Optimizing the UPC Communication Run-Time Library

by

Igor F. Cherpak

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 30, 2000
[June 2000]

Author .....
Department of Electrical Engineering and Computer Science
May 30, 2000

Certified by .....
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Optimizing the UPC Communication Run-Time Library
by
Igor F. Cherpak

Submitted to the
Department of Electrical Engineering and Computer Science

May 30, 2000

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This paper describes the design of a basic communication run-time library for the UPC parallel language and DEC's Memory Channel hardware. Also described is an implementation of Cachet, an adaptive cache coherence protocol for distributed systems, which was added to the basic communication layer to even further boost performance. The implementation of two Cachet micro-protocols: Cachet-Base and Cachet-WriterPush are described. The Cachet cache coherence scheme was implemented entirely in software on Alpha workstations. The results of benchmarks running on the basic communication layer with and without Cachet are presented. These experiments show that the communication optimization can provide a performance improvement of up to an order of magnitude over the unoptimized benchmarks.

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Contents

Chapter 1

Introduction

1.1 Motivation

Application performance depends on the compiler, on the communication run-time system and on the efficiency of their interaction. By fine-tuning these three components, a higher performance can be achieved. This thesis will concentrate on the UPC run-time library and describe one way of optimizing it with an adaptive distributive cache coherence protocol, Cachet.

In the first part of the paper, an implementation of the basic communication library will be presented. The second part, will describe the integration of Cachet with the communication layer.

The performance of the basic communication library and the library enhanced with Cachet will be evaluated on a benchmark suite. This paper also presents the relative performance results of UPC optimized with Base, WriterPush, and with the integrated Cachet protocol, which includes both Base and WriterPush micro-protocols.

1.2 Contributions of this Thesis

The purpose of the project is to build a real-world implementation of an efficient communication run-time library for the UPC language.

This thesis also functions as a case study of implementing Cachet software cache coherency protocol for the purpose of optimizing the communication library.

1.3 Overview

Chapter 2 describes the platform for which the communication system is written. Namely, the UPC compiler and the underlying communication hardware.

Chapter 3 describes the implementation of the basic communication run-time library for the UPC language.

Chapter 4 presents the implementation of Cachet.

Finally, Chapter 5 summarizes the results of the project and makes suggestions for future research.

Chapter 2

UPC Compiler and Underlying Hardware

2.1 UPC Compiler

UPC, a language currently being developed by Compaq, extends the C language and supports a simple model of shared memory for parallel programming. In this model the data can be either shared or distributed among the communicating processors. UPC is designed to be much simpler than MPI or any other existing parallel language. Therefore, it facilitates programming and maintains high performance execution at the same time.

2.2 Hardware

The target platform for UPC and the run-time library are standalone single-processor Alpha machines running STEEL version 4.0 of the Tru64 UNIX Operating System. These machines are connected by the Memory Channel interconnect.

Digital Equipment's Memory Channel (MC) is a low-latency remote-write network that provides applications with access to memory on a remote cluster using memory-mapped regions. Only writes to remote memory are possible -- reads are not supported on the current hardware. The smallest granularity for a write is 32 bits (32 bits is the smallest grain at which the current-generation Alpha can read or write atomically). The adapter for the MC network is connected to the PCI bus. A memory-mapped region can be mapped into a process' address space for transmit, receive, or both (a particular virtual address mapping can only be for transmit or receive). Virtual addresses for transmit regions map into physical addresses located in I/O space, and, in particular, on the MC's PCI adapter. Virtual addresses for receive regions map into physical RAM. Writes into transmit regions bypass all caches (although they are buffered in the Alpha's write buffer), and are collected by the source MC adapter, forwarded to destination MC adapters through a hub, and trans-

ferred via DMA to receive regions (physical memory) with the same global identifier. Regions within a node can be shared across processors and processes. Writes to transmit regions originating on a given node will be sent to receive regions on that same node only if *loop-back* through the hub has been enabled for the region. Loop-back is used only for synchronization primitives.

MC has page-level connection granularity, which is 8 Kbytes for Alpha cluster. The current hardware supports 64K connections for a total of a 128 Mbyte MC address space. Memory Channel guarantees write ordering and local cache coherence. Two writes issued to the same transmit region (even on different nodes) will appear in the same order in every receive region.

Chapter 3

Basic Communication Run-Time Library

3.1 High Level Overview

The main operations of the communication library are remote Get and Put operations. There are many different schemes for implementing these remote communication routines. The most typical approach is to use the message-passing mechanisms provided by the UNIX operating systems. However, many research papers have shown that such a communication mechanism performs poorly because of multiple message coping. Numerous alternative schemes have been suggested to improve the communication [1,2]. Another alternative to the UNIX message-passing mechanism is presented below. This message-passing mechanism fully utilizes the underlying hardware, Memory Channel, and as a result achieves more efficient communication.

Memory Channel provides shared global memory. For each processor, a request buffer is maintained in the global memory. Each node runs a service thread in addition to the main program thread. The purpose of the service thread is to periodically extract and process the messages from the request buffer. Therefore, communication among the participating nodes is accomplished by putting and extracting messages to and from the global memory buffer.

3.2 Main Data Structures

3.2.1 Shared Pointer

The shared pointer is represented by the following structure:

```
typedef struct {
```

```
  unsigned long va:43;
  unsigned int phase:10;
  unsigned int thread:11;
} _UPCRTS_shared_pointer;
```

As can be seen, each shared pointer has three fields:

• va represents the virtual address (or local address) on a node

• phase represents the number of elements in block-cyclic block and speeds up the pointer arithmetic operations.

• thread represents the number of the processor containing the shared storage

3.2.2 Element

```
typedef  struct {
  int type;
  ProtocolType ptype;
  unsigned long datum;
  tag tagg;
  unsigned char* address_to;
  unsigned char* address_from;
 long   block_size;
  unsigned char block[BLOCK_SIZE];
}element;
```

This structure represents a message in the described message-passing scheme.

The description for each of the fields of the element data structure is given below:

• type stands for message type. Depending on operation, type can be one

one of the following:

```
/* Put Requests */
PUT_FLOAT_REQUEST
PUT_DOUBLE_REQUEST
PUT_TEMP_BLOCK_REQUEST
PUT_BLOCK_REQUEST
```

```
/* Get Requests */
GET_ONE_BYTE_REQUEST
GET_TWO_BYTE_REQUEST
GET_FOUR_BYTE_REQUEST
GET_EIGHT_BYTE_REQUEST
GET_BLOCK_REQUEST

/* Put Responds */
PUT_ONE_BYTE_RESPOND
PUT_TWO_BYTE_RESPOND
PUT_FOUR_BYTE_RESPOND
PUT_EIGHT_BYTE_RESPOND
PUT_FLOAT_RESPOND
PUT_DOUBLE_RESPOND
PUT_BLOCK_RESPOND

/* Get Responds */
GET_ONE_BYTE_RESPOND
GET_TWO_BYTE_RESPOND
GET_FOUR_BYTE_RESPOND
GET_EIGHT_BYTE_RESPOND
GET_TEMP_BLOCK_RESPOND
GET_BLOCK_RESPOND

/* Purge */
PURGE_REQUEST
PURGE_RESPOND
PURGE_MEM_GET_REQUEST
PURGE_MEM_GET_RESPOND
UPDATE_REQUEST
UPDATE_RESPOND
```

- ptype: cache protocol type. The meaning of this type will be explained in Chapter 4.
- datum: a data to be transferred.

Note, that all the data is represented by an unsigned long. Unsigned long is represented on Alphas (the underlying nodes) as 8 bytes, which is the longest size for elementary data types. This way any type of data can fit in unsigned long.

• tag - when the message is injected into the network, a unique number sequence is assigned to it. This sequence number is represented by the field tag

• address_to: represents the data address local to the remote node at which data is stored (Put operation)

• address_from: represents the data address local to the remote node remote node from which the data is received (Get operations)

• block size: this field represents the size (in bytes) of data to be transferred

• block: is a block buffer for GetBlock and PutBlock operations

3.2.3 Processor Queue

Each node has request queues associated with it in a shared global space. These queues are accessible by all communicating nodes. When one processor needs to send a message to another processor, it just deposits this message into the other processor's queue.

Each queue is represented by the following data structure:

```
typedef struct {
  que     nord;
  que     rd;
  boolean serviceFlag;
} procQue;
```

Note, that the procQue data structure has, actually, not one but two queues: "nord" and "rd". The "nord" queue is for messages that do not require a reply. At the same time "rd" is the queue for the messages that do require a reply.

3.2.4 Global Region

The global_space data structure represents the shared global space that is accessible by all participating nodes.

```
typedef struct {
  procQue queues[NUMBER_OF_PROCS][NUMBER_OF_PROCS];
  volatile int syncCount; /* a flag for syncronization start */
  barrStruct  barArray[MAX_NUMBER_OF_BARRIERS];
} global_space;
```

Making this structure accessible by all nodes is a tricky task. In order to allocate this structure in the shared global space (which is a region of Memory Channel address space), each processor maps this region into its own process virtual address space using the **imc_asattach()** system call (see API for MC [3]). Once a node is mapped, the region to receive and transmit data, an area of virtual address space of the same size as the Memory Channel region is added to each processor virtual address space.

The global_space structure fields are:

- queues - this is a two dimensional array of processor queues.
- syncCount- a flag for initial syncronization.
- barrArray: used to implement barrier operation.

As can be seen from above, each node is associated with the NUMBER_OF_PROCS request queues. When the remote node deposits its request, this node puts it into one of the request queues that corresponds to the target node's id. For example, if a node with id equal to 1 sends the message to node with id equal to 2, then node 1 deposits the message into **queues[2][1]**.

This design requires more storage compared to the usual one request queue design. However, it is more efficient timewise. The explanation of this is very simple. If there was only one queue to deposit the requests for all nodes, then the following situation might be possible: there are several nodes that want to deposit their requests into the request queue. However, in order to deposit a request the node needs first to lock the queue. Thus, while one node deposits the request all other nodes must wait for to the lock to be released and, thus, there is a waste of processing time. Also if the service thread on the node to which

the queue belongs wants to process the requests it needs to wait, potentially as long as the time required for all remote nodes to deposit their messages.

However, with multiple queues each thread has to wait at most for one thread in order to do operations on a queue.Each node does not need to wait until the other nodes deposit their messages. Also the service thread, does not waste time anymore either because it cycles through the queues and processes only the queues that are not locked and have messages in them. Since the situation in which all remote nodes lock their queues simultaneously is very rare, the service thread almost never has to wait. Thus, this design is more time efficient, even though it consumes more space than the typical design.

