LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Two-Phase Transactions

Computation Structures Group Memo 318
September 21, 1990

Jonathan Young

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Two-Phase Transactions

Jonathan Young

September 21, 1990

Abstract

The Computational Structures Group at MIT is currently building Monsoon, an implementation of the Explicit Token Store architecture. Our research has indicated the need for memory transactions of arbitrary latency in order to provide *scalability* of computer architectures. On Monsoon, memory transactions are referred to as *two-phase transactions*, and include imperative Fetch and Store requests, I-structure (I-Fetch, I-Store, and L-Store) requests, Lock (Take and Put) requests, and two local requests for use in system (storage management) code. Full documentation is given on each memory request, including a token-level description, constraints on user code using each request, and RTL suitable for the implementation of each request on a Monsoon processing element.

1 Introduction

Research at MIT has indicated the need for memory transactions of arbitrary latency in order to provide *scalability* of computer architectures[1]. This document explains how these *two-phase transactions* take place on the Monsoon implementation of the ETS architecture[5] which the Computational Structures Group is currently building.

A memory transaction consists of a series of messages (tokens), initiated by a single *memory request*. A memory request consists of the *operation* to be performed, a *node* and *location* at which to perform the operation, and a possible *parameter*. Operations are either *fetch-like* or *store-like*. The parameter for a storelike operation is the value to be stored at the location. The parameter for a fetch-like operation is a *continuation*. This continuation encodes not only the address of the instruction to which the value fetched is returned, but also any other handlers for other messages (e.g. defer messages) necessary for the semantics of the particular transaction.

Only a Monsoon processing element (PE) may issue a memory request. However, both I-Structure (IS) boards and PEs may process memory requests, and thus both architectures constrain the structure of memory transactions. In particular, the PE can *add* to various fields of a continuation, while the IS can only *exclusive-or* fields.

Each memory location on Monsoon contains both a *value* and some *presence bits*. Each operation specifies a different operation on the presence bits, but in general, when presence bits are *empty*, the value is not meaningful, while a true data value is stored in the location if the presence bits are not empty. A memory operation is *clear-like* if it is store-like and the value is ignored, for example, when the operation sets the presence bits to *empty*.

Related sets of memory operations form memory *paradigms*. In particular, the paradigms supported by Monsoon include write-once *I-Structures*[3, 2, 9], locks[8], and several forms of imperative read and write. In general, accessing the same location using operations from different paradigms is not well defined. However, if it is known that the location is *empty*, then any paradigm may be used. This restriction is documented further in [7].

IP	Port Left	Port Right	Paradigm
4	I-Fetch	I-Store	I-Structures
8	Fetch	Store	Imperative
12	Take	Put	Lock
16	Examine-Lock	(unassigned)	Lock
20	PB-dispatch	(unassigned)	
24	(unassigned)	Clear	
28	PLTake	PLPut	Processor-local Take/Put
32	(unassigned)	L-Store	I-Structures
36	Code-Fetch	Code-Store	Instruction memory
40	Frame-Fetch	Frame-Store	Frame memory

Table 1: Assigned Request IPs

Memory operations are transmitted across the Monsoon *network*, which guarantees that successive messages between any two nodes will be delivered in a FIFO manner[4]. In addition, certain messages can be designated as *circuit-switched*, in which case a delivery acknowledgement is generated before any other token leaves the same originating node. Most store-like messages are circuit-switched in order to certify that the write has been performed. In addition, certain messages must be circuit-switched to ensure that messages arrive in the correct order.

Note that while processing an instruction, a Monsoon PE may do several operations before issuing a memory request. This is irrelevant for the purposes of this paper, and we avoid the issue by writing the pseudo-MONASM syntax "`(IFCH args)`" to denote "any Monsoon instruction which issues an I-Fetch memory request".

This is a working document. Currently, only the Imperative (Fetch and Store requests), I-Structure (I-Fetch, I-Store and L-Store), Lock (Take and Put), and PLMem (PLTake and PLPut) memory paradigms are documented. Additional features of the above paradigms, including the Examine-Lock transaction, are not yet specified.

2 Request Numbers

Because Monsoon processing elements may process requests, there is a well-defined mapping between a request numbers and an (IP,Port) pair. A memory request is encoded as a 24-bit number; the port is obtained as the high (24th) bit of this number, while the IP is the low 23 bits. Conversely, appending one bit of port and 23 bits of IP results in a request number.

Table 1 lists all requests which have been assigned IPs and ports. Note that in general, port left is used for read (fetch-like) operation, while port right is a write (store-like) operation.

Note that the IP is always a multiple of four, because some operations require more than one instruction on a Monsoon PE. The PIU on the PE provides special support for converting a pointer into a request; part of this operation consists of moving a *displacement*, shifted left by two bits, into the request field. The appropriate displacement for a memory request may be obtained by dividing the IP for that request by four.

