LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

On Memory Models and Cache Management for Shared-Memory Multiprocessors

Computation Structures Group Memo 363
March 16, 1995

Jack B. Dennis and Guang R. Gao

This memo is a paper prepared for the IEEE Symposium on Parallel and Distributed Processing, San Antonio, October 1995

The research reported in this docment was performed, in part, using facilities of the Laboratory for 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 the Office of Naval Research contract N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

On Memory Models and Cache Management for Shared-Memory Multiprocessors

Jack B. Dennis
MIT Laboratory for Computer Science
Cambridge, MA, 02139, USA
email: dennis@aj.lcs.mit.edu
phone: 617-253-6856
fax: 617-253-6652
and
Guang R. Gao
School of Computer Science, McGill University
Montreal, Canada
email: gao@andy.cs.mcgill.ca
phone: 514-398-4446
fax: 514-398-3883

Abstract

A popular approach to designing shared-memory computer systems is to specify a *memory model* upon which a variety of program execution models may be implemented. Alternatively, one may choose a desired *program execution model* (PXM) and specify a memory model suited to the PXM. We argue that this second approach is to be preferred because it avoids the trap of specifying features of the memory model (consistency, for example) that may not needed to implement a desired program execution model. If the PXM is a dataflow model (one based on or equivalent to recursive dataflow program graphs), then no cache consistency problem need arise if the memory model supports synchronizing memory operations. Then why use a memory consistency model as a basis for designing shared-memory multiprocessors? One argument is that a general memory model can support a variety of PXMs. However, many good PXMs, object-oiented programming, for example, may be built on top of a basic program model that does not require memory consistency—a dataflow model for example. Perhaps the principal justification for the consistent memory approach is the desire to build massively parallel processors using conventional RISC and superscalar processors—processors that do not offer efficient support for synchronizing memory operations. In contrast, we believe that memory systems supporting synchronizing memory operations will be cheaper to implement than memory consistency protocols, and will support a preferred class of program execution models. This paper is written in support of this alternative view.

1 Introduction

One approach to designing shared-memory computer systems is to specify a *memory model* upon which a variety of program execution models may be implemented. This approach has been used in the design of most shared-memory multiprocessors, which are usually based on a memory consistency model derived from Lamport's concept of *sequential consistency* [Lamport 79]. This requires the execution of a parallel program to appear as some interleaving of the memory operations on a sequential machine. It has been argued that sequential consistency is "the property of the memory system preferred by programmers".

Alternatively, one may choose a desired program execution model (PXM) and specify a memory model suited to the PXM. We argue that this second approach is to be preferred because it avoids the trap of specifying features of the memory model (consistency, for example) that may not needed to implement a desired program execution model.

If the PXM is a dataflow model (one based on or equivalent to recursive dataflow program graphs), then no cache consistency problem arises if the memory model supports synchronizing memory operations. A multiprocessor that implements such a program execution model can support all determinate computation, including, for example, all scientific computations expressible in the Sisal programming language [McGraw 85, Cann 92]. As we have shown elsewhere [Dennis/Gao 95], a general class of nondeterminate computations can be supported by adding a single special memory operation on non-cached locations, without introducing any additional consistency requirement to the memory model.

These circumstances raise the question: Why use a memory consistency model as a basis for designing shared-memory multiprocessors? The consistent memory approach is not the natural way to realize an efficient machine that implements a sound and effective program execution model. One argument is that a general memory model can support a variety of PXMs. However, many good PXMs, object-oiented programming, for example, may be built on top of a basic program model that does not require memory consistency—our dataflow model is an example. Perhaps the principal justification for the consistent memory approach is the desire to build massively parallel processors using conventional RISC and superscalar processors—processors that do not offer efficient support for synchronizing memory operations. On the contrary, we believe that memory systems supporting synchronizing memory operations will be cheaper to implement than memory consistency protocols and will support a preferred class of program execution models. This paper is written in support of this view.

We show how a general computing capability can be specified and implemented without the memory system satisfying any global coherence requirement. We begin by presenting a dataflow program execution model that uses *dataflow signal graphs* to represent function bodies, and uses *incremental arrays*, arrays with I-structure semantics, to represent data structures. Next, an abstract multiprocessor architecture is considered, and the interface of its memory system and the role of a memory model are discussed. A formal memory model is presented that supports the dataflow signal graph program execution model. Then we show how the abstract architecture may be implemented with cache memory to achieve good performance while satisfying the requirements of the model, and note that "coherence" of distributed memory is not an issue.

The closest realization of multiprocessor computer architecture to the implementation scheme for dataflow graphs suggested here is the Monsoon multiprocessor [Papadopoulos 90], but the principles may be applied to other multithreaded architectures derived from dataflow models.

2 Dataflow Signal Graphs

We wish to use a program execution model that has good generality so the concepts and principles discussed will be credible for a broad range of compter applications. We also wish to keep the model simple to avoid needless complexity. For generality, the model must encompass recursive nesting of function activations, and it must allow data structures of arbitrary extent to be created, accessed, and released.

We have chosen a dataflow model because it provides a simple semantic basis for parallel computing that encompasses the desired level of generality. The particular dataflow model is based on the graph/heap model [Dennis 74], but with function bodies specified using named variables and activation of actors by signals. We call this model *dataflow signal graphs*. This choice provides a

```
data Set =
   EMPTY |
   NONEMPTY int Set

Insert set n :: Set -> Int -> Set

Search set n  :: Set -> Int -> Bool

Insert set n =                      Search set n =
  case set of                         case set of
    EMPTY -> NONEMPTY n (EMPTY)         EMPTY -> FALSE
    NONEMPTY m rest ->                  NONEMPTY n rest -> TRUE
      if m > n then                     NONEMPTY m rest ->
        NONEMPTY n set                    Search rest n
      else
        NONEMPTY m (Insert n rest)
```

Figure 1: The `Insert` and `Search` functions written in Haskell.

straightforward mapping to the computer system and memory models that are the subject of this paper.