3.2.5 Tag Buffer

The TagBuffer data structure is located on each node. The purpose of this data structure is to provide Get and Put requests with unique tags as well as to keep track which requests and thus which tags are outstanding (i.e. not completed). A tag can be in one of the three states: *Not_Used*, *In_Process* or *Completed*. *Not_Used* means that the tag does not belong to any request message and can be reused. *In_Process* signifies that this tag has been issued (i.e. belongs) to some request message, and this request has not yet been completed. *Completed* for the tag indicates that the request associated with the tag has been completed.

Each tag has storage associated with it. The responses to the Get requests store the results in these preallocated data structures. When GetSync (described later) command is executed, it first checks whether the state of the tag is *Complete*. If it is, the GetSync routine extracts the data from the storage associated with the tag, sets the tag's state to *Not_Used*, and returns the value.

To support the described behavior, the tagBuffer Module (see Appendix A) implements the following operations on the tagBuffer data structure:

For Get type requests:

```
tag acquire_GetTag()
boolean isGetTagOutstanding(tag tg)
boolean areGetTagsOutstanding()
```

```
void setGetDatum(tag tg,unsigned long  vl)
unsigned long retrieveGetDatum(tag tg)
void setStatusArrived(tagBuffer* tgBuff, tag tg)
void deleteGetTag(tag tg)
void deleteArrivedTags(tagBuffer*)
```

For Put type requests:

```
tag acquire_PutTag()
boolean isPutTagOutstanding(tag tg)
boolean arePutTagsOutstanding()
void setStatusArrived(tagBuffer* tgBuff, tag tg)
void deleteArrivedTags(tagBuffer*)
int deletePutTag(tag tg)
```

3.3 Main Modules

3.3.1 Get Module

The compiler translates the statement of the type: *x=y*; where *y* is a shared variable into the following sequence of statements *Put (GetSync(GetByte(....)))*. If *y* is a structure, then the compiler translates the *x=y;* statement into *GetBlockSync(GetBlock(...))* sequence of statements.

GetBytes, GetBlock, GetSync, GetBlockSync and GetAllSync routines are described below.

3.3.1.1 GetBytes Routine

GetBytes command has the following interface:

```
_UPCRTS_context_tag _UPCRTS_GetBytes(_UPCRTS_SHARED_POINTER_TYPE
                              adr,size_t size)
```

The purpose of atomic GetBytes is to start a load operation from remote or local memory. In other words, GetBytes injects a request message into the network to get the data. This routine immediately returns (before the remote or local fetch operation is completed).

GetBytes takes as an input two parameters: a shared address pointer to the location being fetched and size of the data. The variables of type int or float correspond to size=4, variables of type double or unsigned long correspond to size = 8 and variables of type short correspond to size = 2. The return value is an anonymous data structure called Tag, which is used by the syncronization routines to describe which Get operation is being completed.

Here is a high-level pseudocode for GetBytes routine:

1.Parse the shared pointer *adr* to determine the remote processor's id and the local pointer (va) on the remote processor at which to get the data.

2. If the processor's id = MYTHREAD (i.e local request)
   get the value from local memory deposit it in a location that corresponds to GET_SHORTCUT_TAG tag and return the GET_SHORTCUT_TAG tag.

3.If the request is for remote data, then
   Get the tag from TagModule and set up the request data message.
   Call the Service Thread routine.
   Lock the global space and deposit a request for data in the "rd" queue that corresponds to the id's remote processor.
   Unlock the global space.
   Return the tag.

The reason for calling a serviceThread routine before depositing a message is to process the messages that are located in the "nord" queue of the local processor before depositing the message into "rd" que of the remote processor. By doing so, a potential deadlock situation is avoided. The deadlock problem will be discussed in Section 3.4 in more detail.

3.3.1.2 GetBlock Routine

This command has the following signature:

```
_UPCRTS_context_tag_UPCRTS_GetBlock(_UPCRTS_SHARED_POINTER_TYPE
            addr_from, void *address_to,size_t block_size)
```

The GetBlock operation fetches data from a remote or local address, specified by the shared pointer *adr_from* to the local address, specified by a regular (local) pointer called *address_to*. The number of bytes to be transfered is specified by *size*. As with GetBytes, GetBlock operation starts this transfer and returns a Tag for syncronization purposes.

The high-level pseudocode for GetBlock routine is almost identical to GetBytes routine. The only difference is that the message type is GET_BLOCK_REQUEST. Also, if the data is local, then the MC's **imc_bcopy()** command is used to copy the data instead of conventional C language **bcopy()** command. It has been shown that **imc_bcopy()** command is more efficient [3] and that is why this implementation employs it.

3.3.1.3 GetSync Routine

Depending on the type of data, this routine has the following interface:

```
unsigned long _UPCRTS_GetSyncInteger(_UPCRTS_context_tag t)
float  _UPCRTS_GetSyncFloat(_UPCRTS_context_tag t)
double  _UPCRTS_GetSyncDouble(_UPCRTS_context_tag t)
```

The appropriate GetSync Routine is called when data is needed for the computation. A Get Routine for the correct size of data must be called before the GetSync routine is called. The GetSync routine waits until the data transfer is completed and returns the value to the executing program. The GetSync routine also frees the tag structure.

GetSync routine works as follows. First, a check is performed on the state of the tag. If the tag's state is not *Complete*, then the GetSync routine is blocked until the state of the tag becomes *Complete*. After that, the data is extracted from the tag storage, and the tag is

freed (this is done by setting its state to *Not_Used*). The GetSync routine casts the data to the appropriate type and returns the data.

3.3.1.4 GetBlockSync Routine

This routine has the following interface:

```
void _UPCRTS_GetBlockSync(_UPCRTS_context_tag t)
```

The GetBlockSync operation is performed just before the data is accessed locally. The Tag(_UPCRTS_context_tag) is used to specify which Get operation is being synchronized. This tag must be generated by a GetBlock operation apriori. The GetBlockSync operation waits for the completion of the data transfer and then returns. The tag reference is then deallocated.

The implementation details are almost the same as for the previous GetSync routine. However, in this case, the data is not returned.

3.3.1.5 GetAllSync Routine

The routine has the following interface:

```
void _UPCRTS_GetAllSync()
```

The GetAllSync routine waits for all outstanding Get operations to complete. Note, however, that the occurrence of this operations does not prevent the need for each Get operation to have a matching GetSync operation. The GetAllSync routine simply waits for all Get operations that are outstanding to complete and then returns.

In the implementation, GetAllSync routine waits till **areGetTagsOutstanding()** routine returns false and frees all the tags. **areGetTagsOutstanding()** routine, implemented in the TagBuffer module, accesses the TagBuffer structure on the local node and returns true if the counter of outstanding requests in the TagBuffer structure is not equal to zero.

3.3.2 Put Module

The UPC compiler translates the assignment statements of type *x=const*, where *x* is a shared variable, into the following statements: *_UPCRTS_Put{Integer,Float,Double}(x,const,...).*

The statements of the type *x=k*, where *x* is a shared variable and *k* is a structure are translated into *_UPCRTS_PUTBLOCK(x,k,...)*

Described below are PutInteger, PutFloat, PutDouble, PutBlock, PutBlockSync and PutAllSync routines.

3.3.2.1 Put Routines

This module implements the following interface:

```
void _UPCRTS_PutInteger(_UPCRTS_shared_pointer p,unsigned long datum,
                        size_t  size)
void _UPCRTS_PutFloat(_UPCRTS_shared_pointer p,float datum)
void _UPCRTS_PutDouble(_UPCRTS_shared_pointer p, double datum)
```

The put routines start a store operation into a remote or local memory location. This location is specified by the shared pointer, which is the first argument. The second argument is the data to be stored. The PutInteger routine has size as the third argument, that specifies the actual size of the data. PutFloat and PutDouble routines are for storing float or double types of data correspondingly. Once the Put operation has started, the routine returns. No tag value is returned since no syncronization is needed.

Below is a high-level pseudocode for Put routine:

1.Parse the shared pointer *adr* to determine the remote processor id and the local pointer on the remote processor at which to store the data.

2.If the processor's id = MYTHREAD (i.e local request)

store the data to the local memory and return;

3. If the request is to store remote data, then:

- Get the tag from TagModule and set up the request data message.

  (set the type to one of put requests, set the tag, set the address_to, set the datum)
- Call the Service Thread routine.
- Lock the global space and deposit a request to store data into the "rd" queue that corresponds to the id's remote processor.
- Unlock the global space and return.

During the message setup, the type of a request is set to one of the following depending on the type of the data to be put:

PUT_ONE_BYTE_REQUEST

PUT_TWO_BYTE_REQUEST

PUT_FOUR_BYTE_REQUEST

PUT_EIGHT_BYTE_REQUEST

PUT_FLOAT_REQUEST

PUT_DOUBLE_REQUEST

Notice that the tag for put request is acquired and sent along with the message (see the message format), even though the put operation does not return the data. By acquiring the tag, the TagBuffer data structure gets updated to reflect the fact that the put message is outstanding. PutAllSync relies on that because it needs to know whether there are any outstanding put operations.

3.3.2.2 PutBlock Routine

Interface:

```
UPCRTS_context_tag _UPCRTS_PutBlock(_UPCRTS_shared_pointer pointer,
                    void *local_buf, size_t size)
```

The PutBlock operation starts the transfer of data from the local memory specified by the local pointer *local_buff* to the remote memory location specified by the shared pointer

pointer. size specifies the number of bytes to be transfered. After the transfer is started the operation returns a tag, which is used for synchronization purposes.

If the request to store the data is local, then:

1. **imc_bcopy()** command is used (instead of the usual **bcopy()**) for efficiency reasons.
2. SHORTCUTTAG tag is returned. This tag is of special type. It would signal the PutBlockSync operation that would follow to simply return without doing any actions.

If the request to store the data is remote, then:

PutBlock routine determines the size of the block of data to be send. If needed, the routine fragments the data into several messages because a message data structure has a limited capacity buffer to hold the block data. In fact, the element buffer can hold BLOCK_SIZE bytes of data (see Section 3.2.2). All but the last message are deposited into the "nord" queue of the remote processor. The last message is deposited in the "rd" queue. This way only one acknowledgement (for the last message) is required. The messages that are deposited in the "nord" queue do not require a reply. Also, according to the implementation, these messages will be processed before the message that has been put into the "rd" queue. Thus, the moment the last message is processed and the reply is sent and acknowledged, the state of the tag on the processor that originated the block data request is changed from the *In_Process* to *Not_Used* state. If PutBlock needs to deposit a message into a remote processor's queue, but the queue is full, PutBlock routine is blocked until the queue can accept more messages.

