A Cache-based Message Passing Scheme for a Shared-bus Multiprocessor

Bruno R. Preiss

Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, CANADA, N2L 3G1

V. Carl Hamacher

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, CANADA, M5S 1A4

ABSTRACT

This paper describes a scheme for using cache-based hardware to provide simple and efficient message passing support for message-based software systems on a tightly-coupled, shared-bus multiprocessor. This approach is based on the utilization of the existing interprocessor communications medium, the shared bus, to effect the exchange of single-word messages. Communication between processes is accomplished over logical channels using simple, blocking *send* and *receive* primitives. The physical processor/channel interface is designed so that the message transfer primitives can be implemented as single machine instructions, namely *store* and *fetch*. Special-purpose caches, called *message caches*, mediate channel operations and effect the exchange of messages over the shared bus.

July 5, 1994

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

This research was supported in part by NSERC (Canada) under Grant A5192.

A Cache-based Message Passing Scheme for a Shared-bus Multiprocessor

Bruno R. Preiss

Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, CANADA, N2L 3G1

V. Carl Hamacher

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, CANADA, M5S 1A4

1. Introduction

Complex software systems are commonly organized as a collection of concurrent processes that communicate with each other by exchanging messages. In particular, we are interested in programs that exhibit *fine granularity of parallelism* and *fine communication granularity* and that run on *small* to *moderate* size systems. *Fine granularity of parallelism* means the average block length of instructions containing no interprocess communication (IPC) is on the order of 100 instructions. *Fine communication granularity* means the average length of messages is on the order of several bytes. *Small* to *moderate* size systems have on the order of 10 to 20 processors[1].

There are a number of different programming paradigms that naturally produce the class of programs described above. For example, in the OCCAM programming language, which is based on the CSP concept (communicating sequential processes)[2], data transfers are semantically equivalent to assignment statements[3]. Consequently, the language lends itself to programs in which single-word messages are frequently exchanged.

Data-flow programs also exhibit fine granularity of parallelism and fine communication granularity: Data-flow tokens are equivalent to short messages. Data-flow actors (and macro-actors) correspond to short code blocks. The appeal of data-flow systems is their ability to support software pipelining[4]. Such pipelining results in the frequent exchange of short messages.

Certain numerical matrix algorithms have been developed that are not easily vectorizable. Such algorithms rely on the frequent exchange of single-word messages during the course of the computation[5]. Similar behaviour is exhibited by systolic array algorithms[6].

The focus of this paper is on a hardware mechanism that facilitates the frequent exchange of short messages on a shared-bus multiprocessor. The basis of this mechanism is the use of special-purpose message caches to implement a communications mechanism that behaves like a distributed mailbox[7]. A mailbox provides hardware support for IPC (e.g., see [8] and [9]). In this paper, we present a distributed (decentralized) mailbox mechanism (c.f. [10] where a centralized scheme is described) that uses the existing interprocessor communications medium, the shared bus, to effect message transfers (c.f. [11] where a special mailbox bus is proposed for IPC).

In Section 2, the basic processing model will be described. The processing model specifies the semantics of the message passing system. The basic entity of this model is an abstract entity called a *channel*, which provides an unbuffered, unidirectional communications path between two processes. Two

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

operations, *send* and *receive*, are used to effect the exchange of single-word messages over a given channel.

In Section 3, we will describe a generalized processor/channel interface model. This model describes the interface between a processor and its associated message handling hardware. This interface is a generalization of the conventional processor/memory interface. This allows the use of existing processor technology to implement the multiprocessor system.

The architecture of the message handling hardware will be described in Section 4. The organization of this hardware is based on the use of special caches, called *message caches*, to effect the channel transfers by communicating, in a broadcast mode, over the shared bus of the multiprocessor system. These caches use a protocol, described in Section 5, that efficiently utilizes the existing interprocessor bandwidth. It will be shown in Section 6 that this protocol correctly implements the desired channel semantics.

2. Processing Model

For the purpose of this paper, we will assume that the software system organization is based on a dynamic, message-based processing model. Such software systems are organized as a collection of communicating concurrent processes that can be created and destroyed during the course of a computation. These processes interact by exchanging messages via communication channels that can be dynamically allocated to and recovered from processes. Such channels can be used both for the exchange of data, as well as for the synchronization of processes. It is not our intention to preclude the existence of shared memory. However, access to shared memory is to be controlled by the exchanging of messages, rather than by the use of memory-based semaphores.

We shall adopt a simple channel model that is derived from the semantics of the OCCAM programming language[3]. A channel is an abstract entity that provides an unbuffered, unidirectional (simplex) communications path between two processes. All channels have unique channel identifiers. Processes exchange messages over a specified channel using *send* and *receive* operations. All messages have a fixed size that is equal to the word size of the processor.

A *send* operation places a single-word message on a channel that is specified as an argument of the *send* operation. A *receive* operation removes a single-word message from a channel that is specified as an argument of the *receive* operation. Since channels are unbuffered, both *send* and *receive* block until the data has been transferred. There are two possible action sequences for a data transfer between two processes depending on whether the *send* or *receive* operation is done first (see Figure 1).