For writing textual program examples we will use the Haskell language [Hudak 91]. We use a functional language, not so much for the ease of programming it yields, but because functional programming languages express parallelism implicitly—without use of any explicit parallel processing commands. A compiler can readily translate functional programs into implicitly parallel dataflow code. Figure 1 shows Haskell text representing two recursive functions that, respectively, insert intergers into an ordered list, and search for an integer in the list. Figure 2 shows the `Insert` function represented as a *dataflow program signal graph*. Each actor (denoted by a box) specifies an *action* that occurs after all predecessor boxes (or the *start* node) have completed their actions. A directed arc (`u`,`v`) between two actors `u` and `v` in the graph denotes that when `u` finishes its action, a signal is to be sent to node `v` to signal its completion. Therefore we call these *signal arcs*.

An action is an operation that reads values of input variables, performs computation, and possibly assigns results to an output variable. Some boxes, drawn as oblongs, perform tests on input variables and signal successor boxes depending on outcomes of the tests. Application of a function causes: new instances of the function's variables to be created and initialized with the undefined value; argument values to be made available to the new activation; and the *start* node of the called function graph to be enabled. When the application terminates, the result value is assigned to a specified variable of the caller. Each variable must be assigned a unique and unambiguous value (or not be assigned or referenced) in any instantiation of the program graph. So that this is true, all data dependences between variables in a program graph must be represented by the partial order defined by the signal arcs of the graph.

Incremental arrays are implemented using I-structure operations [Arvind 89]. The `CREATE n` operation creates (as a heap node) an array of n elements indexed by integers $0, ..., n-1$ in which each element has the value `UNDEF`, meaning undefined. The `CREATE` operation yields a *pointer* to the heap node that may be held by a variable, stored as an array element, and used to store and access elements of the array. An action `A[i]:v` defines element `i` of array `A` to have the value `v`. Any reference `A[i]` to an array element completes (yielding the element value) only when the element has become defined.

Haskell programs and the dataflow signal graphs derived from them, as we have described them here, are *determinate*—the result of any function evaluation is independent of the order in which

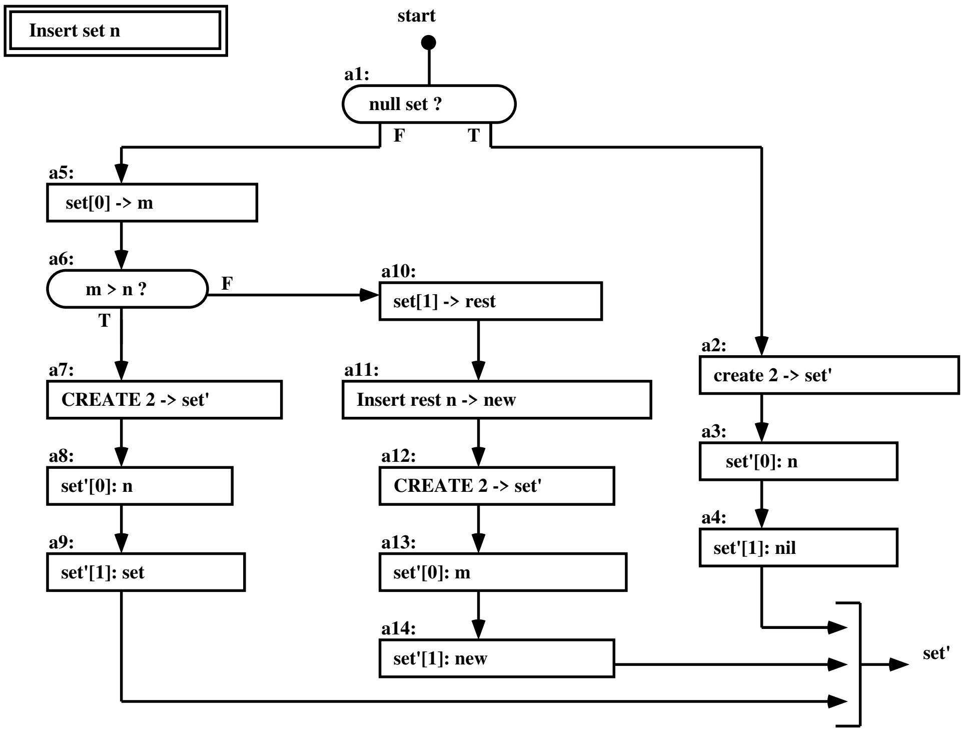

Figure 2: Dataflow program graph for the `Insert` function.

enabled nodes of the graphs are chosen for execution. The extension of the dataflow model to include nondeterminate computations is discussed in [Dennis/Gao 95], where it is shown how typical programs may be implemented using a single special memory operation, `SWAP`, which operates on non-cached memory locations, so that their correct execution does not depend on a memory consistency model.

3 The Memory Model

A shared-memory multiprocessor consists of a collection of processors and a distributed memory system organized so that each processor may access any object held in the memory. We view this general architecture according to the *abstract computer system* model shown in Figure 3. In this model, the processors interact with the memory system by presenting *commands*, and the memory responds by storing information or returning *responses* containing values of stored data. The Interprocessor Network supports function application as discussed below. In this section we describe an architecture-independent memory model that is a specification of such a memory system suited to supporting the chosen dataflow program execution model.