The `GFCH` and `GSTR` instructions are used to perform *generic* memory operations on requests which were previously created from a pointer and an operation number. It is the responsibility of the user to follow the paradigm of the appropriate operation – the generic operations are simply

more expensive means to execute the same operations

The **Bulk-Clear** and **RW-Set-PB-*n*** operations have not yet been assigned numbers.

3 Imperative Operations

Imperative operations allow, with proper synchronization in user code, the reading and writing of any Monsoon memory location. Examples of such operations include **Frame-Fetch** and **Frame-Store**, which read and write the values stored in a frame, **Code-Fetch** and **Code-Store**, which read and write the values stored in instruction memory, and **Fetch** and **Store**, which read and write an arbitrary memory locations. In addition, the presence bits may be read at a location using **PB-dispatch**, and written using **Clear** or any of the the **RW-Set-PB-*n*** family of operations. Only the imperative **Fetch**, **Store** and **Clear** operations are documented here.

3.1 User-level descriptions of Fetch, Store and Clear

The **Fetch** memory request is extremely simple. The pseudo-MONASM syntax is:

```
F:      (FCH args),RA.L
        ...
RA:     (dest)
```

This sequence of instructions obeys the following contract: if the Fetch instruction at F executes, and the location is present, then a token will eventually arrive at the left port of RA (RA.Left).

Imperative **Store** operations exhibit the simplicity of all store-like operations. If a destination is supplied, the *signal token* (with V undefined) arrives when the network certifies that the store request was received.

Case 1 (no signal):

```
S1:     (STR args)
        [> or || stop]
        ...
```

Case 2 (with signal):

```
S2:     (STR args),SD
        [> or || stop]
        ...
SD:     dest
```

Contract: If the **Store** (or store-like) instruction at S2 executes, then eventually a token arrives at SD.

The **Clear** operation is store-like, but no value is stored. User code is similar to **Store**.

3.2 Token-level description

Each token in the imperative fetch/store paradigm can be assigned one of three types, with different properties, as summarized in Table 2.

Name	Circuit Switched?	Destination	Value
Fetch	no	FCH(LOC)	RA
Store	YES	STR(LOC)	Value
Clear	YES	CLR(LOC)	⟨Unspecified⟩
Value	no	RA.L	Value

Table 2: Tokens in the Imperative Paradigm

```
%%RWMEM f1=ignored f2=r (r is expected to be 0)
case port of
  left :
    (Fetch)
    case DMP[ea] of
      read-only :
        temp ← DM[ea]
        C' ← V
        V' ← temp
        newCD(C', V')   net or enqueue   (Value token)
        stop
    esac
  right :
    (Store)
    case DMP[ea] of
      empty :
        DMP[ea] ← read-only
        DM[ea] ← V
        stop
    esac
esac
```

Figure 1: RTL for %%RWMEM

```
%%CLEAR f1=ignored f2=r (r is expected to be 0)
case port of
  right :
    (Clear)
    DMP[ea] ← empty
    stop
esac
```

Figure 2: RTL for %%CLEAR

Name	Circuit Switched?	Destination	Value
Fetch	no	IFCH(LOC)	RA
Store	YES	ISTR(LOC)	Value
Defer	YES	RA.R	RA'
Deferred-value	YES	RA.L	Value
Value	no	FD.L	Value

Table 3: I-Structure Tokens

3.3 System-level description

On the PE, the `%%RWMEM` instruction handles imperative memory requests. The RTL for the `%%RWMEM` instruction appears in Figure 1, in a format similar to that in [6]. The `%%CLEAR` instruction (Figure 2) handles `Clear` requests.

4 The I-Structure Paradigm

I-Structures are write-once locations which *defer* when read before the value has been written. The I-Structure family of operations includes `I-Fetch`, `I-Store`, and `L-Store`. `L-Store` (lazy store) stores a continuation in the location with presence bits *delayed*. When the location is subsequently I-Fetched from, the continuation is ejected into the network, allowing the delayed code to compute and store a value in this location.

4.1 User-level descriptions of I-Fetch, I-Store, and L-Store

To provide for multiple deferred readers on a single I-Structure location, every `I-Fetch` operation in user code must include three instructions: an `I-Fetch` or equivalent instruction, an `%I-defer` instruction, and the actual destination. Note that the destination instruction must expect the fetched value to arrive on the left port.

```
F:      (IFCH args),RA.L
        ...
RA:     %I-DEFER [fp+r]
FD:     (dest)
```

The `%I-defer` instruction must immediately precede the destination instruction, so that that RA+1 = FD. Because this arithmetic must be performed on both the PE (using the PIU) and the IS board (using XOR), the IP of RA must be even.

This sequence of instructions obeys the following contract: if the I-Fetch instruction at F executes, and the location is eventually I-Stored to, then a token will eventually arrive at the destination FD.Left, at which point the frame slot at `r` will be empty.