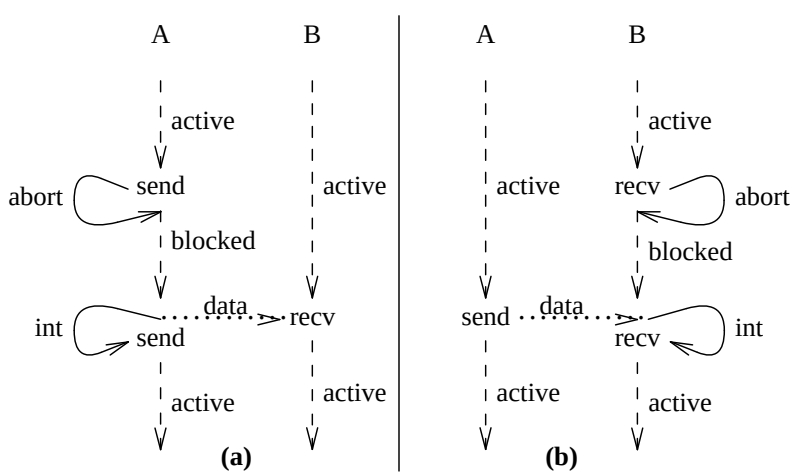

Figure 1.

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

In both action sequences, the operation that begins a channel transfer initially fails, causing the associated process to abort the operation and to become blocked. For example, if process *A* does its *send* operation before process *B* does its *receive* operation, the *send* is aborted and process *A* becomes blocked (see Figure 1(a)).

The actual data transfer occurs when the second process performs the complementary channel operation. The transfer of data causes an interrupt which releases the blocked process. This process then retries the aborted operation which then succeeds. Continuing the example above, when process *B* does the *receive* operation, the data is transferred and an interrupt is generated. As a result of the interrupt, process *A* becomes active. Process *A* then successfully retries the aborted *send* operation. A similar pattern occurs if process *B* executes its *receive* before *A* executes its *send* (see Figure 1(b)).

The interrupt mechanism whereby a blocked process is signalled when the channel transfer has been completed is called *positive wakeup.* A process may also continually retry the blocked operation until it succeeds. This latter option is called a *busy-wait*[10]. The message-passing mechanism described in this paper supports both of these general strategies for handling conditional waits.

3. Processor/Channel Interface Model

We propose a processor/channel interface model that is a generalization of the conventional processor/memory interface. In addition to the usual memory operations, *store* and *fetch*, this interface also provides two channel operations, *send* and *receive*. This generalization of the processor/memory interface is loosely based on the *fetch−and−*ϕ operations proposed for the NYU Ultracomputer, which in turn are a special case of a read-modify-write operation[12]. The intention is to provide a processor/channel interface that allows the implementation of the channel operations, *send* and *receive*, as single instructions.

The components of the processor/channel interface model are shown in Figure 2. This interface supports four operations — memory read (*fetch*), memory write (*store*), channel read (*receive*), and channel write (*send*). The operation is specified by the *read/write* and *memory/channel* control signals. The *memory address/channel identifier* bus specifies the channel identifier for channel operations and the memory address for memory operations. The *data* bus is used to transfer the data between the processor and the appropriate channel or memory location.

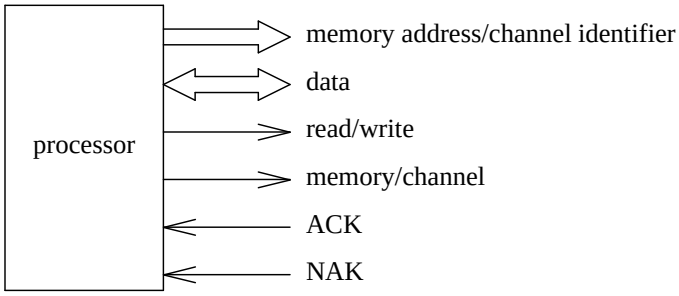

Figure 2.

Two acknowledgement control signals are used to indicate the completion of the memory or channel operation. The *ACK* (acknowledge) control signal indicates successful completion of the requested operation. The *NAK* (negative acknowledge) control signal indicates that the requested operation could not be completed.

In the case of memory operations, the *NAK* signal usually indicates the occurrence of a memory-protection violation or a page fault. In the case of channel operations, the *NAK* control signal indicates that the channel transfer operation could not be completed because another process has not yet initiated the complementary operation. In both cases, the *NAK* signal causes the processor to abort the operation

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

and to execute a hardware trap to an appropriate trap handling routine. Note that for channel transfers, a simple spin-lock can be implemented by using a trap handler that does nothing but returns from the trap and restarts the aborted operation.