3.1 The Processors

Before discussing characteristics of the memory system, let us consider the nature of the processors. For simplicity we assume that any one function activation is performed entirely on one processor; that is, the instruction executions corresponding to one instantiation of a dataflow signal graph are

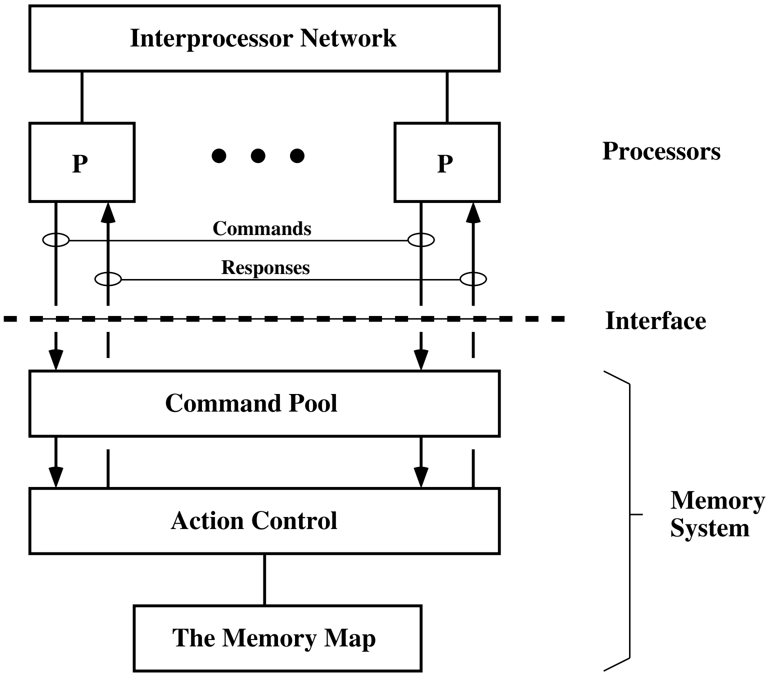

Figure 3: Abstract computer system with memory system model.

all done by the same processor. This choice has been made in most, if not all, of the experimental multiprocessors based on dataflow principles.

Given this choice, it makes sense to allocate memory for the data frames of function activations in the processors executing them. Then all references to variables held in data frames will be local memory references, and may be performed independently of the distributed memory system.

Function application is initiated by an `APPLY` instruction that starts the following sequence of events:

1. A processor is chosen for executing the new function activation.
2. A memory segment is allocated at the chosen processor for the data frame of the new activation, and each cell of the data frame is set to `UNDEF`[1].
3. A *return continuation* consisting of the data frame address and the list of successors of the `APPLY` instruction is stored in the data frame of the new activation.
4. The function argument value is placed in the argument location of the data frame, and the start node is enabled for execution.
5. When a return node of the function is executed, the result value of the function is delivered according to the successor list.

[1]We do not discuss here the method of allocating data frames. If the data frame is always allocated in private memory of the processor chosen to execute the activation, the process need not involve the (global) memory system. However, this choice could lead to failure once local memory is full, even though memory may be available at other processors. In the presence of concurrency, allocation actions are typically examples of nondeterminate actions, and are treated elsewhere.

If a function has more than one argument or result value, the argument or result values are made components of a (heterogeneous) incremental array. This choice provides lenient evaluation semantics. We assume that function initiation and termination are performed using messages sent over the Interprocessor Network.

3.2 The Memory System

For our program execution model, the function of the memory system is to support the operations of defining and accessing elements of incremental arrays. We insist that correct operation of the system as a whole not depend on immediate responses to processors from the memory system. To enforce this requirement, the abstract memory system model (Figure 3) includes a buffer (the Command Pool) for commands that have been presented by processors, but have not yet been acted on by the memory system. The memory system acts upon commands chosen arbitrarily (subject to fairness) from the Command Pool. Thus any criterion of correct behavior must recognize that commands may be arbitrarily delayed[2]. (This is essentially the network delay assumption for scalable multiprocessors.)

The basic memory operations are `READ` and `WRITE`. For the purpose of supporting execution of dataflow graphs, one may view the memory system as a collection of locations where values may be placed by one computing entity for use by others that act concurrently. The `WRITE` command is used to place values in memory locations and the `READ` command is used to retrieve them. Now, even if one could guarantee that each `READ` is presented to the memory system only after presentation of the corresponding `WRITE`, it would be impossible to ensure that the memory system acts on the `WRITE` before acting on the `READ`. This is because either command may remain in the Command Pool arbitrarily long before being selected for action. For this reason we specify the memory system so that a pair of `READ` and `WRITE` operations on the same memory location has the same effect regardless of which operation is acted upon first. This requires that they be "synchronizing" shared-memory operations.

Here we give a specification in the Haskell language of the Action Control module of the memory system. The command messages sent to the memory have the following formats:

```
data Request =
   READ Address Node Continuation  |
   WRITE Value Address
```

In these formats the data type `Address` is the set of (global) memory locations, `Value` is the set of possible contents of memory locations, `Node` is the set of processor identifiers, and `Continuation` is information that identifies a specific activity to be continued by the requesting processor upon completion of the command. For our program model a continuation would consist of the address of a data frame and the offset of the instruction to be activated within the program segment of the function. (The address of the program segment could be included in the continuation, but it is usually preferable to retrieve it from the data frame.)