The `I-Store` operation strongly resembles the `Store` operation.

4.2 Token-level description

Possible messages sent under the I-Structure paradigm are presented in Table 3. An I-fetch from a location which has already been stored to will produce only a token at FD.L. When an I-fetch

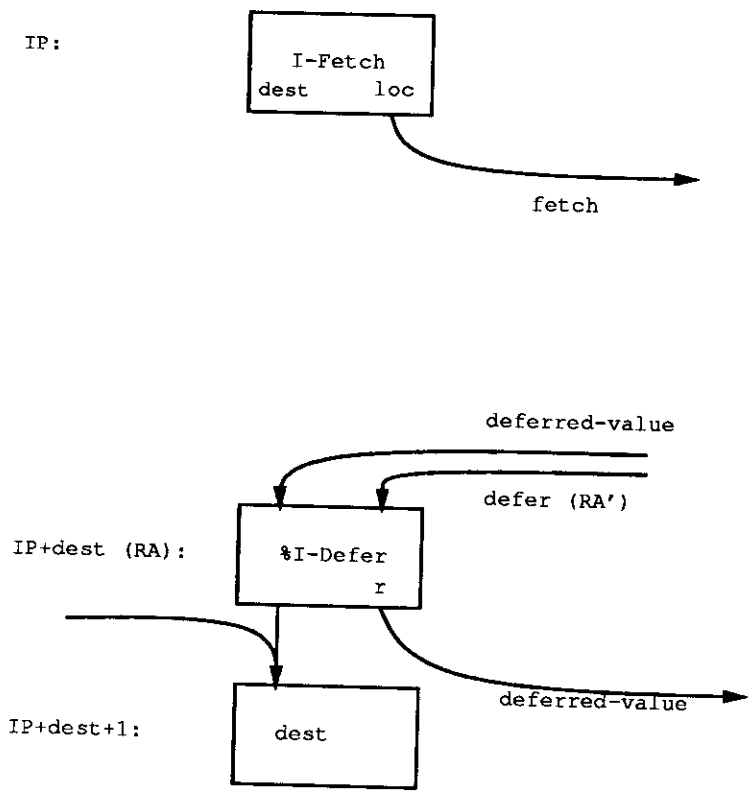

Figure 3: User Code for I-Fetch

arrives at an empty location, however, FD.L is stored with presence bits "deferred". A store at this point will result in the same value token at FD.L. However, another fetch (destination RA') will store RA' (deferred) at the location and send a defer message to RA'.R. Eventually, the value will be stored, and a bunch of deferred-value messages will be sent, terminating with a value message to the last deferred reader.

Note that the PE is not able to store FD.L; RA.L is stored instead, and the last element in a deferred list receives a deferred-value message instead of a value message. This is tolerated because the `%I-DEFER` instruction always fires on a deferred-value message, *even if no tag arrived via a defer message.*[1] Thus, I-Structure deferred lists cost one extra instruction/token when emulated by the PE.

4.3 System-level description

On the PE, the `%%ISTR` instruction handles I-Structure (I-Fetch and I-Store) requests, and the `%%LSTR` instruction handles L-Store requests. RTL for the `%%ISTR` and `%%LSTR` instructions appears in Figures 4 and 5. Note that satisfying a deferred fetch list takes one more instruction on a PE than it does on an I-Structure board because the PE cannot add one to the RA when the first reader defers. Because of this, the `%I-defer` instruction (Figure 6) must also take care to tolerate deferred-value tokens which are not preceded by a defer(RA') token.

Network tokens from the PE fall into two categories. Certain tokens (notated "circuit-switched or critical") must precede other tokens for correct I-Structure behavior. If such a token is sent over the network, it is circuit-switched as described in the introduction. On the other hand, if the token is local to the processor, then it is critically recirculated. Other tokens, notated "net or enqueue", are sent over the network without circuit-switching if remote, and are enqueued (or recirculated) in the normal fashion if local.

4.4 Example

Figure 7 and Table 4 together show an example of the messages sent when three separate I-Structure readers defer on a location before it is stored to. Note that token number 5 must arrive before token 7 and token 3 must arrive before token 8. Both conditions are ensured by the fact that defer tokens are circuit switched.

5 Lock Operations

The lock paradigm allows exclusive access to a resource which occupies one memory location. Access is obtained by the **Take** operation, which is fetch-like, and released by the **Put** operation, which is store-like. In addition, the Examine-Lock operation [TBD], which is fetch-like, is equivalent to a **take** and put sequence, and may be used to reduce memory traffic in special cases.