The rationale for the design of the processor/channel interface was to provide a scheme in which the channel operations *send* and *receive* can be implemented as single processor instructions. Since it is not feasible to add such instructions to existing processors, the channel operations must be mapped onto memory operations instead. The processor/channel interface is the same as a conventional processor/memory interface except for the memory/channel control signal. This signal can be easily derived by mapping the channel identifier space into the memory address space and constructing the appropriate address decoding logic. In this way, *send* and *receive* operations are identical to *store* and *fetch* (respectively).

An implementation of the memory system need not provide any special hardware to support the channel operations. The processor/channel interface can be configured in such a way that all channel operations immediately cause a *NAK* signal. In this way, channels could be implemented entirely in software. however, since dynamic message-based systems rely heavily on interprocess communication via channels, external message processing hardware is needed to support channel operations efficiently (e.g., see [9], [11] and [13]).

4. Message Caches

In this section, we will describe an approach to implementing channels in a shared-bus multiprocessor based on associating a message cache with each processor in the system. The goals in the design of the system are:

- to provide an easily expanded symmetric scheme in which messages are treated uniformly, i.e., messages can be transmitted between a pair or processes, whether they both execute on a single processor or two different processors;
- to provide an approach in which channels are not coupled to processors but to processes, i.e., processes (and their associated channels) are free to migrate from processor to processor;
- to economize the utilization of interprocessor communications bandwidth by using the ability of caches to reduce traffic[14].

The architecture of the proposed approach to implementing channels using dedicated hardware is shown in Figure 3. The message processing hardware consists of a message cache unit which is situated between the processor and the shared interprocessor bus. The *send* and *receive* channel operations are handled by the message caches. The message cache units respond to channel operations initiated by the processors with appropriate acknowledgement signals. The message cache units effect the channel transfers using the shared interprocessor bus using protocols similar to *snooping* data caches[15].

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

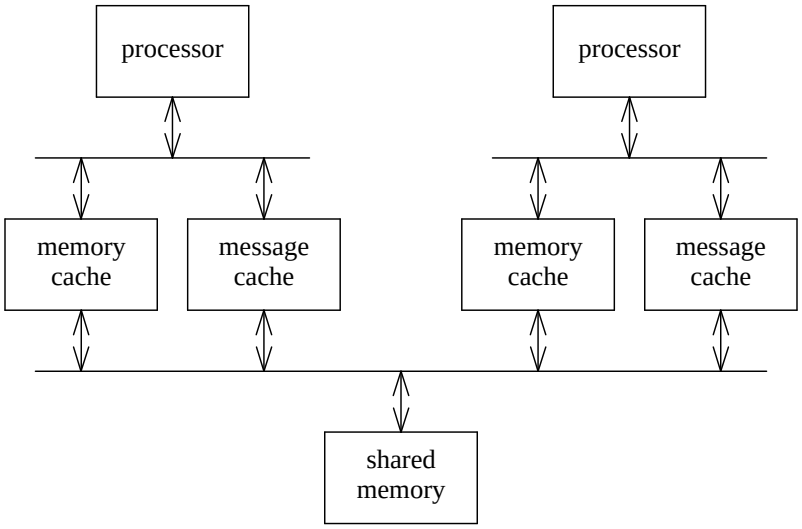

Figure 3.

Each message cache has an entry reserved for every channel in the system. The message cache entry number is directly specified by the channel identifier. Each entry consists of two parts — a one-word message field and a state field. The state field for a given channel is not necessarily the same in all message caches. A message cache entry can be in one of the following states:

idle (I):
No transaction has been initiated (by a local process) on the channel.

send pending (SP):
A local process has initiated the *send* half of a channel transfer. The message is contained in the message field of the cache entry.

receive pending (RP):
A local process has initiated the *receive* half of a channel transfer.

send acknowledge pending (SAP):
A channel transfer has been completed. The local sending process has not yet received its acknowledgement.

receive acknowledge pending (RAP):
A channel transfer has been completed. The message is contained in the message field of the cache entry. The local receiving process has not yet received the message and acknowledgement.

Figures 4-7 illustrate the interaction of processors and message caches in order to effect channel transfers. (We have adopted the graphical notation of[16] in Figures 4-7. Note that Figure 4 is a template for the other figures.) Figure 5 illustrates the actions that occur in a channel transfer between two processes (i.e., from A to B) that are running on two different processors. In this example, process A does the *send* operation first. Its cache unit issues a shared-bus operation called ϕ_1. The ϕ_1 operation interrogates all other caches to see if a process is waiting for the message. Since no cache has a *receive* pending, they all respond with negative acknowledgement. The NAK is passed back to the processor issuing the *send* operation. This causes the processor to abort the operation and to suspend process A (Figure 5(a)).