3.3.2.3 PutBlockSync

Interface: `void _UPCRTS_PutBlockSync(_UPCRTS_context_tag t)`

The PutBlockSync operation waits for the completion of the PutBlock operation specified by the tag. The compiler must perform a PutBlockSync operation after a PutBlock

operation and before the next store into the local memory area being transfered. After the PutBlockSync operation, the tag's state is set to *Not_Used.*

If the tag passed to PutBlockSync operation is SHORTCUTTAG - the transfer is local, and has already occurred and, in this case, the routine simply returns.

In all other cases, the PutBlockSync routine waits for the state of the tag to be *Completed*, and then returns.

3.3.2.4 PutAllSync

Interface: `void _UPCRTS_PutAllSync(void)`

The occurrence of this operation does not obviate the need for matching PutSync operations for each block Put operation. The PutAllSync routine simply waits for all outstanding Put operations (atomic or block) to finish and then returns.

The PutAllSync operation implementation merely waits un till the counter of outstanding (tags with the state equal to *In_Process* state) Put operations becomes zero. Once the counter is zero the PutAllSync returns.

3.4 Deadlock and how to avoid it.

In this section it will be shown how using the "nord" and "rd" message buffers avoids the potential deadlock situation.

If instead of "nord" and "rd" buffers for each node there would be only one message queue, then the following situation might be possible. Let the system consist of two nodes P1 and P2 to simplify the analysis. If P1 and P2 both send many Get type messages to each other at the same time, the message queues for both P1 and P2 nodes might become full. Then, when the service thread on any node tries to process a Get request and send a Get reply to the other node, it can get deadlocked since there is no space in the message queue of the other node. At the same time the other node can not process any message in its message queue either for the same reason - the other node's queue is full and no more messages can be put in this queue.

In order to avoid such a scenario, two different message queues were implemented for each node: the "nord" queue for messages that do not require a reply (a reply does not need a reply) and "rd" queue for messages that do require a reply. Also, there is a service routine that periodically process messages in both queues.

In the current implementation, let "rd" queues for P1 and P2 be full. Now node P1 can process "rd" queue and deposit replies in "nord" queue of the other node. If at some moment the nord queue would become full the service routine would empty it - nothing can stop the service routine since the messages in this queue do not require a reply. The same scenario happens to the other processor and the deadlock is successfully avoided.

3.5 Service Module

3.5.1 Service Routine

The service routine is a utility that extracts messages from the message queues and processes them. This utility function is either called by the service thread or directly invoked before each get/put operation from the main program thread.The service thread is run on each node. For efficiency reasons, the service routine runs periodically.

The service routine is invoked from the main thread before each get or put operation is initiated. If the service thread is not running at the time of invocation, the call to the service utility goes through, otherwise the call returns.

The reason for such a complex operation is efficiency, since running the service thread all the time consumes CPU time. The most preferred solution is to run the service routine before each get or put operation and not to have an additional thread at all. However, such a scenario under certain conditions might result in a deadlock. That is why there is a need for a separate thread that would run service routine periodically.

The routine first tries to get a special service lock. If it fails to acquire such a lock, it means that other thread is already running the service routine, and there is no need to invoke it. In this case, the routine returns. However, if it acquired the lock, then the routine tries to process "nord" queues for this node. Recall that the "nord" queue corresponds to

reply messages of remote processor to the current processor requests. Thus, if there are N remote processors, then there are N "nord" queues on each node. The routine tries to lock one "nord" queue at a time and processes it by calling the *ProcessElement_nord* routine. If it fails to obtain a lock for a particular "nord" queue it does not wait and proceeds to the next "nord" queue. After that, the service routine tries to process "rd" queues in a similar fashion by calling ProcessElement_rd routine. In addition to locking an "rd" queue, the service routine must lock the corresponding "nord" queue of the remote node (when processing the "rd" queue, the replies are generated and must be deposited in the "nord" queue of the remote node).

3.5.2 ProcessElement Routines

Now, let us look at the ProcessElement_nord and ProcessElement_rd routines.

3.5.2.1 ProcessElement_nord Routine

This auxiliary routine extracts messages from the "nord" message queue and processes them. The routine processes messages of the following types:

PUT_ONE_BYTE_RESPOND

PUT_TWO_BYTE_RESPOND

PUT_FOUR_BYTE_RESPOND

PUT_EIGHT_BYTE_RESPOND

PUT_BLOCK_RESPOND

GET_ONE_BYTE_RESPOND

GET_TWO_BYTE_RESPOND

GET_FOUR_BYTE_RESPOND

GET_EIGHT_BYTE_RESPOND

GET_TEMP_BLOCK_RESPOND

GET_BLOCK_RESPOND

Upon receipt of the respond message, the processElement_nord routine sets the state of the tag corresponding to the message (that is the same tag as for originating operation request message tag) to *COMPLETE* state, so that the subsequent Sync operation will know that the tagged operation has completed.

3.5.2.2 ProcessElement_rd Routine

This routine extracts messages from the "rd" message queue that corresponds to the local node and processes these messages. As mentioned above, the "rd" queue contains messages that require replies. Thus, the implementation of this routine is more difficult than the processElement_nord routine.

This routine processes the messages of the following types:

PUT_ONE_BYTE_REQUEST

PUT_TWO_BYTE_REQUEST

PUT_FOUR_BYTE_REQUEST

PUT_EIGHT_BYTE_REQUEST

PUT_TEMP_BLOCK_REQUEST

PUT_BLOCK_REQUEST

GET_ONE_BYTE_REQUEST

GET_TWO_BYTE_REQUEST

GET_FOUR_BYTE_REQUEST

GET_EIGHT_BYTE_REQUEST

GET_BLOCK_REQUEST

If the the request is a Put type request, then the procedure extracts the data from the message and stores it in the local node address space. Once this is done, a reply message is generated and deposited in the "nord" queue of the processor that has sent the original request. In the case of PUT_TEMP_BLOCK_REQUEST the reply is not generated, but it

is generated in the case of PUT_BLOCK_REQUEST that corresponds to the last message of the series of messages to get the block data.

If the request is a Get type request, then the procedure gets the requested data from the local memory, sets up a reply message with the data and the tag and deposits the newly created message into the "nord" queue of the processor that has sent the original request. However, if the request is for block data, then the size of the message buffer designed to hold the block data might be smaller than the requested block data size. In this case, several messages with different parts of block data are created and injected into the network. The "nord" buffer into which the messages are deposited might become full. In that case, the Get_Block_Request message is modified: the "from" and "to" addresses are modified in order to account for messages with partial block data already sent. The modified message is then put back into the "rd" queue of the current node, so that the next time the service routine runs, it will start with the modified message and process it further.

3.6 Barrier Routine

Interface: `void UPCRTS_Barrier(int value)`

The barrier operation completes all the previous communications involving data transfer and then waits for all other processors to reach the same barrier. The routine takes an optional integer parameter. If all of the processors do not provide the same value at the barrier then the program stops and the error is reported.

The implementation of barrier module is described next.

An array of barStruct is allocated in the global space. The purpose of this array is to provide syncronization for the barrier routine.

```
typedef struct {
  volatile int barCount; /* a flag for barrier syncronization  */
  int barValue; /* a flag for barrier syncronization  */
  volatile int barError; /* a flag for barrier error  */
}barrStruct
```

When a barrier statement is encountered, the first thing each node does is it executes GetAllSync() and PutAllSync() series of statements. Once the program counter passes those statements, it means that all the previous communications involving data transfer are complete. Now, the node needs to wait until all other nodes reach the same barrier statement. The barrier statement serves, in a sense, as a rendezvous point for all the nodes. The mechanism for accomplishing this follows.

Once the GetAllSync and PutAllSync statements are executed, the barrier routine checks whether an error has occurred. This is accomplished by checking the *barError* flag. If this flag is set, it means that the error has occurred at some other node and the program must exit. The program outputs the error message and exits.

If the *barError* flag is not set, then the barrier routine compares its parameter to *barValue*. The *barValue* represents the value of the parameter passed to the barrier on all nodes. It is set by the first node that enters the barrier statement. If the *barValue* and the parameter passed to the barrier routine are not equal, and the *barValue* is not equal to the default value (a check whether it is the first node to enter the barrier statement), then it means that at least two different nodes have reached the barriers with different values. In this case *barError* flag is set and the program exits. The *barError* flag serves as a way to tell other nodes that the error has occurred, and that they should exit.

On the other hand if the *barError* flag is not set, the program increases the *barCount* by one and checks whether the *barCount* value is equal to the number of processors that participate in the communication process (NUMBER_OF_PROCS). If *barCount* has reached this number, all the nodes have reached the same point, and the barrier routine returns. The same will happen at all other nodes. If the *barCount* has not yet reached the NUMBER_OF_PROCS, the barrier routine waits for this to happen and loops checking periodically the values of *barError* and *barCount*. In order to allow the *barCount* and *barError* variable to be changed dynamically by other processors, they are declared volatile. The update of any bar variables is atomic and requires obtaining a lock.

3.7 Signals and Errors

The following four signal might be raised during the execution of the program:

ILLEGAL_SHARED_POINTER - this signal occurs if the shared pointer passed to one of the routines was not of valid form.

ILLEGAL_REMOTE_ADDRESS - this signal occurs if the shared pointer referenced non - existent memory in the remote thread. A pointer that is all zeros is a shared NULL pointer. A dereference of a shared NULL pointer will raise this exception.

PROGRAM_TERMINATION - this signal occurs if one of the threads have terminated abnormally. This signal will be raised in all other threads to cause the termination of the UPC program.

3.8 Results

Ping-pong.c program was used as a benchmark to calculate the bandwidth and the time passed for Get and Put operations. Described below are the results.