5.1 User-level description: Take and Put

Slightly more machinery is required in user code to obtain exclusive access to a *lock* using the **Take** and **Put** two-phase transactions. In particular, any user code doing a **Take** must retain enough state to enable another **Take** operation to be performed, in case the operation needs to be retried. (Note

[1] This is the source of the triangle inequality problem, forcing the deferred-value messages to be circuit-switched. Ken Steele has shown that this problem may be avoided at the expense of more complicated microcode and an additional presence state ("multiply-deferred").

```
%%ISTR f1=ignored f2=r (r is expected to be 0)
case port of
  left :
    (I-Fetch)
    case DMP[ea] of
      empty :
        DMP[ea] ← deferred
        DM[ea] ← V
        stop
      present :
        temp ← DM[ea]
        C' ← V
        C'_IP ← C'_IP + 1
        V' ← temp
        newCD(C', V')   net or enqueue    (Value token)
        stop
      delayed :
        DMP[ea] ← deferred
        temp ← DM[ea]
        DM[ea] ← V
        C' ← temp
        V' ← ⟨Unspecified⟩
        newCD(C', V')   net or enqueue    (Force token)
        stop
      deferred :
        temp ← DM[ea]
        DM[ea] ← V
        C' ← V
        C'_PORT ← right
        V' ← temp
        newCD(C', V')   circuit switched or critical    (Defer token)
        stop
    esac
  right :
    (I-Store)
    case DMP[ea] of
      empty :
        DMP[ea] ← present
        DM[ea] ← V
        stop
      deferred :
        DMP[ea] ← present
        temp ← DM[ea]
        DM[ea] ← V
        C' ← temp
        V' ← V
        newCD(C', V')   circuit switched or critical    (Deferred-value token)
        stop
    esac
esac
```

```
%%LSTR f1=ignored f2=r (r is expected to be 0)
case port of
  left :
    (L-Store)
    case DMP[ea] of
      empty :
        DMP[ea] ← delayed
        DM[ea] ← V
        stop
    esac
esac
```

Figure 5: RTL for %%LSTR

```
%I-DEFER f1=ignored f2=r
case port of
  left :
    (Deferred-value)
    case DMP[ea] of
      right-present :
        DMP[ea] ← empty
        temp ← DM[ea]
        C' ← temp
        V' ← V
        newCD(C', V')   net or enqueue   (Deferred-value token)
        C_IP ← C_IP + 1
        C_PORT ← left
          (Value token)
      empty :
        C_IP ← C_IP + 1
        C_PORT ← left
          (Value token)
    esac
  right :
    (Defer token – RA)
    case DMP[ea] of
      empty :
        DMP[ea] ← right-present
        DM[ea] ← V
        stop
    esac
esac
```

Figure 6: RTL for %I-DEFER

Number	Message	From	To	Contents
1	I-Fetch	A	Loc	RA(A)
2	I-Fetch	B	Loc	RA(B)
3	Defer	Loc	B	RA(A)
4	I-Fetch	C	Loc	RA(C)
5	Defer	Loc	C	RA(B)
6	I-Store	D	Loc	value
7	Deferred-value	Loc	C	value
8	Deferred-value	C	B	value
9	Deferred-value	B	A	value

Table 4: Multiply-deferring I-Structure Example

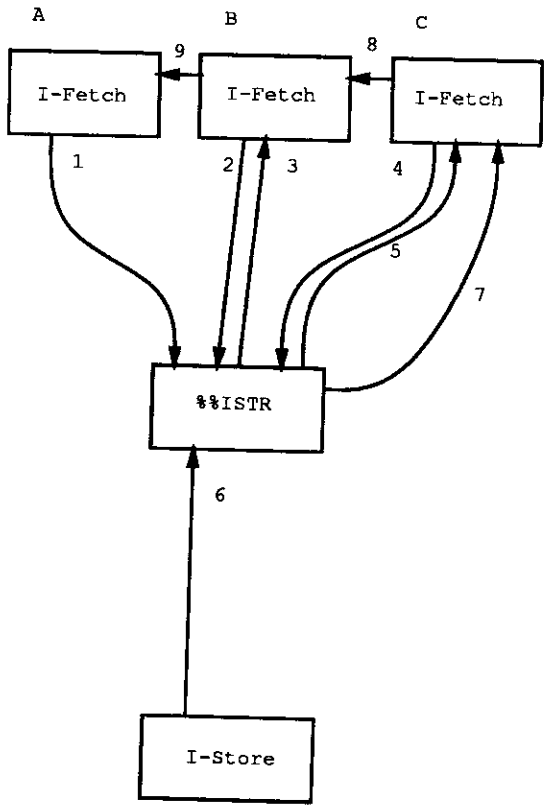

Figure 7: Multiply-deferring I-Structure Example

Name	Limit	Destination	Value
Take	1	TAKE(LOC)	RA
Put	1	PUT(LOC)	Value
Defer	(see text)	RA.R	RA'
Value	1	RA.L	Value
Value'	1	dest	value
Send-retake	0 or 1	RA+1.R	RA'
Retake	(see text)	RA+2.L	(ignored)