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

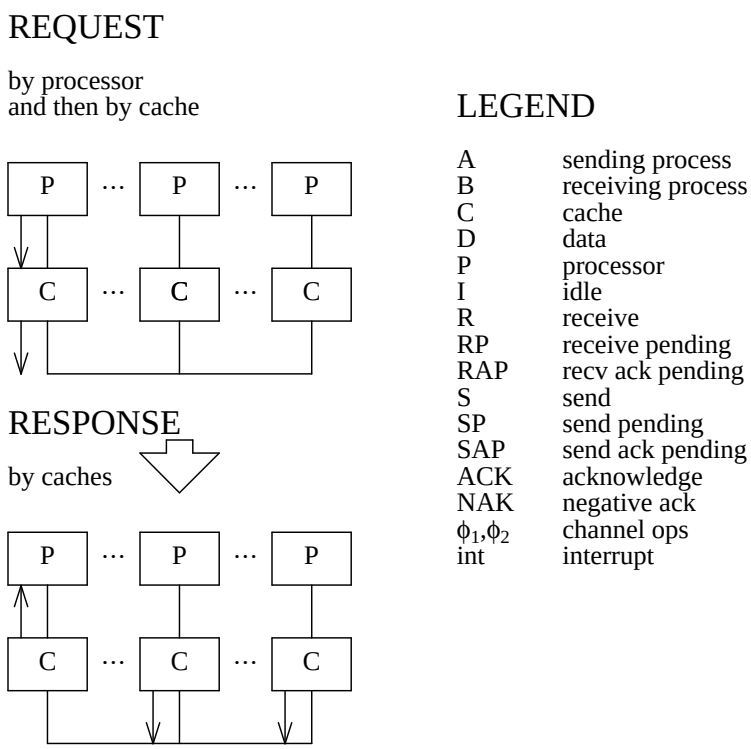

Figure 4.

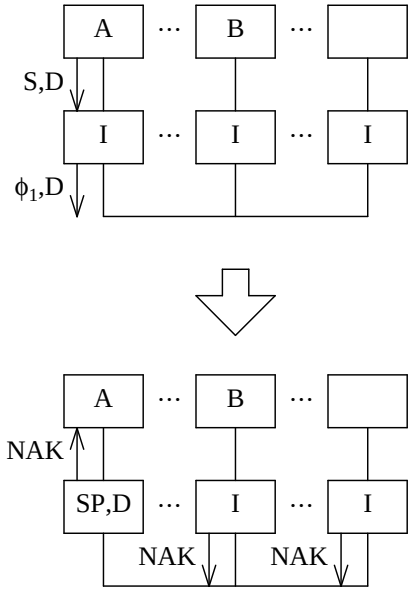

Figure 5(a).

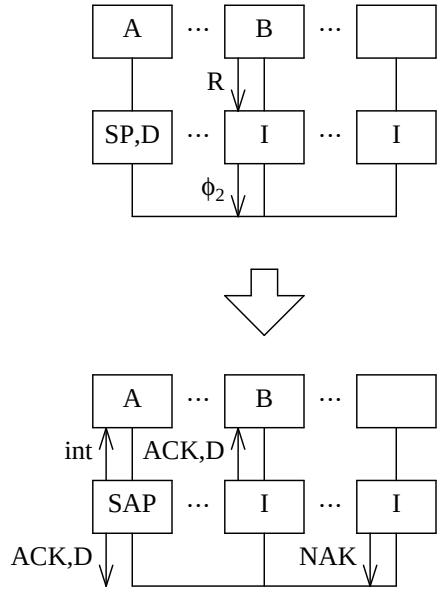

Figure 5(b).

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

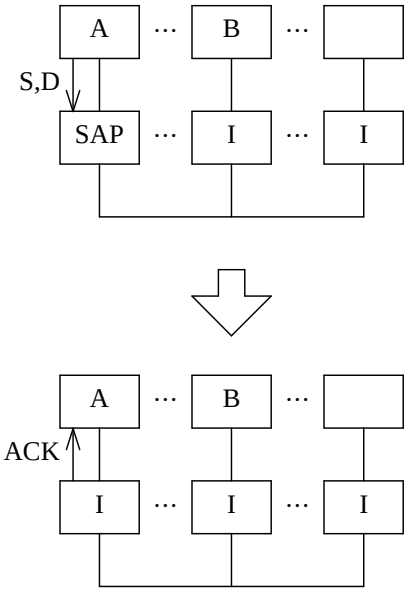

Figure 5(c).

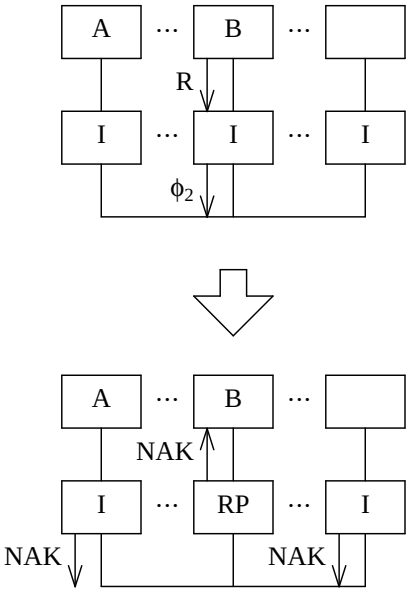

Figure 6(a).