Messages sent by the memory system to processors in response to `READ` commands have the format:

```
data Message =
   (MESSAGE Node Response)
data Response =
   REPLY Value Continuation
```

The state of the memory system consists of a pool of messages that have been presented by processors to the memory system, but have not been acted upon, and a mapping from addresses to memory items:

[2] Of course, timing behavior must be analyzed and bounded where real-time performance guarantees are required.

```
data MemoryState :: ([Request], MemoryMap)

data MemoryMap :: Int -> MemoryItem
```

Items are of the following kinds:

```
data MemoryItem =
   (UNDEF)          |
   (DEFINED Value)  |
   (QUEUE [Entry])

data Entry = (ENTRY Node Continuation)
```

The memory system acts on request messages according to the rules given in Figure 4[3]. Note that `map` denotes a function that maps locations into items, so `map location` evaluates to the memory item of interest. The function `MapUpdate` yields a new mapping in which the specified location contains the specified item.

A `READ` request contains the address of the location to be read, the processor identifier, and the continuation that specifies the graph node (or nodes) to which the value read should be delivered. If the state of the location is `DEFINED`, the value is sent in a response to the processor. If the state is undefined (`UNDEF`), the `READ` request must be the first read access to the location. In this case it creates a queue with the `node` and `continuation` in its first entry. If the state of the location contains a queue, it just adds a new entry to the queue. A `WRITE` request contains a value and the address of the location where the value is to be written. If the state of the location is undefined (`UNDEF`), it writes the value into the location and changes the state to defined (`DEFINED`). If the state is a queue, it writes the value into the location and sends a response message for each entry in the defer queue. If the state is found to be defined (`DEFINED`), it signals an error.

3.3 Allocation and Reuse of Memory Cells

In the program execution model we have chosen, each array element has the value `UNDEF` when the array shell is created. Thereafter each element may become defined exactly once, by a `WRITE` operation, and then read any number of times. In principle, the definition persists forever. An implementation of the model must ensure that memory cells used for array elements are not reused before the last reference is made to the former value. Normally, this is done by run-time garbage collection and storage allocation routines.

Since memory management functions are often relatively expensive, one may wish to add mechanisms to permit reuse of memory cells without reallocation. A more sophisticated abstract computer system could use memory cells with more than three states so that a second and further write commands are delayed until all read operations intended for previous values have been performed. On the other hand, it may be possible to transform programs so that exactly one read is performed for each write, leading to a simpler implementation.

4 Implementation and Caching

Here we discuss an implementation of the distributed memory system that includes a caching capability. The goal of the design is to ensure that a large fraction of memory references are satisfied in the cache memory associated with the processor making the reference. There are two gains from caching:

1. Less demand for high bandwidth between processors and main memory

[3] In Haskell the prime character (') is valid in identifiers and primed identifiers are often used to denote the modified value of an object.

```
MemoryAction :: Request -> MemoryMap ->
                        ([Message], MemoryMap)

MemoryAction (READ address processor continuation) map =
  let item = map address
  in case item of
      (UNDEF)  -> let
          q = (ENTRY node continuation (EMPTY))
          map' = MapUpdate map address (QUEUE q)
        in ([], map')
      (QUEUE q) -> let
          q' = AddToQueue node continuation q
          map' = MapUpdate map address (QUEUE q')
        in ([], map')
      (DEFINED value) ->
            (SendReply value node continuation, map)

MemoryAction (WRITE value address) map =
  let item = map address
  in case item of
      (UNDEF) -> let
          map' = MapUpdate map address (DEFINED value)
        in ([], map')
      (QUEUE queue) ->let
          msgs = MakeResponses queue node value
          map' = MapUpdate map address (DEFINED value)
        in (msgs, map')
      (DEFINED value) -> ERROR

MakeResponses queue node value =
  case queue of
    [] -> []
    [(ENTRY node continuation):queue'] -> let
        content = (RESPONSE value continuation)
        msg = (MESSAGE node content)
      in [msg:(MakeResponses queue' node value)
```

Figure 4: The state transition rules of the Abstract Memory System.

2. Lower average latency of memory accesses seen by processors

With caching, many read requests will be answered quickly from the local cache of a processor. However, multithreaded architecture of the processor is needed to avoid wasting processor resources and risking deadlock on attempts to read from undefined locations. Here we describe the organization of the memory system, discuss options for implementing deferred read requests, and give a detailed protocol for a scheme that appears attractive.

As shown in Figure 5, the memory system is built with a Memory Management Unit (MMU) and fast Cache Memory (C) associated with each processor. The Memory Network interconnects the MMUs with Memory Units (MU) that implement memory locations in disjoint portions of the global address space. We assume that the network is reliable, but that messages are subject to arbitrary but finite delays, and the network does not guarantee preservation of message order.

An important implementation question is where and how a record is kept of pending read requests that must be answered when a location becomes defined. Possible approaches to storing this *defer*

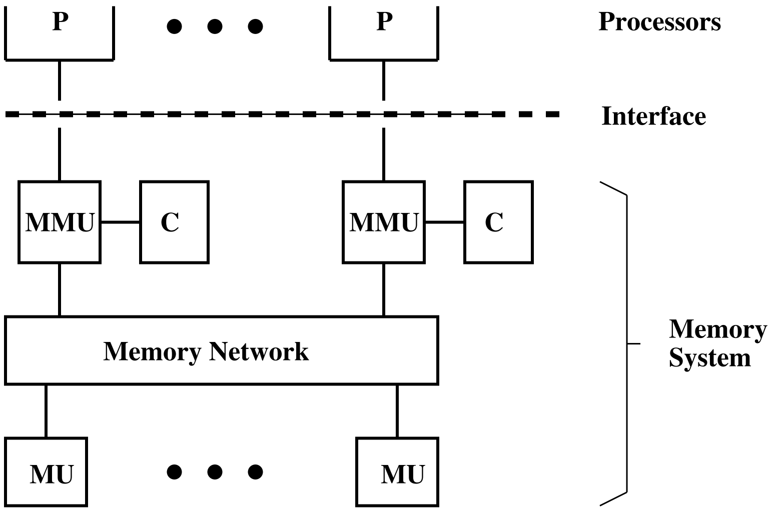

Figure 5: An implementation of the memory system with caching.

queue include:

1. Store all continuations for pending reads of a location in a continuation queue at the corresponding Memory Unit.
2. Store the continuation for each pending read only at the processor originating the request. When the location becomes defined, the home processor of the location broadcasts the value to all processors.
3. Store at the memory unit a list (the *waiting set*) of identifiers of those processor nodes that have one or more pending reads for a memory location. Each of these processors holds the set of continuations for read requests it originated.
4. The MMU associated with the requesting processor tries repeatedly to access the location (at gradually less frequent intervals) until the location becomes defined.