3.8.1 Get Operation

3.8.1.1 Get 2 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	0.960	0 secs 4167 microsecs
10	1.263	0 secs 31666 microsecs
100	1.404	0 secs 285000 microsecs
1000	1.464	2 secs 731667 microsecs

3.8.1.2 Get 4 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	1.9203	0 secs 4166 microsecs
10	2.5263	0 secs 31667 microsecs
100	2.8070	0 secs 285000 microsecs
1000	3.0255	0 secs 644166 microsecs

3.8.1.3 Get 8 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	4.800	0 secs 3333 microsecs
10	4.923	0 secs 32500 microsecs
100	5.664	0 secs 282500 microsecs
1000	5.779	2 secs 759167microsecs

3.8.2 GetBlock Operation

3.8.2.1 Get Block size=4 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	1.920	0 secs 4166 microsecs
10	2.342	0 secs 34166 microsecs
100	3.117	0 secs 256667 microsecs
1000	3.409	0 secs 346666 microsecs

3.8.2.2 Get Block size=40 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	19.198	0 secs 4167 microsecs
10	23.415	0 secs 34166 microsecs
100	30.094	0 secs 26834 microsecs
1000	33.910	2 secs 359167 microsecs

3.8.2.3 Get Block size=400 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	96.004	0 secs 8333 microsecs
10	145.455	0 secs 55000 microsecs
100	179.105	0 secs 446666 microsecs
1000	197.531	4 secs 50000 microsecs

3.8.2.4 Get Block size=4000 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	177.778	0 secs 45000 microsecs
10	303.798	0 secs 263333 microsecs
100	335.664	2 secs 383333 microsecs

3.8.3 Put Operation

3.8.3.1 Put 2 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	1.200	0 secs 3333 microsecs
10	1.455	0 secs 27500 microsecs
100	1.708	0 secs 234167 microsecs
1000	1.853	2 secs 158334 microsecs

3.8.3.2 Put 4 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	2.400	0 secs 3334 microsecs
10	2.823	0 secs 28334 microsecs
100	3.529	0 secs 226667 microsecs
1000	3.656	2 secs 188333 microsecs

3.8.3.3 Put 8 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	4.800	0 secs 3333 microsecs
10	5.189	0 secs 30834 microsecs
100	6.784	0 secs 235837 microsecs
1000	7.436	2 secs 151666 microsecs

3.8.4 PutBlock Operation

3.8.4.1 Put Block size=4 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	1.920	0 secs 4167 microsecs
10	2.462	0 secs 32500 microsecs
100	3.028	0 secs 264166 microsecs
1000	3.238	2 secs 470833 microsecs

3.8.4.2 Put Block size=40 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	19.203	0 secs 4166 microsecs
10	25.263	0 secs 31667 microsecs
100	30.968	0 secs 258334 microsecs
1000	31.209	2 secs 563334 microsecs

3.8.4.3 Put Block size=400 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	106.667	0 secs 7500 microsecs
10	117.074	0 secs 68333 microsecs
100	149.068	0 secs 53667 microsecs
1000	149.091	5 secs 365833 microsecs

3.8.4.4 Put Block size=4000 bytes

Number of messages	Bandwidth Mbytes/sec	Time passed
1	177.778	0 sec 45000 microsecs
10	271.955	0 sec 294166 microsec
100	284.276	2 sec 814167 microsecs

Chapter 4

Adaptive Cache Coherence Protocol

Presented below is an implementation of Cachet - an adaptive cache coherence protocol for distributed systems. Cachet was designed by Xiaowei Shen, Arvind and Larry Rudolph [4]. This paper presents the first actual implementation of the protocol on top of the basic communication layer described in Section 3 above.

4.1 Motivation

Many programs have various access patterns, and empirical data suggest that no fixed cache coherence protocol would work well for all the access patterns [6,7,8]. Thus, there is an evident need for an adaptive cache-coherence protocol that would automatically switch from one protocol to another to adapt to changing memory access patterns.

4.2 Cachet overview

The implementation Cachet is a seamless integration of two micro-protocols, each of which has been optimized for a particular memory access pattern. The protocol changes its actions to address the changing behavior of the program. Limited directory resources is no problem for Cachet - it simply switches to an appropriate micro-protocol when it runs out of directory space. Cachet embodies both intra and inter protocol adaptivity and achieves higher performance under changing memory access patterns.

4.3 CRF Memory Model[5]

In order for Cachet to work correctly it implements the CRF (Commit-Reconcile&Fences) memory model. CRF is a mechanism-oriented memory model that exposes the data replication and instruction reordering at the programming level. CRF decomposes

load and store instructions into finer-grain operation that operate on local (semantic) caches. There are five memory-related instructions: **Loadl** (load-local), **Storel** (store-local), **Commit**, **Reconcile** and **Fence**.

The purpose of **Loadl** instruction is to read the data from the cache (in this case a software cache). If the data is in the cache, the operation returns the data, otherwise it stalls and waits for the data to be fetched into the cache.

The **Storel** instruction simply writes the data into the cache if the address is cached and sets the cache cell's state to Dirty. CRF allows store operations to be performed without coordinating with other caches. As a result, different caches can contains cells with different values for the same address.

The Commit and Reconcile instructions are used to insure that the data produced by one node can be observed by another node. A Commit instruction on dirty cells will stall until the data is written back to the memory. A Reconcile instruction stalls on a clean cell until the data is purged from the memory. The memory behaves, in a sense, as a rendezvous point, for the writer and the reader.

The next sections will describe these operations and implementations of Cachet-Base and Cachet-WriterPush with these operations.

4.4 New Get and Put Operations

The Get or Put operations in view of CRF Model can be thought of as follows:

Get(a) = Reconcile(a); Loadl(a);

Put(a) = Store(a,v); Commit(a);

Both Loads and Stores are performed directly on local caches. The Commit instruction ensures that a modified value in the cache is written back to the memory, while the Reconcile instruction ensures a stale value is purged from the cache.

There is still a need for sync operations because they are required by the UPC compiler. The implementation of sync routines has not changed from the implementation described in Section 3.

There is another twist in implementation of Get and Put routines: since the Cachet protocol switches from one micro-protocol to another depending on memory access behavior, there is a need to account for each load or store request. Thus, the first thing the get or put routine does is to call *determineProtocolType* routine. This routine updates the data structure which accounts for the number and the relative order of store/load requests on a particular address. It also determines which protocol: Base or WriterPush protocol should be associated with the address and returns this protocol type as a result. The Get or Put routine passes this protocol type along with the address to Reconcile, Loadl, Store, or Commit routine. The implementation of *determineProtocolType* routine is described in Section 4.15

4.5 Cachet-Base Overview

Cachet-Base protocol is ideal when a memory location is randomly accessed by multiple processors, and only necessary commit and reconcile operations are invoked. Its implementation is the most straightforward, since it uses the memory as the rendezvous point. When a Commit instruction is executed for an address, whose state is Dirty, the data must be written back to the memory before the operation completes. A Reconcile instruction for an address whose state is Clean requires the data to be purged from the cache before the instruction completes. No state is required to be maintained for Base protocol at the memory side.

4.6 Cachet-WriterPush Overview

Cachet-WriterPush micro-protocol is ideal when processors are likely to read an address many times before another processor writes the address. A reconcile operation performed on a clean copy causes no purge operation. Thus, subsequent get(load) operation to the address can continually use the cached data without causing any cache miss.

However, when a Commit instruction is performed on a dirty cell, the Commit instruction cannot complete before the clean copies of the address are updated in all other caches. Therefore, it can be a lengthy process to commit an address that is stored in the Dirty state.

4.7 Software Caches

In order to implement a cache-coherence protocol, a software cache was implemented on each node. For efficiency reasons the cache was implemented as a hashtable. Each cache's cell has an associated state. The state can be one of the following: CleanB, CleanW, CachePending,WbPending or Dirty.

The CleanB state indicates that the data have not been modified since data have been-cached or last written back, and the associated protocol for the data address is the Base protocol. Same is with CleanW. However, in this case the protocol associated with the address is Cachet-WriterPush. The Dirty state indicates that the data has been modified and have not been written back to the memory since then. The CachePending indicates that the value for the memory cell is in the process to be determined. WbPending indicates that write-back operation is in progress.

The cache hashtable stores the entries of type Cache Record. The data structure for a cache record is presented below.

```
typedef struct{
  Address adr;
  ValueType type;
  Value value;
  CacheState state;
  int accessCount;
}CacheRecord;
```

4.8 Memory Directory

There is a separate memory directory on each node. The purpose of the memory directory is to keep information in which caches the local address is stored.

Writer-Push micro-protocol is the principal user of this information. In order to perform a Commit operation for a local address with Dirty state, the Writer-Push protocol finds the nodes in which this local address is cached and sends requests to update the remote cache addresses to the new values. Also, when a Load operation is performed for an address under Writer-Push protocol, the id of the cache where the address is going to be stored gets added to the memory directory on the home node of the address.

For simplicity and efficiency, the memory directory is implemented as a hashtable. The hashTable stores entries of Memory Record type. Presented below is Memory Record data structure.

```
typedef struct{
  Address adr;
short ids[NUMBER_OF_PROCS];
}MemRecord;
```

For each address, an array of ids is kept. Each id represents a cache. If, for example, the address is cached in remote cache on node 2, then ids[2] is set to 1 to indicate that fact.

Loadl, Reconcile, Storel and Commit operations of integrated Cache protocol are presented below.

4.9 Reconcile

Interface: `void Reconcile()`

According to the specifications, of CRF model if the protocol under which the address is cached is Writer-Push, then the Reconcile command should simply return. However, in the case of the Base protocol the purpose of the Reconcile command is to purge the data out of cache.

In order to make the implementation simpler the reconcile command simply returns for all cases. Since a Reconcile instruction is always followed by a Loadl instruction, in order to Get the data the purging capabilities have been directly incorporated into Loadl routine for the Base protocol. By doing so, the correctness of the CRF model has been preserved and the implementation has become simpler.

4.10 Loadl

Interface: `Tag Loadl(Address adr, size_t size, ProtocolType ptype)`

Depending on the protocol type, which is determined for an address by *determineProtocolType* procedure and passed to Loadl as a parameter *ptype,* Loadl performs the following actions:

1. If *ptype* is Base

First, Loadl routine checks whether the address is cached. If it is not cached, the routine makes a new entry for the address in the cache table, sets the state of the address to CachePending state, and sets the address's protocol type to Base. The Loadl instruction then injects a request to get the data into the network. In order to inject the message, the underlying basic communication mechanism is used. The Loadl instruction returns the tag that corresponds to the id of the injected request.

If Loadl found that the address is cached and current status of the address is CleanB, then the value for the address in the cache might be stale, and the most current value must be fetched from the memory. For this purpose, Loadl sets the state of the address to Cache Pending and injects a request for the data into the network. The tag associated with the request is returned.

However, if Loadl found that the address is cached and the current state of the address is CleanW (meaning that the current entry is clean and the associated protocol with the address was WriterPush), then the following actions happen next:

If the address resides on the local processor, then the local cache's id is deleted from memory directory for the memory directory entry corresponding to the address. This is done because the address is now associated with a new protocol. The value that was associated with the address is left unchanged, however. The state for the address is CleanW and meaning that both cache and memory have the most current values for the address, and there is no need to retrieve the value from the memory. Now, the state for the address is simply updated from CleanW to CleanB.

If the address is remote, then updating the memory directory is more complicated. The memory directory for the remote address is located on the remote processor on which the address resides in memory. To delete the local cache id from that remote memory directory, the Loadl routine injects a special Memory Purge Request message into the network. Once this is done, the Loadl routine sets the address's state to CachePending and returns the tag associated with the injected message.