Table 5: Tokens in the Lock Paradigm

that this results in dataflow graphs which are *not well-behaved.*) There are three entry points and two auxiliary instructions needed for the correct functioning of multiply-deferred takers, organized as follows:

```
T:      TAKE v, RA.L
        ...
RA:     %TAKE-AUX  [FP+r], dest
RA+1:   %TAKE-AUX1
RA+2:   (code for retake)
        ...
dest:   (destination of take)
```

The rationale behind this design is actually rather subtle. The I-Structure board can only generate one output token, which must contain the value taken. After a take is satisfied, however, the rest of the takers on the deferred list must continue to defer on the location. Thus, **Take** operations which defer must retain a pointer to the location read.

Although the retake message could be sent directly from the satisfied taker to the location, we actually pass the baton to the first taker on the deferred list. This avoids dealing with the triangle inequality problem on the network.

User code for the **Put** operation is similar to **Store**.

5.2 Token-level description

Table 5 presents the possible messages in the lock paradigm. Due to the possibility of merging deferred take lists, it is difficult to characterize the number of tokens which are needed to process **Take** and **Put** operations. Each **Take** operation takes four tokens if the value is present or it is the only deferred reader on the location. Two more tokens are needed if another **Take** has already deferred on this location. Note, however, that a **Retake** operation may result in two deferred lists being merged, and this operation will take time proportional to the length of one of the lists.

5.3 System-level description: Take and Put

On the PE, the **%%LOCK** instruction handles Take and Put memory requests. The RTL for the **%%LOCK**, **%Take-Aux** and **%Take-Aux1** instructions appear in Figures 9, 10 and 11.

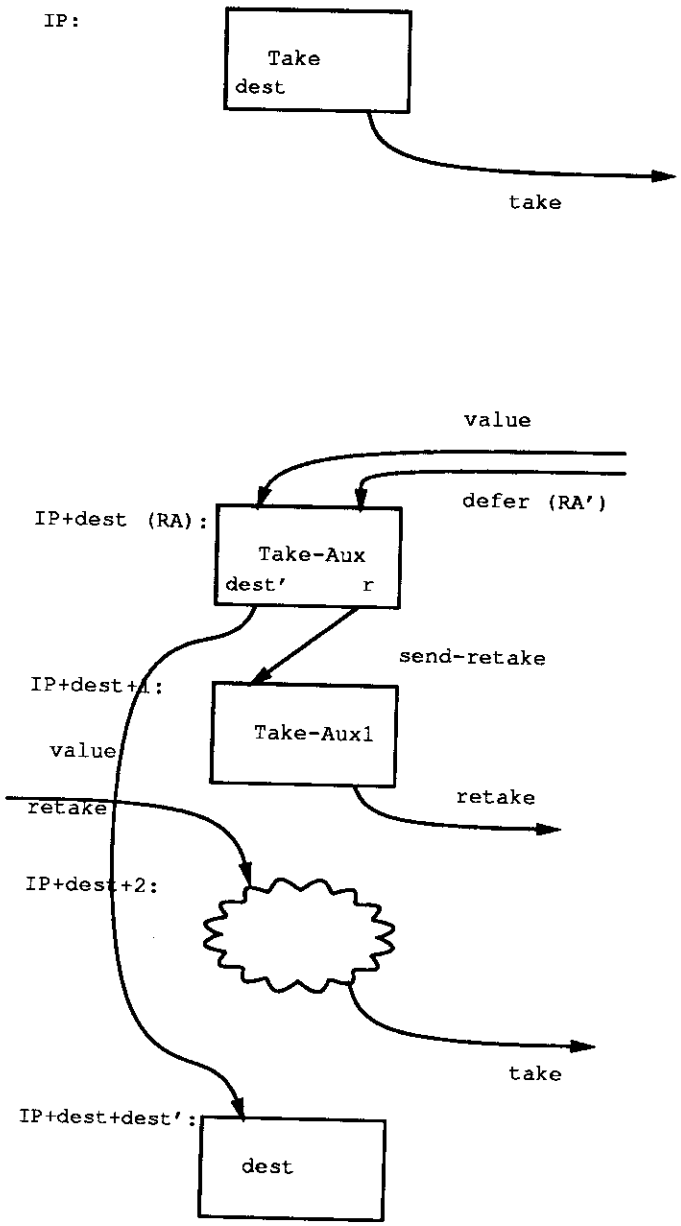

Figure 8: User code for Take

```
%%LOCK f1=ignored f2=r (expects r=0)
case port of
  left :
    (Take)
    case DMP[ea] of
      empty :
        DMP[ea] ← lock-deferred
        DM[ea] ← V
        stop
      present :
        DMP[ea] ← empty
        temp ← DM[ea]
        C' ← V
        V' ← temp
        newCD(C',V')   net or enqueue    (Value token)
        stop
      lock-deferred :
        temp ← DM[ea]
        DM[ea] ← V
        C' ← V
        C'_PORT ← right
        V' ← temp
        newCD(C',V')   net or enqueue    (Defer token)
        stop
    esac
  right :
    (Put)
    case DMP[ea] of
      empty :
        DMP[ea] ← present
        DM[ea] ← V
        stop
      lock-deferred :
        DMP[ea] ← empty
        temp ← DM[ea]
        C' ← temp
        V' ← V
        newCD(C',V')   net or enqueue    (Value token)
        stop
    esac
esac
```