We have chosen (3) as the most promising approach, and outline the corresponding protocol in detail below. Each location has a *home node*, the processor/memory module that has the location in its portion of the main memory address space.

4.1 Assumptions

The specification presented below contains several assumptions for the sake of simplicity. One is that the cache is fully associative. A benefit is that a new cache line may be stored in any free slot of the cache memory. For simplicity, we assume that no cache line that contains a defer queue is ever replaced until after the entry becomes defined; that is, we assume there is more than enough cache memory to hold placeholders for all pending read requests. If a set-associative or direct mapped cache organization were used, a more sophisticated mechanism would be needed to avoid blocked reads due to occupancy of all possible slots by defer queue entries. A feature to help avoid potential deadlock would be a buffer for unexecutable read commands at the processor interface with each MMU. Since write commands are never blocked by program delays, they should be allowed to bypass read commands held in the buffer. Discussion of these alternatives is beyond the scope of the present paper.

Another assumption is that all memory units are in the same relationship to each and every processor—we are not modeling a NUMA architecture. In the case of NUMA architecture it should be possible to respond somewhat quicker to a remote read request if the location is represented in the cache of the home processor, but we do not discuss these details here.

4.2 Overview of Memory System Operation

A `READ` command causes the local cache entry to be examined. If it is `DEFINED`, the value is returned. If it is a defer queue, the new continuation is appended. Otherwise, a new cache entry must be created. This is possibile only if there is a cache slot with state `EMPTY` or `DEFINED`. If all cache slots are filled with `QUEUE` entries, processing of the `READ` command must be postponed. When a slot containing a `DEFINED` entry is chosen for replacement, the existing entry may simply be deleted because there will be a duplicate at its home memory unit. When a new cache entry is created for a `READ` request, a read message is also forwarded to the home memory unit for the specified address. If the location is `DEFINED` at the home MU, a reply message is generated. Otherwise the identifier of the requesting processor is added to the waiting set of the location,

In the case of a `WRITE` command, the local cache line is examined and the queue of continuations is replaced by the value. If there is no matching cache line, one is created if a slot can be made free. In any case, a write message is sent to the home memory unit. The value is also passed to activate any continuations held by the local MMU. At the home memory unit, the entry for the location of interest is changed to `DEFINED` and each processor in the set of waiting processors for the addressed location is sent a reply message notifying it of the written value.

4.3 Formal Model

In this section we use the Haskell language to give a specification of the implementation of our memory model. We use Haskell lists to represent sets of various objects. In doing this we, of course, do not wish to imply that a hardware implementation would utilize list operations!

The specification consists of functions that specify the actions performed by the Memory Managemnent Units (MMUs) and the Memory Units (MUs). The states of an MMU and of an MU are modelled by elements of the data types `Cache` and `MemMap`, respectively. For an MMU the specification is a pair of functions: one that specifies its action in reponse to commands presented by the associated processor. The `ProcessCommand` function acts on `READ` and `WRITE` commands to produce an updated state of the local cache, a set (possibly empty) containing at most one message directed to the home MU of the addressed location, and a set of responses to completed `READ` commands.

```
ProcessCommand :: Request -> Cache ->
        ([Response], [Message], Cache)
```

The `ProcessMessage` function specifies the MMU's action in response to `REPLY` messages from the MUs.

```
ProcessMessage :: Message -> Cache ->
        ([Response], [Message], Cache)
```

The MUs are specified by a function that describes the action performed in response to messages received from the MMUs.

```
ExecuteCommand :: Message -> MemMap ->
        ([Message], MemMap)
```

These functions may be viewed as an operational semantics for the memory system as follows: The overall state of the memory system consists of the states of all MMUs (elements of `Cache`), the states of the MUs (elements of `MemMap`), the set of messages in transit through the memory network, and, for each processor, the set of commands presented but not yet acted upon, and the set of responses generated by the MMUs but not yet acted upon by the processors. The overall state is advanced by selecting arbitrarily a message or command and using the appropriate semantic function to determine the new overall system state.

The states of the MMUs and MUs are modelled as follows:

State of the Memory Units: This is simply the memory map function that maps locations to entries.

```
data MemMap = Location -> Entry

data Entry = (UNDEF) | (DEFINED Value) | (QUEUE Set)

data Set = [Node]
```

An element of the data type `Set` is a set containing identifiers of the processors in the waiting set for the location. This set would most likely be implemented as a fixed-length bit vector, as in simple directory protocols of cache coherence schemes.

State of the Memory Management Units: Each cache entry contains an address key and the contents of the line that includes a state tag and either a value or the defer queue.

```
data CacheState = [CacheLine]

data CacheLine =
  (EMPTY) |
  (DEFINED Address Value) |
  (QUEUE Address Defer)

data Defer = [Continuation]
```

As memtioned earlier, the `Continuation` type contains a data frame pointer and an instruction address. The defer queue would probably be implemented as a list structure. A single-element queue would be held in the cache slot; otherwise the slot would contain a pointer to an overflow area in separate memory within the MMU.