2. If *ptype* is equal to WriterPush

Loadl checks whether the address is cached. If not the steps identical to described for case 1 are taken. The only difference is that the protocol type for the address is set to WriterPush instead of Base. The same actions also happen in case the address is cached and the state associated with the address is CleanB.

However, if the address is cached and the associated state with the address is ClearW, then nothing happens, and routine merely returns.

4.11 Storel

Interface: `void Storel(Address adr, unsigned long datum, size_t size)`

Storel routine checks whether the address is already cached. If the address is not cached, then the routine adds a new entry into the cache and initializes it by storing the value and its address into the entry data structure. Storel changes the cell's state to Dirty and exits.

If the address is already cached, then the value gets updated, and the state for the cell is changed to Dirty.

4.12 Commit

Interface: `void Commitl(Address adr,unsigned long datum,size_t size,`
`ProtocolType ptype)`

Commit always operates on addresses with associated states equal to Dirty. Depending on the protocol type, which is determined for an address by *determineProtocolType* procedure and passed to Storel as a parameter ptype, Storel performs the following actions:

1. If *ptype* is equal to Base the following actions happen:

If address is local, then the routine deletes the address from the local memory directory (if there is an entry for the address there). If the address is not cached in other remote caches (this can be determined by checking the *ids* array values that correspond to address entry in memory directory), the routine operation writes data to local memory, sets the state of the address to CleanB and exits. However, if the address is, in fact, cached in other remote caches then the routine needs to delete those values from other caches. To do this, the routine injects a special Purge Message request into the network. Then, the routine sets the state for the address to WbPending, writes the data to the local memory and exits.

If address is remote, then the routine sets the state to WbPending and calls an auxiliary routine called *addMsgWb*, which will send the message to the remote node and will take care of all the structures' updates.

2. If *ptype* is equal to WriterPush, then the following sequence of actions takes place:

If the address is local, then the routine updates the local memory directory. By looking into the local memory directory, the routine also determines whether the address is cached in remote caches, and, if it is then the routine calls the **addMsgUpdateRequest** routine. The purpose of that routine is to send update messages to the nodes in which the address is cached. Please note that the memory directory will list only the caches in which the address is cached under the WriterPush protocol. If the same address is cached somewhere under Base protocol, this cache address is not reflected in the memory directory for the address. In case the addMsgUpdateRequest is invoked, the Commit routine sets the state for the address to WbPending, writes the data to local memory and returns.

If the address is not cached in remote caches then, the routine writes the data to memory, sets the state to CleanW and exits.

In case the address is remote, the routine sets the state to WbPending and calls an auxiliary routine **addMsgWb**, which will send the message to the remote node and will take care of the updates to all structures.

Described next are the modifications that were made to routines that process messages: processElement_nord and processElement_rd routines.

4.13 Modifications to processElement_rd

The routine was appended to process the messages of the following types: PURGE_REQUEST, UPDATE_REQUEST, PURGE_MEM_GET_REQUEST.

The purpose of these messages is to update the structures on remote nodes, since the local node can not directly operate on other nodes structures and memories. These type of messages are sent to remote nodes, where they are retrieved by the service routine and given to proccessElement_rd routine to be processed and executed. The processElement_rd routine processes these messages and updates the local memory, memory directory and cache as needed. Once that is done the routine generates a corresponding response and sends it to the node from where the request has arrived.

For UPDATE_REQUEST the routine updates the cache entry for address with a new value and sends a response message back. Such type of requests are sent by Commit routines for addresses that operate under WriterPush protocol.

For PURGE_REQUEST type of messages the routine deletes the entry for the address from cache and sends a PURGE_RESPOND message back.

The PURGE_MEM_GET_REQUEST type of messages are sent by node that wants to delete its local cache id for the address(which is remote) from remote memory directory.

The processElement_rd routine deletes the indicated cache id for an address from memory directory and sends back the PURGE_MEM_GET_RESPOND message.

4.14 Modifications to processElement_nord

Analogical modifications have been made to processElement_nord routine. The routine was appended to process messages(responses) of the following types: PURGE_RESPOND, UPDATE_RESPOND, PURGE_MEM_GET_RESPOND. Once the responses have arrived the processElement_nord routine sets the state for address in local cache to be either ClearB or ClearW depending on the protocol type.

Described next is determineProtocolType procedure - a procedure that determines which protocol to use for a particular address.

4.15 determineProtocol Routine

Cachet implements the dynamical switching from one protocol to another for an address. The **determineProtocol** routine represents the routine that determines whether to switch to a different protocol or not. Each time the Get or Put routine is executed the **determinerProtocol** routine is called. The routine calculates the number of Get operations that are uninterrupted by PUTs and the number of Put operations that are uninterrupted by GETs for the address. For the address under the Base protocol the moment the number of uninterrupted Gets exceeds NUMBER_OF_LOADS_TO_SWITCH value the protocol is changed to WriterPush. Similarly for the address that is operated on under WriterPush Protocol: the moment the determine protocol calculates the number of uninterrupted Puts exceeds the NUMBER_OF_STORES_TO_SWITCH value the WriterPush protocol is substituted by Base. The explanation for this is that if there are plenty of loads, then the address better off to be operated under WriterPush protocol and if there are lots of Puts the program better off to switch to Base protocol. By employing the described amortized scheme the determineProtocol routine dynamically determines which protocol to pick.

The two parameters of the routine: NUMBER_OF_LOADS_TO_SWITCH and NUMBER_OF_STORES_TO_SWITCH can be changed. By doing this the performance may be fine-tuned for a particular range of applications.

4.16 Results

The benchmark suite consists of three programs: Heat-Transfer Program, BaseSimul and BaseWriterSimul programs. The code of those programs is given in Apendix A.

In order to see the advantage of the run-time library enhanced with Cachet the performance of the benchmark suite was measured on a system with the communication library enhanced with only Base coherency protocol, on a system with the communication library enhanced with only Writer coherency protocol, and finally on a system with the communication library enhanced with Cachet (the integrated protocol).

4.16.1 Heat-transfer Program

The first test was performed on Heat-transfer program.

numOfIters	with Base	with Writer	with Base and Writer
10	3 secs 121667 usecs	0 secs 508333 usecs	0 secs 910833 usecs
100	27 secs 937500 usecs	3 secs 58335 usecs	3secs 464833 usecs
200	56 secs 151666 usecs	5 secs 850000 usecs	6 secs 188334 usecs
300	85 secs 325833 usecs	8 secs 678333 usecs	9 secs 65834 usecs
500	142secs 606667usecs	14 secs 514166 usecs	14 secs 633334 usecs
800	226secs 455833usecs	22 secs 690833 usecs	22 secs 989167 usecs

The Heat-transfer program does a lot of loads and relatively few writes. The Writer-Push protocol exploits this. The most perfect protocol for this program as can be seen from the data is WriterPush. The Base protocol loses a lot on each iteration. The integrated Cachet is very close in performance to WriterPush and as the number of iterations increase the performance gets closer to WriterPush performance.

4.16.2 BaseSimul Program(Simulates Typical Base Behaviour)

The BaseSimul program does a lot of store and few loads. As can be seen from the results below Base is a perfect protocol for it. The integrated Cachet protocol exhibits very good results as well and very close to performance of Base protocol.

numOfIters	with Base	with Writer	with Base and Writer
10	0 secs 540000 usecs	0 secs 562500 usecs	0 secs 540033 usecs
20	1 secs 65123 usecs	1 secs 81667 usecs	1 secs 65756 usecs
30	1 secs 556667 usecs	1 secs 605833 usecs	1 secs 557241 usecs
40	2 secs 119167 usecs	2 secs 334167 usecs	2 secs 120345 usecs

4.16.3 BaseWriterSimul Program

The BaseWriteSimul program changes the behaiviour in a cyclic fashion. It first does a lot of reads, then does a lot of writes, then repeats. The memory access pattern of this program periodically changes. As we can see from the results the integrated Cachet (with Base and Writer) cache-coherency protocol produces best results due to its ability to switch from one protocol to another as the memory access changes.

numOfIters	with Base	with Writer	with Base and Writer
10	0 secs 446667 usecs	0 secs 415000 usecs	0 secs 420000 usecs
20	0 secs 826667 usecs	0 secs 815833 usecs	0 secs 800523 usecs
30	1 secs 260000 usecs	1 secs 224167 usecs	1 secs 210084 usecs
40	1 secs 659167 usecs	1 secs 662500 usecs	1 secs 570000 usecs
50	2 secs 24166 usecs	1 secs 930000 usecs	2 secs 19166 usecs

Chapter 5

Conclusions and Future research

5.1 Conclusions

The paper presented the design of the basic communication run-time library for the UPC language. In particular the implementations of Get, Put and Barrier operations were shown. The design was implemented on top of Memory Channel hardware. This hardware provides the participating nodes with a shared global space via which the communication was accomplished.

The implementation of Cachet, an adaptive cache coherence protocol, using the basic communication layer was also described. This is the first actual implementation of Cachet and this paper served as a case study for it. The two of three micro-protocols for integrated Cachet protocol: Cachet-Base and Cachet-WriterPush were implemented. Using Cachet to optimize the performance of the basic communication layer resulted in improvements of performance up to an order of magnitude. Cachet proved to be a highly efficient adaptive cache coherence protocol and could be used as an optimization to other communication libraries.

Also the results showed that Memory Channel is a reasonable alternative to other exiting interconnect hardware. Now the UPC language is implemented with Cachet coherency scheme and supports the Memory Channel interconnect that provides high bandwidth transfer.

5.2 Directions for Future Work

This section gives directions for future work on optimizing the communication run-time library. The following optimizations are suggested to be implemented: Cachet-

Migratory, Communication Movement, Communication Combination and Calculation Combination. The ideas behind these optimizations are briefly discussed next.

5.2.1 Cachet-Migratory

Due to lack of time only two out of three micro-protocols have been implemented for Cachet. The next step to improve the performance is to implement the third micro-protocol for Cachet: Cachet-Migratory. This micro-protocol gives a further gain in performance when an address is exclusively accessed by one processor for a reasonable period of time. It is reasonable to give the cache the exclusive ownership so that all instructions on the address become local operations.

5.2.2 Communication Movement

The goal of Communication movement is to move the reads earlier in the program. Moving remote reads earlier has several advantages. Because the remote operations are split-phase, by issuing the remote reads as early as possible would allow communication to overlap with the computation that follows the read. By moving the remote reads we can expose some possibilities for discovering redundant communication or opportunities for blocked communication i.e. moving several remote reads together.

Another goal of this optimization is to move remote writes. However, in this case, we have two conflicting goals. If we move remote writes earlier we can overlap communication and computation more, however moving remote writes later may expose the possibilities for pipelining or blocking. There is a very fine balance whether to move remote writes earlier or later.