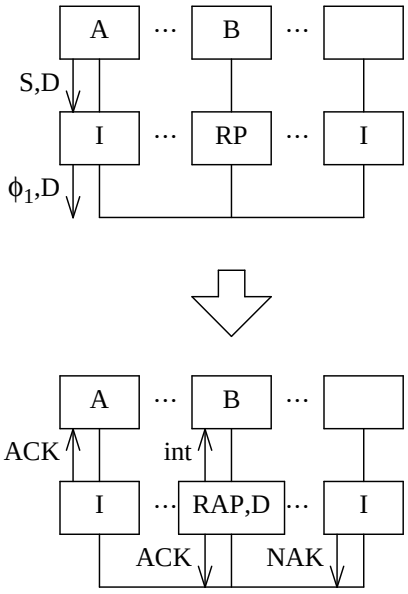

Figure 6(b).

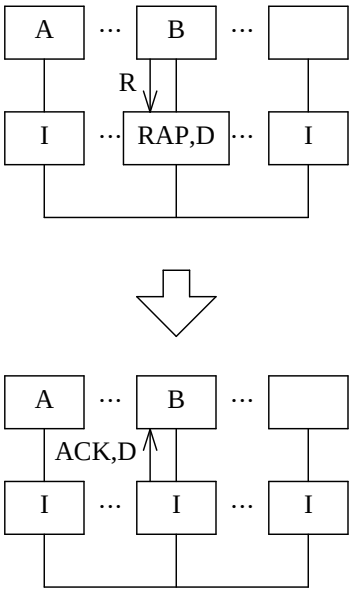

Figure 6(c).

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

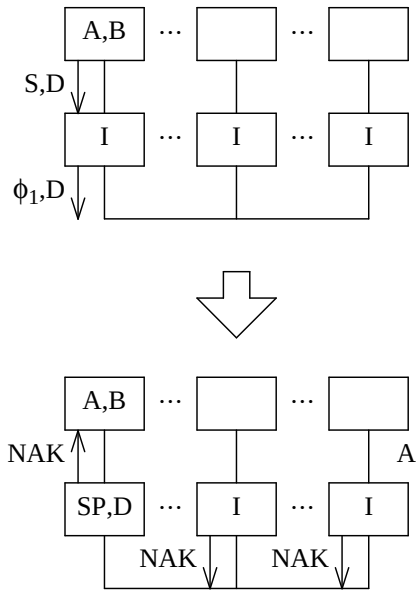

Figure 7(a).

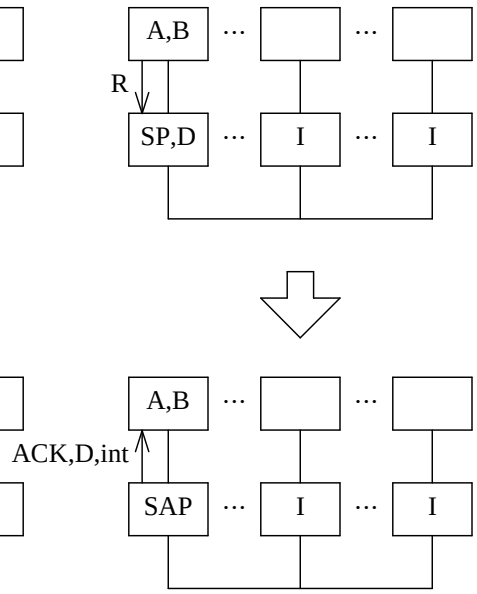

Figure 7(b).

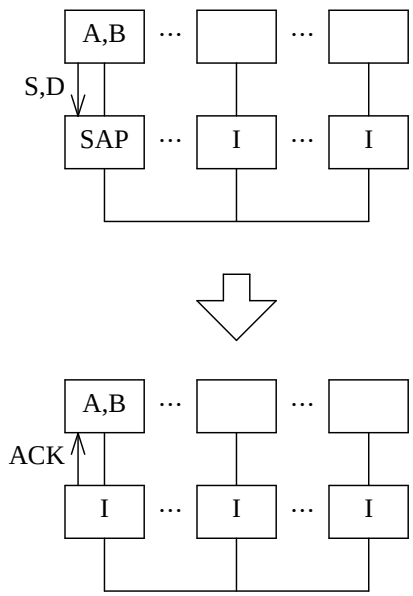

Figure 7(c).

Eventually, process *B* does the *receive* operation. Its cache unit issues a shared-bus operation called ϕ_2. The ϕ_2 operation interrogates all other caches to see if message data is available. The cache unit containing the data responds with the data and an acknowledgement. The data and ACK are passed back to the processor issuing the *receive* operation. The responding cache unit also issues an interrupt to its processor (Figure 5(b)).

Finally, process *A* retries the aborted operation. Its cache unit responds immediately with ACK and returns to the idle state (Figure 5(c)).

Figure 6 illustrates the actions that occur in a channel transfer between processes running on different processors in which the *receive* operation occurs first. Process *B* issues a *receive* operation which causes its cache unit to issue a ϕ_2 operation. Since no cache has a *send* pending, they all respond with

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

negative acknowledgement. The NAK is passed back to the processor issuing the *send* causing the processor to abort to the operation and to suspend process *B* (Figure 6(a)).