The formats of commands and responses, and of the messages passed in the Memory Network are as follows:

Commands and Reponses: The two command types presented to the MMUs by processors, and one type of response have been specified in Section 3.

Messages held in the Memory Network: There are two command types sent from the MMUs to the MUs and a single reply message type.

```
data Message = (MESSAGE Node Content)
data Content =
  (READ Location Node)   |
  (WRITE Value Location) |
  (REPLY Address Value)
```

In these messages `Location` is the set of local addresses for any MU, and `Node` is the set of processor identifiers.

4.4 The Memory Management Unit

The actions performed by the MMUs in response to `READ` and `WRITE` commands are specified by the `ProcessCommand` function given in Figures 6 and 7. An MMU's processing of reply messages from MUs is specified by the `ProcessMessage` function given in Figure 8. In these routines, special functions are used to invoke operations on the cache memory

- `CacheHit cache address`: Return `True` if the cache contains an entry for address `address`.
- `CacheAvailable cache`: Return `True` if the cache contains any entry in state `UNDEFINED` or `DEFINED`.
- `CacheFetch cache address`: The line held by the cache for address `address` is returned. If no entry exists, an error occurs.
- `CacheUpdate cache address line`: If the cache contains a line for `address`, it is replaced with the given line. Otherwise an error occurs.
- `CacheReplace cache address line`: If the cache contains a line for `address`, it is replaced with the given line. Otherwise, the line replaces any one line that is `UNDEFINED`. If there is no such cache line, a line that is `DEFINED` is replaced. If none exist, an error occurs.

Three functions are used to disect global addresses into node identifiers and locations, and to construct global addresses from their components:

- `AddressNode :: Address -> Node`
- `AddressLocation :: Address -> Location`
- `MakeAddress :: Node -> Location -> Address`

4.5 The Memory Unit

The Memory Unit reponds to write commands by storing the given value and sending `REPLY` messages to each node, if any, in the waiting set. For read commands, a `REPLY` message is sent if the location is `DEFINED`; otherwise, the number of the originating node is added to the waiting set. These actions are specified by the function `ExecuteCommand` defined in Figures 9 and 10. The `MakeReplies` function returns a set (list) of reply messages, one for each node in the waiting processor set.

5 Cache-Coherent Multiprocessors

Shared-memory multiprocessor architectures based on memory coherence models start from an assumption that the *sequential consistency model* of Lamport is the ideal view of memory for the processors of a parallel computer. It provides programmers with a memory model familiar to them from working with single-processor computers, and it is hoped that (multi-process) software written for single processor systems may be migrated to the parallel processor without significant modification. This philosophy of multiprocessor architecture is represented by two main styles: the cache-coherent nonuniform memory access architecture (CC-NUMA), and the cache-only memory architecture (COMA) Examples of CC-NUMA machines are the Stanford DASH architecture [Lenoski 90, Lenoski 92] and the MIT Alwife Machine [Agarwal 90], while examples of COMA machines include the KSR-1 [KSR 1992] and the DDM [Hagersten 92].

In these systems, cache coherence is achieved by maintaining a directory at each processor. The directory contains, for each block of memory in its portion of the global address space, a record of

```
ProcessCommand (READ address continuation) cache =

  if not (CacheHit cache address) then              --- cache miss

    if (CacheAvailable cache) then let
        list = [continuation]                       --- make new
        line' = (QUEUE list')                       --- cache
        cache' = CacheReplace cache address line'   --- entry

        node = AddressNode address                  --- forward
        location = AddressLocation address          --- read
        content = (READ location MyNode)            --- request
        msg = (MESSAGE node content)                --- to the MU
      in ([],[msg], cache')

    else Error "no space for read request"

  else let                                          --- cache hit

      line = CacheFetch cache address
    in case line of
      (DEFINED value) -> let
          response = (RESPONSE value continuation)  --- return
        in ([response],[],cache)                    --- value

      (QUEUE list) -> let
          list' = [continuation:list]               --- append
          line' = (QUEUE list')                     --- continuation
          cache' = CacheUpdate cache address line'  --- to the queue
        in ([],[],cache')
```

Figure 6: Actions by an MMU for a `READ` command.

which processors hold cached copies of the block. When a processor writes a location in a block, messages are sent to all processors in the directory record to invalidate cached copies of the block. An attempt to read an invalidated block causes the block to be fetched from its home node. Some cached blocks will be invalidated even though there may be no future attempt to read them. Also, when an invalidated block is read, there will be two additional messages to request and retrieve the updated value.

In the execution model of this paper, a write operation causes the written value to be delivered to all processors containing activities that have pending reads. This protocol uses fewer messages (two instead of three) to implement the transaction, and all messages are productive (each request and its reply concern a word actually used in the computation). Thus communication from a producer of data to one or more consumers is more efficient.