5.2.3 Communication Combination

Several messages that are bound to the same processor may be combined into a single, larger message. This way we can reduce the number of messages sent, and thus reduce the overall start-up overhead. UPC currently provides a blocked communication mechanism.

However, it has a limited use due to the fact that the block should be continuous. The API for blocked communication for UPC is the following: Tag GetBlock(pointer, long numer_of_bytes)

The block of the length number_of_bytes is fetched from a remote location specified by pointer. This mechanism is useful for continuous blocks of data. However, the facility to combine several requests to fetch a data from a remote processor into one blocked request would be of great use. Such mechanism currently exists for many parallel languages including the standard of parallel communication MPI (gather, scatter). By introducing such mechanism we can further optimize the UPC programs due to the fact that we save a lot on the overheads of injecting and receiving several messages versus only one.

5.2.4 Calculation Movement

If the processor P1 does some calculation N = A + B + C, where A, B and C are remote values and are stored at processor P2, N is stored at local processor P1. The naïve implementation would require three remote fetches from processor P2. However, the whole calculation can be moved to processor P1 and then only one remote write is required instead of three remote reads. Thus, an analysis can be performed and moving certain calculations to other processors can optimize a program.

Appendix A

A.1 Get Module

```
#include "upc.h"
#include "lock_api.h"
#include "pointer_api.h"
#include "buff_api.h"
#include "cache.h"

/* function prototypes*/
void service(int tyoe, int procId);
Tag get(pointer pntr,int type);
unsigned long Loadl(Address,size_t,ProtocolType);
void Reconcile();
unsigned long addMsg_CacheReq(_UPCRTS_SHARED_POINTER_TYPE,size_t,ProtocolType);
unsigned long addMsgPurgeMemGetRequest(Address pointer,unsigned long datum);

_UPCRTS_context_tag _UPCRTS_GetBytes(_UPCRTS_SHARED_POINTER_TYPE adr,size_t size)
{
  unsigned long tg;
  ProtocolType ptype;

  /*Determine the type of Protocol to use and update Adr.access Record*/
  ptype=determineProtocolType(adr,Load);

  Reconcile();
  tg=Loadl(adr,size,ptype);
  return tg;
}

unsigned long Loadl(Address adr,size_t size,ProtocolType ptype)
{
  int index;
  CacheState state;
  ValueType type;
  unsigned long tg;
  CacheRecord *rec;

  rec=cache_findRecord(adr);

  /* 1. Case 1 */
  if(ptype==Base)
    {
      /* 1.a is not in a cache */
      if (rec==NULL)
                {
                  shaddress_loadl=adr;
                  sh_ptype=Base;
                  type=sizeToValueType(size);
                  cache_addRecord(adr,type,1,CachePending);
                  tg=addMsg_CacheReq(adr,size,Base);
                  return tg;
                }
      else
                {
                  /* 2. state== Clean or Dirty*/
                  state=rec->state;
                  if((state==CleanW)||(state==Dirty))
                    {
```

```
                    if(adr.thread==MYTHREAD)
                              {
                                /* Case 1: adr is local */
                                cache_setState(adr,CleanB);
                                mem_deleteId(adr,MYTHREAD);
                                shvalue_loadl=rec->value;
                                return SHTAG_LOADL;
                              }
                    else
                              {
                                /* Case 2: adr is remote */
                                cache_setState(adr,CachePending);
                                /* now we need to delete id from remote memory */
                                tg=addMsgPurgeMemGetRequest(adr,rec->value);
                                return tg;
                              }
                  }
                if(state==CleanB)
                  {
                    shaddress_loadl=adr;
                    sh_ptype=Base;
                    cache_setRecord(adr,CachePending);
                    tg=addMsg_CacheReq(adr,size,Base);
                    return tg;
                  }
              }
    }

  /* 2. Case 2 */
  if (ptype==Writer)
    {
      /* 1.a is not in a cache */
      if (rec==NULL)
              {
                shaddress_loadl=adr;
                sh_ptype=Writer;
                type=sizeToValueType(size);
                cache_addRecord(adr,type,1,CachePending);
                tg=addMsg_CacheReq(adr,size,ptype);
                return tg;
              }
      else
              {
                /* 2. state== Clean or Dirty*/
                state=rec->state;
                if((state==CleanW)||(state==Dirty))
                  {
                    shvalue_loadl=rec->value;
                    return SHTAG_LOADL;
                  }
                if(state==CleanB)
                  {
                    shaddress_loadl=adr;
                    sh_ptype=Writer;
                    cache_setState(adr,CachePending);
                    tg=addMsg_CacheReq(adr,size,ptype);
                    return tg;
                  }
              }
    }
}
```

```
void Reconcile()
{return;}


unsigned long addMsg_CacheReq(_UPCRTS_SHARED_POINTER_TYPE pointer,size_t size,
                                                    ProtocolType ptype)
{
  int           procId;
  local_pointer lp;
  Tag           tagg;
  int type;

  procId = pointer.thread;
  lp     = (unsigned char *)pointer.va;

  /* shortcut case MYTHREAD==procId ***********************************/
  if (MYTHREAD==procId)
    {
      switch((int)size)
               {
               case 1:
                 shortcut_value= *lp;
                 break;
               case 2:
                 shortcut_value= *((unsigned short *)lp);
                 break;
               case 4:
                 shortcut_value= *((unsigned int *)lp);
                 break;
               case 8:
                 shortcut_value= *((unsigned long *)lp);
                 break;
               default :
                 printf("get.c:1: size is incorrect\n");
                 break;
               }
      if(ptype==Writer)
               mem_addId(pointer,MYTHREAD);
      return SHORTCUTTAG;
    }
  /*******************************************************************/


  switch((int)size)
    {
    case 1:
      type=GET_ONE_BYTE_REQUEST;
      break;
    case 2:
      type=GET_TWO_BYTE_REQUEST;
      break;
    case 4:
      type=GET_FOUR_BYTE_REQUEST;
      break;
    case 8:
      type=GET_EIGHT_BYTE_REQUEST;
      break;
    default:
      printf("error::get.c::size is unknown\n");
      exit(-1);

    }
  /* first call serviceFunc */
  service(FUNC,procId);
```

```
  getLockFullTest_rd(procId,MYTHREAD,5);
  tagg = acquire_GetTag();
  assert(tagg!=0);
  assert(addElement_rd(procId, MYTHREAD)== 1);
  setType_rd(procId,MYTHREAD,type);
  setTag_rd(procId,MYTHREAD,tagg);
  setAddressFrom_rd(procId,MYTHREAD,lp);
  setPtype_rd(procId,MYTHREAD,ptype);
  releaseLock_rd(procId,MYTHREAD);
  return (unsigned long) tagg;
}

unsigned long addMsgPurgeMemGetRequest(Address pointer,unsigned long datum)
{
  /* Effects: should purge id=MYTHREAD for pointer from remote Memory */

  local_pointer lp;
  Tag tagg;

  lp = (unsigned char *)pointer.va;
  service(FUNC,procId);
  getLockFullTest_rd(procId,MYTHREAD,5);
  tagg = acquire_GetTag();
  assert(tagg!=0);
  assert(addElement_rd(procId, MYTHREAD)== 1);
  setType_rd(procId,MYTHREAD,PURGE_MEM_GET_REQUEST);
  setDatum_rd(procId,MYTHREAD,datum);
  setTag_rd(procId,MYTHREAD,tagg);
  setAddressFrom_rd(procId,MYTHREAD,lp);
  releaseLock_rd(procId,MYTHREAD);
  return (unsigned long) tagg;
}

unsigned long gethlp(_UPCRTS_context_tag t)
{
  Tag tagg;
  Address adr;
  unsigned long datum;
  CacheState state;
  tagg=(int*)t;

  /*1.Case 1*/
  if (t==SHORTCUTTAG)
    {
      datum=shortcut_value;
      adr = shaddress_loadl;
      cache_setValue(adr,datum,1);
      state=(sh_ptype==Writer)?CleanW:CleanB;
      cache_setState(adr,state);
      return datum;
    }

  /*2.Case 2*/
  if (t==SHTAG_LOADL)
    return shvalue_loadl;

  /*3.Case 3*/


  while(isGetTagOutstanding(tagg))
    /* wait */;
  datum = retrieveGetDatum(tagg);
  deleteGetTag(tagg);
  /* sanity check */
  state=cache_getState(shaddress_loadl);
```

```
  assert((state==CleanB)||(state==CleanW));
  return datum;
}
/*****************************************************************************/

unsigned long _UPCRTS_GetSyncInteger(_UPCRTS_context_tag t)
{ return gethlp(t);}

float  _UPCRTS_GetSyncFloat(_UPCRTS_context_tag t)
{
  unsigned long datum=gethlp(t);
  return *((float*)&datum);}

double  _UPCRTS_GetSyncDouble(_UPCRTS_context_tag t)
{unsigned long datum=gethlp(t);
 return *((double*)&datum);}

void _UPCRTS_GetBlockSync(_UPCRTS_context_tag t)
{
  Tag tagg= (int*) t;

  if (t==SHORTCUTTAG)
    return;              ;

 while(isGetTagOutstanding(tagg))
    /* wait */;
  deleteGetTag(tagg);
  return;
}

void _UPCRTS_GetAllSync()
{
  while(areGetTagsOutstanding())
    /* wait */;
  deleteArrivedTags(getTagBuffer);
}
/*****************************************************************************/
_UPCRTS_context_tag _UPCRTS_GetBlock(_UPCRTS_SHARED_POINTER_TYPE addr_from,
                                     void *address_to,size_t block_size)
{
  int           procId;
  local_pointer address_from;
  Tag           tagg;
  int status;

  procId       = addr_from.thread;
  address_from = (unsigned char *)addr_from.va;

  /* this is a shortcut in case procId==MYTHREAD */
  if (procId==MYTHREAD)
    {
      int prev_err;
      do {
               prev_err = imc_rderrcnt_mr(0);
               imc_bcopy(address_from,address_to,block_size,1,0);
      } while ((status = imc_ckerrcnt_mr(&prev_err,0)) != IMC_SUCCESS);
      return SHORTCUTTAG;
    }
  /*********************************************/



  /* first call service Func */
  service(FUNC,procId);
```

```
  getLockFullTest_rd(procId,MYTHREAD,6);

  tagg = acquire_GetTag();
  assert(tagg!=0);

  assert(addElement_rd(procId,MYTHREAD)== 1);
  setType_rd(procId,MYTHREAD, GET_BLOCK_REQUEST);
  setTag_rd(procId,MYTHREAD,tagg);
  setAddressTo_rd(procId,MYTHREAD,address_to);
  setAddressFrom_rd(procId,MYTHREAD,address_from);
  setBlockSize_rd(procId,MYTHREAD,block_size);

  releaseLock_rd(procId,MYTHREAD);
  return (unsigned long)tagg;
}
/*****************************************************************/

ValueType sizeToValueType(size_t size)
{
  switch((int)size)
    {
    case 1:
      return OneByte;
    case 2:
      return TwoByte;
    case 4:
      return FourByte;
    case 8:
      return EightByte;
    default:
      printf("\nerror::get.c::sizeToValueType::no such size=%d\n",size);
      exit(-1);
    }
}

ProtocolType determineProtocolType(Address Adr,AccessType accessType)
{
  /* Effects: 1.determines protocol to use i.e returns
   *            either Writer or Base
   *          2.updates count of load for adr
   */

  /*  Idea if there is "k" Loads in a row and protocol Base
   *  then swithc to Writer
   *  If there are "k" Stores and protocol is Writer then
   *  switch to Base
   */

  ProtocolType ptype;

  ptype = getPtype(adr);

  if(ptype==UNASSIGNED)
    /* adr is not in the database=> set it up */
      setUpDefault(adr);


  /* If Writer Protocol ***/
  if(ptype==Writer)
    {
      if(accessType==Load)
                {
                  setNumberStoresToZero(adr);
                  return Writer;
                }
```

```
      else
                {
                  /* i.e accessType=Store */
                  increaseNumberStores(adr);
                  if(getNumberStores(adr)==NUMSTORESTOSWITCH)
                    {
                      setNumberLoadsToZero(adr);
                      return Base;
                    }
                }
    }

  /* If Base Protocol ***/
  if(ptype==Base)
    {
      if(accessType==Store)
                {
                  setNumberLoadsToZero(adr);
                  return Base;
                }
      else
                {
                  /* i.e accessType=Load */
                  increaseNumberLoads(adr);
                  if(getNumberLoads(adr)==NUMLOADSTOSWITCH)
                    {
                      setNumberStoresToZero(adr);
                      return Writer;
                    }
                }
    }

}
```