Figure 9: RTL for %%LOCK

```
%Take-Aux dest r
case port of
  left :
    (Value)
    case DMP[ea] of
      empty :
        C_IP ← C_IP + dest_S
        C_PORT ← dest_PORT
          (Value' token)
      right-present :
        DMP[ea] ← empty
        temp ← DM[ea]
        C' ← C
        C'_IP ← C'_IP + 1
        C'_PORT ← left
        V' ← temp
        newCD(C', V')   net or enqueue    (Send-retake token)
        C_IP ← C_IP + dest_S
        C_PORT ← dest_PORT
          (Value' token)
    esac
 right :
    (Defer RA')
    case DMP[ea] of
      empty :
        DMP[ea] ← right-present
        DM[ea] ← V
        stop
      right-present :
        temp ← DM[ea]
        C' ← temp
        C'_PORT ← right
        V' ← V
        newCD(C', V')   net or enqueue    (Defer token)
        stop
    esac
esac
```

Figure 10: RTL for %Take-Aux

```
%Take-Aux1 f1=ignored f2=ignored
case port of
  left :
    (Send-retake - RA')
    C' ← V
    C'_IP ← C'_IP + 2
    C'_PORT ← left
    V' ← ⟨Unspecified⟩
    newCD(C',V')   net or enqueue   (Retake token)
    stop
esac
```

Figure 11: RTL for `%Take-Aux1`

5.4 Example

Figure 12 and Table 6 together show an example of the messages sent when three separate lock readers defer on a location before it is available, and two additional readers arrive before the two other deferred readers have a chance to reestablish their deferred status. This exhibits the merging of deferred Take lists, by propagating the defer message down an already established deferred list until it reaches the end. Note that the only possibility of messages arriving out of order (e.g. a value message arriving before the second defer message) is precluded by the FIFO semantics of point-to-point messages in the network.[2] Thus, none of these messages need to be circuit-switched in order to enforce the semantics of the Lock memory paradigm.

6 System-Level Operations

One other variation on two-phase transactions still remains. The system code responsible for storage management on a Monsoon processing element may need to imperatively read or write local memory. System code, however, is constrained not to relinquish the pipeline beat (or *thread*) – even during a two-phase memory transaction. Thus, we have recently introduced the *processor-local* paradigm, with operations `PLTake` and `PLPut` which are guaranteed to preserve the current thread when the location resides on the current PE.

These operations correspond most closely to `Take` and `Put` on locks, except that no deferring is allowed. That is, the value must be present when the `PLTake` occurs. Equivalently, the paradigm supports alternating imperative writes and reads.

6.1 "User-level" descriptions of PLT and PLP

Because no deferring can possibly occur, no special machinery is necessary for the `PLT` and `PLP` operations. It is not expected that users will ever use these operations; they are specifically intended for use within the exception handlers and in the ID Run-Time System.

```
PLT v,tdest >
```

[2] This is by intentional design, not by accident. This property would not hold, for example, if instead of a `Send-retake` message, we sent a `Defer` message back to the location.