Let us illustrate this by a simple example. Let `X` be a shared location used to transmit data values between a producer and a consumer. To implement this interaction, a semaphore or lock, say `S`, must be used to ensure the correct synchronization. The producer code may look like

```
  ...
X := ...
unlock(S)
  ...
```

```
ProcessCommand (WRITE address value) cache =

  if not (CacheHit cache address) then              --- cache miss

    if (CacheAvailable address) then let
        line' = (DEFINED value)                      --- make new
        cache' = CacheReplace cache address line'    --- entry

        node = AddressNode address                   --- forward
        location = AddressLocation address           --- write
        content = (WRITE location MyNode value)      --- request
        msg = (MESSAGE node content)                 --- to the MU
      in ([],[msg], cache')

    else let
        node = AddressNode address                   --- forward
        location = AddressLocation address           --- write
        content = (WRITE location MyNode value)      --- request
        msg = (MESSAGE node content)                 --- to the MU
      in ([],[msg], cache')

  else let                                           --- cache hit

      line = CacheFetch cache address
    in case line of
      (DEFINED value) -> Error "multiple write"

      (QUEUE queue) -> let
          responses = MakeResponses queue value      --- respond to
          line' = (DEFINED value)                    --- deferred
          cache' = CacheUpdate cache address line'   --- reads
        in (responses, [], cache')
```

Figure 7: Actions by an MMU for a `WRITE` command.

and the consumer code may look like:

```
  ...
lock(S)
.. := ..X..
  ...
```

The lock and unlock operations can be viewed as playing the role of semaphore operations such as the P and V operations of Dijkstra. Since performing a semaphore operation requires at least one (global) memory reference, this mechanism is less efficient than when the synchronization is combined with data transfer using I-struture operations. Moreover, implementation of the lock operation calls for either busy-waiting or calls to operating system scheduling routines if a long wait is anticipated. Furthermore, given the network delays in a shared-memory multiprocessor that utilizes a multistage network, the implementation of locks is likely to be more expensive than in a simple shared-bus architecture. Note that if the above producer and consumer code is in an innermost loop, the cost will be proportional to the loop bound!

For the write to the shared variable `X` itself, different CC-NUMA cache-coherence protocols may incur different overhead. Under a memory model based on (strong) sequential consistency, the

```
ProcessMessage (REPLY address value) cache =
  if (CacheHit cache address) then let
      line = CacheFetch cache address
    in case line of
      (DEFINED _) -> Error "address already defined"

      (QUEUE list) -> let
          responses = MakeResponses list value       --- update
          line' = (DEFINED value)                    --- cache and
          cache' = CacheUpdate cache address line'   --- respond
        in (responses, [], cache')

  else Error "no pending reads"
```

Figure 8: Actions by an MMU for a `REPLY` message.

```
ExecuteCommand (READ location node) map = let
    entry = map location
  in case entry of
    (UNDEF) -> let
        set' = [node]
        entry' = (QUEUE set')
        map' = MapUpdate map location entry'
      in ([], map')

    (DEFINED value) -> let
        address = MakeAddress MyNode location
        content = (REPLY address value)
        msg = (MESSAGE node content)
      in ([msg], map)

    (QUEUE set) -> let
        set' = SetUnion set [node]
        entry' = (QUEUE set')
        map' = MapUpdate map location entry'
      in ([], map')
```

Figure 9: Actions by an MU for a `READ` message.

write to `X` in the producer will need to send invalidate signals to the consumers, and wait for their acknowledgement before it proceeds. Under a weaker memory model such as the release consistency model, some optimization may be introduced to relax the order constraints. For example, the write operation may proceed before receiving the acknowledgement signals until the unlock operation (called *release*) is performed. However, the overhead of the semaphore operations must still be paid.

An advantage of our program execution model comes from combining data transfer and synchronization, avoiding altogether any separate cost of synchronization operations. In addition, our cache management scheme involves no non-productive cache-coherence (invalidation) traffic. Of course, the synchronization overhead may be reduced (per word) by grouping data into blocks and synchronizing on each block. The result is medium or coarse-grain parallelism, and suffers from the increased delay of the start of processing by the consumer.

```
ExecuteCommand (WRITE location value) map = let
    entry = map location
  in case entry of
    (UNDEF) -> let
        entry' = (DEFINED value)
        map' = MemUpdate map location value
      in ([], map')

    (DEFINED value) -> Error "multiple write"

    (QUEUE set) -> let
        address = MakeAddress MyNode location
        msgs = MakeReplies set address value
        entry' = (DEFINED value)
        map' = MemUpdate map location entry'
      in (msgs, map')

MakeReplies :: Set -> Address -> Value -> [Message]

MakeReplies set address value =
  case set of
    [] -> []
    [node:set'] -> let
        content = (REPLY address value)
        msg = (MESSAGE node content)
      in [msg:(MakeReplies set' address value)
```

Figure 10: Actions by an MU for a `WRITE` message.

Another issue deserving attention concerns the problem of switching between jobs on a shared-memory computer. The fact that continuations contain data frame pointers and are sequestered in defer queues that may have a long lifetime will make it difficult to switch between jobs economically unless the data frame pointers are in a universal address space and can be guaranteed to be unique. In a conventional multiprocessor, some version of "gang scheduling" may be used in which all network traffic pertaining to inactive jobs is intercepted and either reinterpreted or held in software queues until the corresponding job is reactivated. This can be effective because only a relatively short time is needed for network traffic related to one job to die out. With pointers stored in defer queues that may produce network messages at unknown times in the future, a universal addressing mechanism will be essential.

6 Multithreaded Architecture Projects

In this section, we breiefly discuss two related multithreaded architecture projects. These are the Monsoon project, which implements a pogram execution model closely related to the one we have presented; and the Tera computer and its forerunners, which also address multiprocessor latency and synchronization issues.

6.1 The Monsoon Project

Several projects have designed and constructed multiprocessors to execute nested function programs using dataflow principles. Of these, the design that comes closest to fulfilling the characteristics we have outlined is the Monsoon dataflow multiprocessor [Papadopoulos 90]. The program execution model adopted in these machines derives from the original *unravelling interpreter* [Arvind 78], which has much in common with the graph/heap model of [Dennis 74]. In Monsoon there are two forms of data memory, the frame store and I-structure storage. These correspond, respectively, to the data frames and heap storage of our implementation. Monsoon uses dataflow graphs as an intermediate program form for compiling. Each function application in Monsoon is executed entirely by one processor, and all local variables are held in a stack frame allocated in the local frame store.