A.2 Put Module

```
/*
 * put.c:: implements the user interface for upc put* procedures
 */
#include "upc.h"
#include "buff_api.h"
#include "pointer_api.h"
#include "lock_api.h"
#include "cache.h"


#define boolean int
#define true    1
#define false   0

/* functions prototypes*/
void service(int type, int procId);
TimesDiff * getTimesDiff(long block_size);
ValueType sizeToValueType(size_t size);
int getOtherThreadId(int mythread);
void Storel(Address adr, unsigned long datum, size_t size);
void Commitl(Address adr,unsigned long datum,size_t size);
int getOtherThreadId(int mythread);
void mem_addId(Address adr,int id);
int  mem_isIdIn(Address adr,int id);
void mem_deleteId(Address adr,int id);
void addMsgWb(_UPCRTS_SHARED_POINTER_TYPE,unsigned long, size_t,ProtocolType);
void put(_UPCRTS_SHARED_POINTER_TYPE ,unsigned long,int,ProtocolType);
void addMsgPurgeRequest(Address adr,ProtocolType);
void addMsgUpdateRequest(Address,unsigned long,ProtocolType);
void writeDataLocalMemory(Address,unsigned long,size_t);
int getOtherThreadId(int);
ProtocolType determineProtocolType(Address,AcessType);

void Storel(Address adr, unsigned long datum, size_t size)
{
  CacheRecord *rec;
  CacheState state;
  ValueType type;

  rec = (CacheRecord *) cache_findRecord(adr);

  if (rec==NULL)
    {/*2. a is not in a cache */
      type = sizeToValueType(size);
      cache_addRecord(adr,type,datum,Dirty);
    }
  else
    { /*1. a is in cache */
      state=cache_getState(adr);
      if((state==CleanB)||(state==CleanW)||(state==Dirty))
                {
                  cache_setState(adr,Dirty);
                  type = sizeToValueType(size);
                  cache_setValue(adr,datum,type);
                }
      else
                exit(-1);
    }
  return;
}
```

```
void Commit1(Address adr,unsigned long datum,size_t size,ProtocolType ptype)
{
  CacheState state;
  int otherThreadId;


  state=cache_getState(adr);

  if(ptype==Base)
    {
      if(state==Dirty)
                {

                  /*** if address is local */
                  if (adr.thread==MYTHREAD)
                    {
                      mem_deleteId(adr,MYTHREAD);
                      otherThreadId=getOtherThreadId(MYTHREAD);
                      if(mem_isIdIn(adr,otherThreadId))
                                {
                                  mem_deleteId(adr,otherThreadId);
                                  /* to kick it out of other cache */
                                  cache_setState(adr,WbPending);
                                  /* now we need to write data to physical memory */
                                  writeDataLocalMemory(adr,datum,size);
                                  addMsgPurgeRequest(adr,ptype);
                                  return;
                                }
                      else
                                {
                                  /* now we need to write data to physical memory */
                                  writeDataLocalMemory(adr,datum,size);
                                  cache_setState(adr,CleanB);
                                  return;
                                }
                    }
                  else
                    {
                      /*** if address is remote */
                      cache_setState(adr,WbPending);
                      addMsgWb(adr,datum,size,ptype);
                      return;
                    }
                }
      else
                {
                  /* other cases: for now ignore them */
                  exit(-1);
                }
    }

  if(ptype==Writer)
    {
      if(state==Dirty)
                {
                  /*** if address is local */
                  if (adr.thread==MYTHREAD)
                    {
                      mem_addId(adr,MYTHREAD);
                      otherThreadId=getOtherThreadId(MYTHREAD);

                      if(mem_isIdIn(adr,otherThreadId))
                                {
                                  /* now we need to write data to physical memory */
                                  writeDataLocalMemory(adr,datum,size);
```

```
                                /* update the other cache */
                                cache_setState(adr,WbPending);
                                addMsgUpdateRequest(adr,datum,ptype);
                                return;
                              }
                        else
                              {
                                /* now we need to write data to physical memory */
                                writeDataLocalMemory(adr,datum,size);
                                cache_setState(adr,CleanW);
                                return;
                              }
                      }
                    else
                      {
                        /*** if address is remote */
                        cache_setState(adr,WbPending);
                        addMsgWb(adr,datum,size,ptype);
                        return;
                      }
                  }
        else
                  {
                    /* other cases: for now ignore them */
                    exit(-1);
                  }
    }
}
/*****************************************************************************/
void addMsgUpdateRequest(Address adr,unsigned long datum, ProtocolType ptype)
{
  /* we mask this message as Put msg.This way PutallSync will detect it */
  put(adr,datum,UPDATE_REQUEST,ptype);
}

void addMsgPurgeRequest(Address adr,ProtocolType ptype)
{
  /* we mask this message as Put msg.This way PutallSync will detect it */
  put(adr,0,PURGE_REQUEST,ptype);
}

void addMsgWb(_UPCRTS_SHARED_POINTER_TYPE pntr,unsigned long datum,
                      size_t  size,ProtocolType ptype)
{
  int type;

  switch((int)size)
    {
    case 1:
      type=PUT_ONE_BYTE_REQUEST;
      break;
    case 2:
      type=PUT_TWO_BYTE_REQUEST;
      break;
    case 4:
      type=PUT_FOUR_BYTE_REQUEST;
      break;
    case 8:
      type=PUT_EIGHT_BYTE_REQUEST;
      break;
    default:
      printf("error::put.c::size is unknown\n");
      exit(-1);
    }
  put(pntr,datum,type,ptype);
```

```
}

/*****************************************************************************/
void _UPCRTS_PutInteger(_UPCRTS_SHARED_POINTER_TYPE pntr,
                                          unsigned long datum,size_t  size)
{
  ProtocolType ptype;

  /*Determine the type of Protocol to use and update Adr.access Record*/
  ptype=determineProtocolType(pntr,Store);

  Storel(pntr,datum,size);
  Commitl(pntr,datum,size,ptype);
}

void _UPCRTS_PutFloat(_UPCRTS_SHARED_POINTER_TYPE pntr,float datum)
{
  unsigned long dt;
  ProtocolType ptype;

  /*Determine the type of Protocol to use and update Adr.access Record*/
  ptype=determineProtocolType(pntr,Store);

  dt=*((unsigned long *)&datum);

  Storel(pntr,dt,4);
  Commitl(pntr,dt,4,ptype);
}

void _UPCRTS_PutDouble(_UPCRTS_SHARED_POINTER_TYPE pntr,double datum)
{
  unsigned long dt;
  ProtocolType ptype;

  /*Determine the type of Protocol to use and update Adr.access Record*/
  ptype=determineProtocolType(pntr,Store);

  dt=*((unsigned long *)&datum);
  Storel(pntr,dt,8);
  Commitl(pntr,dt,8,ptype);
}

/*****************************************************************************/
void put(_UPCRTS_SHARED_POINTER_TYPE pntr,unsigned long datum,
               int request_type,ProtocolType ptype)
{
  int  procId;
  local_pointer lp;
  Tag tagg;

  procId = pntr.thread;
  if((request_type==PURGE_REQUEST)||(UPDATE_REQUEST))
    procId=getOtherThreadId(MYTHREAD);

  lp     = (unsigned char *)pntr.va;

  /* first call service function */
  service(FUNC,procId);

  getLockFullTest_rd(procId,MYTHREAD,7);

  tagg = acquire_PutTag();
  assert(tagg!=0); /*special case */

  assert(addElement_rd(procId,MYTHREAD)==1);
```

```
  setType_rd(procId, MYTHREAD,request_type);
  setPtype_rd(procId, MYTHREAD,ptype);
  setTag_rd(procId,MYTHREAD, tagg);
  setAddressTo_rd(procId,MYTHREAD,lp);
  setDatum_rd(procId,MYTHREAD,datum);

  releaseLock_rd(procId,MYTHREAD);
}
/*****************************************************************************/
void _UPCRTS_PutAllSync(void)
{
  while(arePutTagsOutstanding())
    /* wait */;
  deleteArrivedTags(putTagBuffer);
}
/*****************************************************************************/

_UPCRTS_context_tag  _UPCRTS_PutBlock(_UPCRTS_SHARED_POINTER_TYPE addressto,
                                                                  void *addressfrom,
                                                                  size_t block_size)
{
  local_pointer address_from = (local_pointer) addressfrom;
  local_pointer address_to,address_to_pntr,address_from_pntr;
  int           status = 0;
  Tag           tagg;
  TimesDiff    *temp;
  int procId,i,offset,times,times_rd,times_rd_unch;
  int counter,count,count_rd;
  long diff;
  boolean flag_rd;
  int status1;

  procId    =  addressto.thread;
  address_to = (unsigned char *)addressto.va;



  /***  this is a shortcut in case procId==MYTHREAD ****/
  if (procId==MYTHREAD)
    {
      int prev_err;
      do {
               prev_err = imc_rderrcnt_mr(0);
               imc_bcopy(addressfrom,address_to,block_size,1,0);
      } while ((status1 = imc_ckerrcnt_mr(&prev_err,0)) != IMC_SUCCESS);
      return SHORTCUTTAG;
    }
  /*************************************************/



  tagg = acquire_PutTag();
  assert(tagg!=0);

  temp = getTimesDiff(block_size);
  times = temp->times;
  diff =  temp->diff;

  times_rd_unch = times_rd= (diff==0)?(times-1):times;

  /* first call service Func */
  service(FUNC,procId);

  flag_rd   = true;
  counter = 0;
```

```
  if (times_rd!=0)
    {
      while(flag_rd)
                {
                  getLockFullTest_rd(procId,MYTHREAD,8);
                  count_rd= BUFFER_SIZE - getCount_rd(procId,MYTHREAD);
                  count = (times_rd>count_rd)?count_rd:times_rd;

                  for (i=0;i<count;i++)
                    {
                      assert (addElement_rd(procId,MYTHREAD)!=-1);
                      setType_rd(procId,MYTHREAD, PUT_TEMP_BLOCK_REQUEST);
                      offset=(i+counter)*BLOCK_SIZE; /* zdes' bilo /4 I have no idea why so
I deleted it */
                      address_to_pntr=address_to+offset;
                      address_from_pntr=address_from+offset;
                      setAddressTo_rd(procId,MYTHREAD,address_to_pntr);
                      setBlockSize_rd(procId,MYTHREAD,BLOCK_SIZE);
                      setBlock_rd(procId,MYTHREAD,address_from_pntr,BLOCK_SIZE);
                    }
                  times_rd= times_rd-count;
                  counter = counter+count;
                  assert(times_rd>-1);
                  if (times_rd==0)
                    flag_rd= false;

                  releaseLock_rd(procId,MYTHREAD);

                }/* end of while:flag_rd*/
    }

  getLockFullTest_rd(procId,MYTHREAD,9);
  assert(addElement_rd(procId,MYTHREAD)==1);

  address_to_pntr  =address_to+times_rd_unch*BLOCK_SIZE;
  address_from_pntr=address_from+times_rd_unch*BLOCK_SIZE;
  setAddressTo_rd(procId,MYTHREAD,address_to_pntr);

  if (diff>0)
    {
      setBlockSize_rd(procId,MYTHREAD,diff);
      setBlock_rd(procId,MYTHREAD,address_from_pntr,diff);
    }
  else
    {
      setBlockSize_rd(procId,MYTHREAD,BLOCK_SIZE);
      setBlock_rd(procId,MYTHREAD,address_from_pntr,BLOCK_SIZE);
    }

  setType_rd(procId,MYTHREAD, PUT_BLOCK_REQUEST);
  setTag_rd(procId,MYTHREAD,tagg);

  releaseLock_rd(procId,MYTHREAD);
  return (unsigned long)tagg;
}
/*****************************************************************************/
void _UPCRTS_PutBlockSync(_UPCRTS_context_tag t)
{
  Tag tagg = (int*)t;

  if (t==SHORTCUTTAG)
    return;

  while(isPutTagOutstanding(tagg))
```

```
    /* wait */;
  deletePutTag(tagg);
  return;
}
/*****************************************************************************/
TimesDiff* getTimesDiff(long block_size)
{
  TimesDiff* temp;
  long diff  = block_size%BLOCK_SIZE;
  int  times = (block_size - diff)/BLOCK_SIZE;
  temp = (TimesDiff*)malloc(sizeof(TimesDiff));
  temp->times=times;
  temp->diff=diff;

  return temp;
}
/*****************************************************************************/
int getOtherThreadId(int mythread)
{
  if (mythread==0)
    return 1;
  else return 0;
}

void writeDataLocalMemory(Address adr,unsigned long datum,size_t size)
{
  int  procId;
  local_pointer lp;
  Tag tagg;

  procId = adr.thread;
  lp     = (unsigned char *)adr.va;


  /* to account for local put ***************************************/
  if(MYTHREAD==procId)
    {
      switch((int)size)
               {
               case 1:
                 *((unsigned char *)lp) = *((unsigned char *)&datum);
                 break;
               case 2:
                 *((unsigned short *)lp) = *((unsigned short *)&datum);
                 break;
               case 4:
                 *((unsigned int *)lp) = *((unsigned int *)&datum);
                 break;
               case 8:
                 *((unsigned long *)lp) = *((unsigned long *)&datum);
                 break;
                 /*
                    case PUT_FLOAT_REQUEST:
                    *((float *)lp) = *((float *)&datum);
                    break;
                    case PUT_DOUBLE_REQUEST:
                    *((double *)lp) = *((double *)&datum);
                    break;
                    */
               default:
                 printf("error1::put.c::size is unknown\n");
                 exit(-1);
               }
      return;
    }}
```