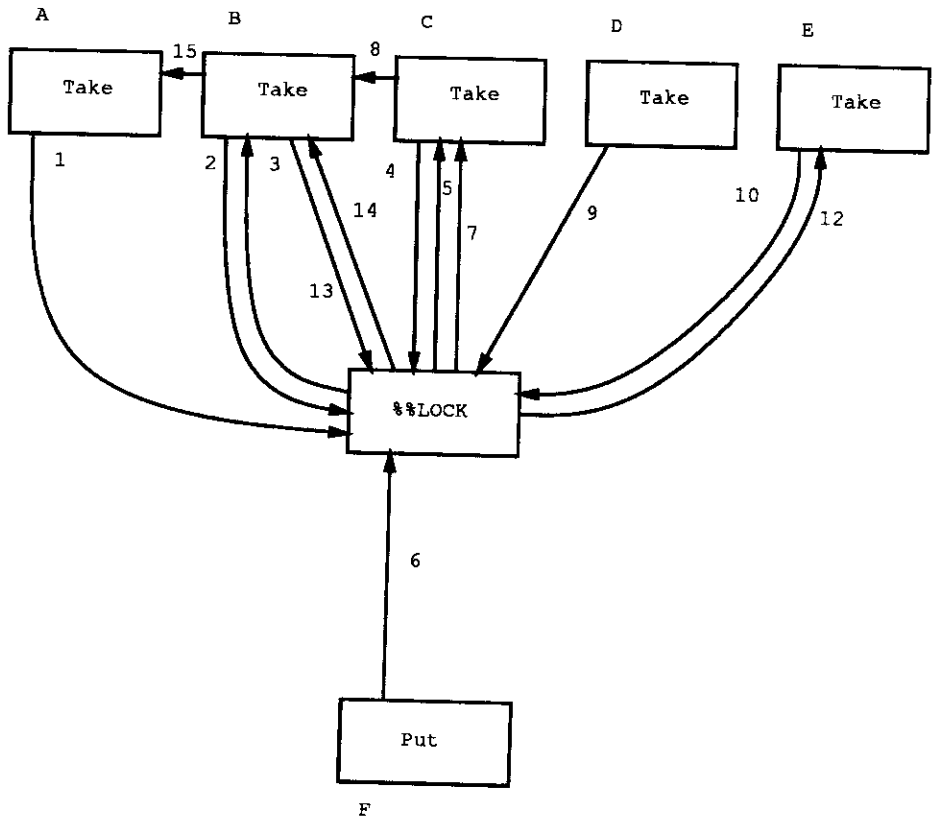

Figure 12: Multiply-deferring Lock Example

Number	Message	From	To	Contents
1	Take	A	Loc	RA(A)
2	Take	B	Loc	RA(B)
3	Defer	Loc	B	RA(A)
4	Take	C	Loc	RA(C)
5	Defer	Loc	C	RA(B)
6	Put	F	Loc	value1
7	Value	Loc	C	value1
8	Send-retake	C	B	⟨Unspecified⟩
9	Take	D	Loc	RA(D)
10	Take	E	Loc	RA(E)
11	Defer	Loc	E	RA(D)
12	(Re)Take	B	Loc	RA(B)
13	Defer	Loc	B	RA(E)
14	Defer	B	A	RA(E)
15	Put	F	Loc	value2
16	Value	Loc	B	value2
17	Send-retake	B	A	⟨Unspecified⟩
18	Take	A	Loc	RA(A)

Table 6: Multiply-deferring Lock Example

Name	Destination	Value
PLTake	PLTAKE(LOC)	RA
PLTValue	PLT-dest	Value
PLPut	PLPUT(LOC)	RA in V, Value in XB
PLPut1	PLPUT1(LOC)	Value
PLPValue	PLP-dest	Value

Table 7: Processor-Local Tokens

```
        ...
tdest:  ...

PLP v,[FP+r],pdest >
        ...
pdest:  ...
```

6.2 Token-level description

Table 7 presents the possible messages in the processor-local memory paradigm. Note that the `PLPut` request passes three logical arguments: the location, the value to store, *and a return address.* This exceptional request is both store-like and fetch-like because it makes use of some extra state (register *XB*) in the Monsoon processing element to pass both a value to store and a return address to the request handler.