Eventually, process *A* does the *send* operation causing its cache unit to issue a ϕ_1 operation. The ϕ_1 operation interrogates all other caches to see if a *receive* is pending. The cache unit waiting for data accepts the data and issues an acknowledgement. The ACK is passed back to the processor issuing the *send* operation. The responding cache unit also issues an interrupt to its processor (Figure 6(b)).

Finally, process *B* retries the aborted operation. Its cache unit responds immediately with the data and an ACK and returns to the idle state (Figure 5(c)).

Figure 7 illustrates the sequence of actions that occur when a message transfer takes place between processes running on the same processor. In this example, the *send* occurs first. Note that when both processes run on the same processor, only one shared-bus operation (either ϕ_1 or ϕ_2) needs to be performed.

5. Protocol Mechanics

The message cache units effect channel transfers by communicating with each other in a broadcast mode over the shared bus. In addition to memory read and write operations, the shared interprocessor bus supports two channel related operations called ϕ_1 and ϕ_2. The operations ϕ_1 and ϕ_2 are global operations. That is, all the message cache units (except the initiating unit) perform the operation when the request is placed on the shared interprocessor bus. The message cache entry state transition tables for these two operations are shown in Tables 1-2.

$\phi_1(chan,mesg,ack)$			
state	action	ack	next state
I	—	NAK	I
SP	— (error)	NAK	SP
RP	$cache(chan) \leftarrow mesg$ (interrupt)	ACK	RAP
SAP	— (error)	NAK	SAP
RAP	—	NAK	RAP

Table 1.

$\phi_2(chan,mesg,ack)$			
state	action	ack	next state
I	—	NAK	I
SP	$mesg \leftarrow cache(chan)$ (interrupt)	ACK	SAP
RP	— (error)	NAK	RP
SAP	—	NAK	SAP
RAP	— (error)	NAK	RAP

Table 2.

A message cache unit issues the ϕ_1 operation to effect a channel *send* operation. The ϕ_1 operation is analogous to a memory write operation — the issuing unit places the channel identifier (address) and message (data) on the interprocessor bus. If a message cache unit finds its cache entry in state *RP*, (receive pending) (meaning a local process is waiting to receive a message), the message is stored in the

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

cache and a positive acknowledgement is returned to the message cache unit issuing the ϕ_1 operation.

A message cache unit issues the ϕ_2 operation to effect a channel *receive* operation. The ϕ_2 operation is analogous to a memory read operation — the issuing unit places the channel identifier (address) on the interprocessor bus and expects to receive the message (data) in response. If a message cache unit finds its cache entry in state *SP*, (send pending) (meaning a local process is waiting to send a message and the message is stored in the cache entry), the message is retrieved from the cache and returned along with a positive acknowledgment to the message cache unit issuing the ϕ_2 operation.

In both cases above, although all the message cache units (except the one issuing the operation) perform the operation, at most one message cache unit will respond with a positive acknowledgement to the ϕ_1 or ϕ_2 operation in a semantically correct program. This is because the semantics of channel operations ensure that only one process may send on a given channel and only one process may receive on a given channel. Those actions in Tables 1-2 marked *error* are conditions that should never occur in semantically correct programs. They result when more than one process attempts to *send* or *receive* on the same channel, which violates the channel semantics. When an error condition is encountered, the message cache unit can interrupt its associated processor to inform it of the error condition. In addition, Tables 1-2 show that in response to a successful ϕ_1 or ϕ_2 operation, the message cache unit may (but need not) interrupt its attached processor to signal that a channel transfer has been completed.

The manner in which the interprocessor bus operations ϕ_1 and ϕ_2 are used to effect the channel operations *send* and *receive* is specified in Tables 3-4. These tables show the cache entry state transitions and actions that the message cache unit performs in response to *send* and *receive* operations issued by the attached processor.

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

send(*chan*,*mesg*,*ack*)			
state	action	ack	next state

I	ϕ_1(*chan*,*mesg*,*ack*)
	if *ack*=*NAK* then
	cache(*chan*)←*mesg*
	(interrupt)
	else
	—
		
		NAK
		
		ACK
			
			SP
			
			I

SP	—	NAK	SP
RP	*cache*(*chan*)←*mesg* (interrupt)	ACK	RAP
SAP	—	ACK	I
RAP	—	NAK	RAP

Table 3.

receive(*chan*,*mesg*,*ack*)			
state	action	ack	next state

I	ϕ_2(*chan*,*mesg*,*ack*)
	if *ack*=*NAK* then
	—
	else
	—
		
		NAK
		
		ACK
			
			RP
			
			I

SP	*mesg*←*cache*(*chan*) (interrupt)	ACK	SAP
RP	—	NAK	RP
SAP	—	NAK	SAP
RAP	*mesg*←*cache*(*chan*) (interrupt)	ACK	I

Table 4.