Monsoon supports *I-structures*. Incremental arrays are held in separate memory units that act on messages sent by processors to request read and write operations. Attempts to read an undefined location in I-structure memory cause the continuation of the processor activity to be entered in a defer queue held in the I-structure memory. If the queue has only one entry, it is stored in place of the expected value; if more entries are needed, they are listed in an overflow area. For more on I-structures, see [Arvind 89]. A write operation makes an entry in I-structiure memory defined, and causes all waiting activities to be resumed with the element value. Experiments with Monsoon have shown that defer queues contain only a few entries during program execution, most often no more than one [Hicks 93].

There is no cache memory in Monsoon. Because the frame store is local and is built of fast static RAM, there would be little benefit in caching local variables. The implementation scheme we have presented could be applied to the Monsoon I-structure memory. It shows how caching of I-structure words may be done so as to increase efficiency by reducing the number of network transactions and decreasing the average latency of I-structure accesses.

A successor project to Monsoon, the Start-NG project [Ang 94], plans to implement a closely related program execution model using a conventional superscalar microprocessor, the PowerPC chip.

6.2 The HEP, Horizon and Tera

The HEP, Horizon and Tera multiprocessor architectures have considerable conceptual commonality with the designs considered here. The processors are multithreaded and can tolerate latency of memory accesses through the memory network. Each processor has a large register set built of fast static memory. The Tera commercial supercomputer [Alverson 90] is designed upon the ideas and experiences from the Denelcor HEP multiprocessor [Smith 81] and the Horizon project at the Supercomputing Research Center [Kuehn 88]. The Tera architecture includes four tag bits in each memory word to support thread synchronization. These tags could be used to support I-structure operations for incremental arrays, but software support would be required to manage the queue of deferred read operations and to schedule activities. To avoid excess network traffic from repeated attempts to complete a memory access blocked until a tag is set by another process, an automatic retry mechanism is provided.

The Tera architecture does not include cache memory. Instead, it depends on its large register set to reduce required main memory bandwidth, and multithreading to ameliorate the latency of main memory access. Implementing caching as we have described above could improve the efficiency of I-structure operations significantly.

The program execution model for these machines is a set of independent processes running in a shared address space, and the architecture does not provide any specific support for function application as in Monsoon. Hence it appears that support of modular nests of functions will be left to the run-time software and compilers for high-level languages.

7 Conclusions

We have presented our view that the design of shared-memory multiprocessors should be guided by a sound program execution model. In particular, we have outlined a program execution model derived from dataflow and functional programming principles that applies to determinate computations. We have also described an implementation of the model as a multiprocessor architecture containing a distributed memory system supporting synchronizing read and write operations. We specified the memory using a state-transition system and showed how it may be implemented to incorporate caching of remote words without depending on any memory consistency property.

Our program execution model and its implementation lie in contrast to shared-memory multiprocessor design based on memory consistency models, in which the memory interface is designed to present desirable characteristics to programmers. We believe that memory systems supporting synchronizing memory operations will be cheaper to implement than memory consistency protocols, and will support a preferred class of program execution models.

Acknowledgement

The authors would like to thank the MIT Laboratory of Computer Science and the School of Computer Science of McGill University for facilitating this effort. The second author would like to thank the National Science and Engineering Research Council (NSERC) of Canada for their support.

References

[Agarwal 90] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114. ACM and IEEE, 1990.

[Alverson 90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, 1990.

[Ang 94] Boon Seong Ang, Arvind, and Derek Chiou. StarT the next generation: Integrating global caches and dataflow architecture. Computation Structures Group Memo 354, M.I.T. Laboratory for Computer Science, February 1994.

[Arvind 78] Arvind, Kim P. Gostelow, and Wil Plouffe. *An Asynchronous Programming Language and Computing Machine.* Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[Arvind 89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[Cann 92] David Cann. Retire Fortran: A debate rekindled. *CACM*, 35(8), August 1992.

[Dennis 74] Jack B. Dennis. First version of a data-flow procedure language. In *Proceedings, Colloque sur la Programmation*, no. 19 in *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1974.

[Dennis/Gao 1994] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Archtecture*. Kluwer, 1994.

[Dennis/Gao 95] Jack B. Dennis and Guang R. Gao. Multiprocessor Implementation of Nondeterminate Computations in a Functional Programming Framework. Computation Structures Group Memo 372, M.I.T. Laboratory for Computer Science, March 1995.

[Hagersten 92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM—A cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[Hicks 93] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993.

[Hudak 91] Paul Hudak (editor), Simon Peyton Jones (editor), Philip Wadler (editor), Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict purely functional language (version 1.1). Yale University, Department of Computer Science, August 1991.

[Kuehn 88] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing '88*, pages 28–34. IEEE and ACM, November 1988.

[Kuskin 94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313. IEEE and ACM, April 1994.

[KSR 1992] Kendall Square Research. KSR-1 Technical Summary. Waltham, MA, 1992.

[Lamport 79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, September 1979.

[Lenoski 90] Daniel Lenoski, Kkourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The DASH approach. In *COMPCON Spring '90: Digest of Papers*, pages 62–81. IEEE Computer Society, March 1990.

[Lenoski 92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103. IEEE and ACM, May 1992.

[McGraw 85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. Technical Report M-146, Rev. 1, Lawrence Livermore National Laboratory, 1985.

[Papadopoulos 90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the Seventeenth Annual International Symposium of Computer Architecture*, pages 82–91. IEEE and ACM, 1990.

[Smith 81] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE: Real-Time Signal Processing IV*, volume 298, pages 241–248, 1981.