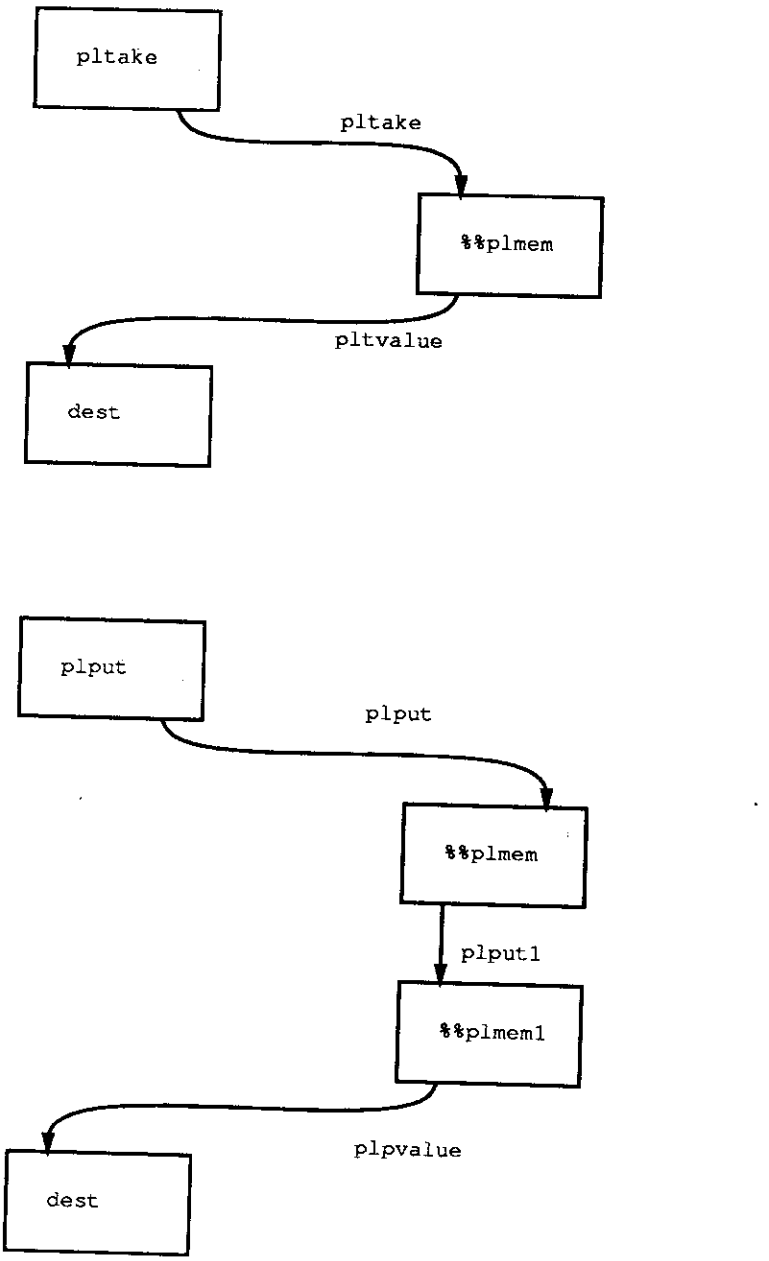

Figure 13: Processor Local Takes and Puts

```
%%PLMEM f1=ignored f2=r (expects r=0)
case port of
  left :
    (PLTake)
    case DMP[ea] of
      pl-present :
        DMP[ea] ← empty
        temp ← DM[ea]
        C' ← V
        V' ← temp
        newCD(C', V')   critical    (Value token)
        stop
    esac
  right :
    (PLPut)
    case DMP[ea] of
      empty :
        DMP[ea] ← pl-present    (May change for safety)
        DM[ea] ← V
        C_IP ← C_IP + 1
        C_PORT ← left
        V ← XB
    esac
esac
```

Figure 14: RTL for %%PLMEM

6.3 System-level description: PLTake and PLPut

On the PE, the %%PLMEM instruction handles PLT and PLP memory requests.[3] RTL for the %%PLMEM instruction, including the %%PLMEM1 auxiliary instruction, appears in Figures 14 and 15.

[3]The *pl-present* state is not defined in [7]; the current implementation of %%PLMEM uses *present* instead. This should be considered a bug.

```
%%PLMEM1 f1=ignored f2=r (expects r=0)
case port of
  left :
    (PLPut1)
    case DMP[ea] of
      pl-present  (see PLPut, above) :
        DMP[ea] ← pl-present    (Redundant, but see above)
        temp ← DM[ea]
        C' ← temp
        V' ← V
        newCD(C', V')   critical    (Put-Value token)
        stop
    esac
esac
```

Figure 15: RTL for %%PLMEM1

References

[1] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. Computation Structures Group Memo 226-6, Massachusetts Institute of Technology Laboratory for Computer Science, May 1987. In Proceedings of DFVLR - Conference 1987, "Parallel Processing in Science and Engineering," June 25-26, 1987, Bonn-Bad Godesberg (supersedes MITLCS-TM-241).

[2] Arvind and R. S. Nikhil. Executing a program on the Massachusetts Institute of Technology tagged-token dataflow architecture. In *PARLE: Parallel Architectures and Languages Europe Volume II*, volume 259 of *Lecture Notes in Computer Science*, pages 1–29. Springer-Verlag, June 1987.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer-Verlag, October 1986.

[4] C. F. Joerg. Design and Implementation of a Packet Switched Routing Chip - S.M. Thesis. Technical Report 482, Massachusetts Institute of Technology Laboratory for Computer Science, May 1990.

[5] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor - Ph.D. Thesis. Technical Report 432, Massachusetts Institute of Technology Laboratory for Computer Science, August 1988.

[6] G. M. Papadopoulos and K. R. Traub. Monsoon Assembly Language Reference. Computation structures group memo, Massachusetts Institute of Technology Laboratory for Computer Science. (In preparation).

[7] G. M. Papadopoulos and K. R. Traub. Monsoon macroarchitecture reference manual. MCRC Technical Memo ??? (in preparation), Motorola Cambridge Research Center, Cambridge, MA, 1990.

[8] R. M. Soley. On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control - Ph.D. Thesis. Technical Report 443, Massachusetts Institute of Technology Laboratory for Computer Science, June 1989.

[9] K. M. Steele. Implementation of an I-Structure Memory Controller - S.M. Thesis. Technical Report 471, Massachusetts Institute of Technology Laboratory for Computer Science, March 1990.