When a *send* or *receive* operation fails (i.e., returns negative acknowledgment, *NAK*), the processor aborts the offending instruction, and the process performing the operation becomes blocked. Since the message cache entry state reflects the fact that a channel transfer is pending, it is required that the blocked processes do not migrate to a different processor until the channel operation has been successfully retried and completed.

6. Analysis of the Message Cache Protocols

The message cache protocols described above have been designed to implement the desired channel semantics. That is, a channel transfer between exactly two processes (that execute either on the same or two distinct processors) must operate correctly for all possible sequences of *send* and *receive* operations. Correct operation means that for every successful *send* and *receive* pair, exactly one message is transferred, no messages are lost, and no spurious messages are created. Furthermore, the protocols must detect all illegal sequences of *send* and *receive* operations. If, in some program, more that two processes attempt to use the same channel, it is a program error, not a protocol error. The protocols may, but are not required to detect this situation. The concern is that for all semantically correct programs, (i.e., ones in which exactly two process communicate on each channel), the message cache protocols must operate

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

correctly.

The analysis of the message cache protocols is greatly simplified by the fact that one needs to consider the state of only two message caches. This is because at most two message caches will be involved in a semantically correct channel transfer between two processes, since processes are prevented from migrating from processor to processor during the course of a channel transfer. Thus, there are at most 25 composite states in which any pair of message caches can be at any point during the course of a channel transfer on a given channel.

In order to verify the message cache protocols, a state transition table was constructed for all *accessible* states of a pair of message caches involved in a channel transfer between two processes. (The state of a pair of message caches, A, and B, is represented by an ordered pair of message cache entry states, (s_A,s_B).) An *accessible* state is defined as a state that can be reached through zero or more non-error transitions from the state (I,I). (A non-error transition is a transition for which the protocols do not detect an error (see Tables 1-2).) At this point, no assumptions are made about where the pair of communicating processes are running. There are four possible situations: i) both processes run on processor A; ii) both processes run on processor B; iii) the sending process runs on processor A, the receiving process runs on processor B; and iv) the sending process runs on processor B, the receiving process runs on processor A. These four situations lead to four possible actions: i) processor A does a *send* operation, ii) processor B does a *send* operation, iii) processor A does a *receive* operation, and iv) processor B does a *receive* operation. The state transition table shown in Table 5 lists the next state for each of the four possible actions.

Of the 25 possible system states, exactly 15 are accessible. For each non-error transition in the state table, valid messages are not destroyed and spurious messages are not created. Furthermore, for each positively acknowledged *send* and *receive* pair, exactly one message is transferred. Thus, for all accessible states, the message cache protocols operate correctly.

state (s_A, s_B)	A:send	A:recv	B:send	B:recv
(I,I)	(SP,I)	(RAP,I)	(I,SP)	(I,RAP)
(I,SP)	—	(I,SAP)	(I,SP)	(I,SAP)
(I,RAP)	(I,RAP)	—	(I,RAP)	(I,RAP)
(I,SAP)	—	(RAP,SAP)	(I,I)	(I,SAP)
(I,RAP)	(SP,RAP)	—	(I,RAP)	(I,I)
(SP,I)	(SP,I)	(SAP,I)	—	(SAP,I)
(SP,RAP)	(SP,RAP)	(SAP,RAP)	(SP,RAP)	(SP,I)
(RAP,I)	(RAP,I)	(RAP,I)	(RAP,I)	—
(RAP,SAP)	(RAP,SAP)	(RAP,SAP)	(RAP,I)	(RAP,SAP)
(SAP,I)	(I,I)	(SAP,I)	—	(SAP,RAP)
(SAP,RAP)	(I,RAP)	(SAP,RAP)	(SAP,RAP)	(SAP,RAP)
(SAP,RAP)	(I,RAP)	(SAP,RAP)	(SAP,RAP)	(SAP,I)
(RAP,I)	(RAP,I)	(I,I)	(RAP,SP)	—
(RAP,SP)	(RAP,SP)	(I,SP)	(RAP,SP)	(RAP,SAP)
(RAP,SAP)	(RAP,SAP)	(I,SAP)	(RAP,I)	(RAP,SAP)

Table 5.

The eight empty transition table entries in Table 5 correspond to erroneous operations detected by the protocols. An examination of these transitions reveals that they belong to two distinct classes: The first class of error occurs when both processors attempt to perform the same operation on the same channel. Since channels are simplex, this is a violation of channel semantics. The second class of error occurs after a channel transfer has completed and one of the processes attempts to use the channel in the opposite direction. Since channels are unidirectional, this too is a violation of the channel semantics. These errors are detected by the message cache protocols. When a message cache unit detects an error, it can interrupt the processor to abort the computation. Since all other transitions are error-free, the message cache protocols correctly implement the desired semantics.

