, July 1995.

[BGSP94] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency Hiding in Message-Passing Architectures. In *Eighth International Parallel Processing Symposium*, April 1994.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18, July 1993.

[CGvE95] C.-C. Chang, C. Grzegorz, and T. von Eicken. Performance of Active Messages on the SP-2. Cornell University, May 1995.

[HM93] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proc. of Hot Interconnects*, August 1993.

[Kea94] J. Kuskin and et. al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st International Symposium on Computer Architecture*, April 1994.

[KNW95] A. Krishnamurthy, J. Neefe, and R. Wang. Towards Designing and Evaluating Network Interface Support: A Case Study. UC Berkeley, May 1995.

[LC95] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of Intel Supercomputer Users Group Conference*, April 1995.

[Mar94] R. P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proc. of Hot Interconnects II*, August 1994.

[MLMC95] R. Martin, L. T. Liu, V. Makhija, and D. E. Culler. Lanai Active Messages. Online at http://now.cs.berkeley.edu/AM/lam_release.html, September 1995.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing*, December 1995.

[PR94] P. Pierce and G. Regnier. The Paragon implementation of the NX message passing interface. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994.

[RLW94] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, April 1994.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.

[SSFK95] K. Schauser, C. Scheiman, J. Ferguson, and P. Kolano. Exploiting the Capabilities of Communications Co-processors. Technical report, UC Santa Barbara, December 1995.

[TM94] L. W. Tucker and A. Mainwaring. CMMD: Active Messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. Symposium on Operating Systems Principles*, 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.