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

7. Conclusions

In this paper, a cache-based hardware mechanism for message passing in a shared-bus multiprocessor system has been proposed. This scheme has a number of advantages:

- The *send* and *receive* operations are no different from memory store and fetch instructions for the viewpoint of the processor. Consequently, by mapping the channels directly into the memory address space, the same machine instructions may be used for both memory and channel operations.
- By using single machine instructions (store and fetch) for the *send* and *receive* operations, the page-fault instruction abort and restart mechanisms provided by modern processors that support demand-paged execution can be used to facilitate the aborting and restarting of *send* and *receive* operations.
- The same mechanisms used to support multiple processes and virtual memory can be used to support virtual channels. In particular, processes are not tied to physical channel identifiers; they can be mapped from virtual channel identifiers to physical channel identifiers using an existing memory management unit.
- The memory protection mechanisms can be used to provide channel protection. Thus, channels allocated to a given process can be protected from unauthorized access by other processes in the same way as memory is protected.
- The message passing is accomplished using the same processor interconnect that supports the shared memory access. A special message passing network is not required.
- The message passing protocols use the shared-bus resource efficiently. Each transfer between a pair of processes running on two different processors takes exactly two shared-bus transactions. Transfers between a pair of processes running on the same processor take exactly one shared-bus transaction. Furthermore, all spin locks or channel polling loops are handled locally between the processor and its message cache unit without loading the shared bus.
- Channel numbers allocated to processes are independent of the physical processor on which they are running. In effect, messages are addressed to channels, not processors. Consequently, processes may migrate transparently from processor to processor (i.e., they need not know that they have migrated).

8. References

1. A. Z. Spector, ‘‘Multiprocessing Architectures for Local Computer Networks,’’ Ph.D. Dissertation, Report STAN-CS-81-874, Stanford University, 1981.
2. C. A. R. Hoare, ‘‘Communicating Sequential Processes,’’ *Communications of the ACM*, Vol. 21, No. 8, August 1978. pp. 666-667,
3. D. May, ‘‘OCCAM,’’ *SIGPLAN Notices*, Vol. 18, No. 4, April 1983. pp. 69-79,
4. J. B. Dennis, G.-R. Gao, and K. W. Todd, ‘‘Modeling the Weather with a Data Flow Supercomputer,’’ *IEEE Trans. Comput.*, Vol. C-33, No. 7, July 1984. pp. 592-603,
5. D. P. O’Leary and G. W. Stewart, ‘‘Data-Flow Algorithms for Parallel Matrix Computations,’’ *Communications of the ACM*, Vol. 28, No. 8, August 1985. pp. 840-853,
6. H. T. Kung, ‘‘Why Systolic Architectures?,’’ *Computer*, Vol. 15, No. 1, January 1982. pp. 37-46,
7. B. R. Preiss, ‘‘Data Flow on a Queue Machine,’’ Ph.D. Dissertation, University of Toronto, Dept. of Elec. Eng., Toronto, Canada, 1987.
8. W. S. Ford and V. C. Hamacher, ‘‘Hardware Support for Inter-process Communication and Processor Sharing,’’ *Proc. 3rd Ann. Symp. Comp. Arch.*, January 1976. pp. 113-118,

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.

9. H. K. Reghbati, ‘‘An Efficient Time-shared Link Processor for Supporting Communication in Parallel Systems with Dynamic Structure,’’ *Proc. 6th Ann. Symp. Comp. Arch.*, April 1979. pp. 147-153,

10. H. K. Reghbati and V. C. Hamacher, ‘‘Hardware Support for Concurrent Programming in Loosely coupled Multiprocessors,’’ *Proc. 5th Ann. Symp. Comp. Arch.*, April 1978. pp. 195-201,

11. W. S. Ford and V. C. Hamacher, ‘‘Low level Architecture Features for Supporting Process Communication,’’ *The Computer Journal*, Vol. 20, No. 2, May 1977. pp. 156-162,

12. A. Gottlieb and et. al., ‘‘The NYU Ultracomputer — Designing a MIMD Shared-Memory Computer,’’ *IEEE Trans. Comput.*, Vol. C-32, No. 2, February 1983. pp. 175-189,

13. F. J. Burkowski, ‘‘A Vector and Array Multiprocessor Extension of the Sylvan Architecture,’’ *Proc. 11th Ann. Symp. Comp. Arch*, June 1984. pp. 4-11,

14. J. R. Goodman, ‘‘Using Cache Memory to Reduce Processor-Memory Traffic,’’ *Proc 10th Ann. Symp. Comp. Arch.*, June 1983. pp. 124-131,

15. R. H. Katz, S. J. Eggers, D. A. Wood, and R. G. Sheldon, ‘‘Implementing A Cache Consistency Protocol,’’ *Proc. 12th Ann. Symp. Comp. Arch.*, June 1985. pp. 276-283,

16. P. Bitar and A. M. Despain, ‘‘Multiprocessor Cache Synchronization Issues, Innovations, Evolution,’’ *Proc. 13th Ann. Symp. Comp. Arch.*, June 1986. pp. 424-431,

Copyright (c) 1988 by The Institute of Electrical and Electronics Engineers, Inc.