A.3 Barrier Module

```
#include "upc.h"

void _UPCRTS_Barrier(int value)
{

  static int barrIndex = -1;/* static barrIndex for barArray */
  int status;
  int barCount, barValue, barError;

  barrIndex++;
  /* before I acquire a lock let's finish all the communications */

  _UPCRTS_GetAllSync();
  _UPCRTS_PutAllSync();

  /* acquire lock */

   status = imc_lkacquire(lock_id_barrier,0,0,IMC_LOCKWAIT);

  if(status!=IMC_SUCCESS)
    {
      imc_perror("imc_lkacquire5::",status);
      exit(status);
    }

  barCount = global_space_rx->barArray[barrIndex].barCount;
  barValue = global_space_rx->barArray[barrIndex].barValue;
  barError = global_space_rx->barArray[barrIndex].barError;

  if(barError==true)
    {
      printf("\n barrier.c::exiting ... due to barrier error\n");
      exit(-1);
    }

  if ((barCount==0)&&(barValue==BARVALUE))
    {
      /* set barValue */
      do
               {global_space_tx->barArray[barrIndex].barValue = value;}
      while(global_space_rx->barArray[barrIndex].barValue!= value);

      /* set barCount */
      do
               {global_space_tx->barArray[barrIndex].barCount = 1;}
      while(global_space_rx->barArray[barrIndex].barCount!= 1);
    }


   if( ((barCount!=0)&&(barValue==BARVALUE))||
     ((barCount==0)&&(barValue!=BARVALUE)))
    {
      printf("\n error1:barrier.c\n");
      /* set barError */
      do
               {global_space_tx->barArray[barrIndex].barError = true;}
      while(global_space_rx->barArray[barrIndex].barError!=true);

      exit(-1);
    }
```

```
  if ((barCount!=0)&&(barValue!=BARVALUE))
    {
      /* make sure that barValue corresponds to value */
      if (barValue!=value)
                {
                  printf("\n barrier.c::exiting ...barValue!=value\n");
                  /* set barError */
                  do
                    {global_space_tx->barArray[barrIndex].barError = true;}
                  while(global_space_rx->barArray[barrIndex].barError!=true);
                  exit(-1);
                }
      else
                {
                  barCount++;
                  do
                    {global_space_tx->barArray[barrIndex].barCount = barCount;}
                  while(global_space_rx->barArray[barrIndex].barCount!=barCount);
                }
    }
  imc_lkrelease(lock_id_barrier,0);

  /* now let's wait till everything is syncronized */
  while (global_space_rx->barArray[barrIndex].barCount!=NUMBER_OF_PROCS)
    {

      /* check for barError */
      if (global_space_rx->barArray[barrIndex].barError==true)
                {
                  printf("\n:1: barrier.c::exiting....due to the error flag\n");
                  exit(-1);
                }
    }

}
```

A.4 Service Module

```
#include "upc.h"
#include "lock_api.h"
#include "buff_api.h"


/*****************************************************************************/
void service(int flag, int procId_tosend)
{
  /* Effects: should be called before making any get/put request
   *           procId_tosend is the proc to deposit a new message
   */

  int status,count,test;
  int slots_emptied;
  int next_to_fill, next_to_take;
  int queNum, index,count_rd,count_nord;

  /* 1.check whether service_thread is running */

  status = getServiceLockNoWait();
  if (status == -1)
    return;




  /* 2.process nord queues  */

  for (queNum = 0;queNum<NUMBER_OF_PROCS;queNum++)
    {
      if (getCount_nord(MYTHREAD, queNum)==0)
               continue;

      if ((queNum==procId_tosend)&&(flag==FUNC))
               getLock_nord(MYTHREAD,queNum,1);
      else
               if(getLockNoWait_nord(MYTHREAD,queNum,2)==-1)
                   continue;
      /* otherwise we got the lock */
      count = getCount_nord(MYTHREAD,queNum);

      for (index=0;index<count;index++)
                processElement_nord(queNum,index);
      resetBuffer_nord(queNum);
      releaseLock_nord(MYTHREAD,queNum);
    }


  /* 3.process rd queues  */
  for (queNum = 0;queNum<NUMBER_OF_PROCS;queNum++)
    {
      if (queNum==MYTHREAD)
               continue;

      if (getCount_rd(MYTHREAD, queNum)==0)
               continue;
      if(getLockNoWait_nord(queNum,MYTHREAD,3)==-1)
               continue;

      /* otherwise we got the lock */
      getLock_rd(MYTHREAD,queNum,4);
```

```
    count_rd = getCount_rd(MYTHREAD,queNum);
    count_nord = BUFFER_SIZE - getCount_nord(queNum,MYTHREAD);
    count = (count_nord>count_rd)?count_rd:count_nord;
    if (count == 0)
             {
               releaseLock_rd(MYTHREAD,queNum);
               releaseLock_nord(queNum,MYTHREAD);
               releaseServiceLock();
               return;
             }

    slots_emptied = 0;
    next_to_fill = getNextToFill_rd(queNum);
    next_to_take = getNextToTake_rd(queNum);
             if(next_to_take<=(next_to_fill-1))
             {
               /* 1. [from next_to_take to (next_to_fill-1)] */
               for (index=next_to_take;index<next_to_fill;index++)
                 if(processElement_rd(queNum,index)!=-1)
                   slots_emptied++;
             }

    if(next_to_take>(next_to_fill-1))
             {
               /* 1. [next_to_take to (BUFFER_SIZE-1)] */
               for (index=next_to_take;index<BUFFER_SIZE;index++)
                 if(processElement_rd(queNum,index)!=-1)
                   slots_emptied++;

               /* 2. [0 to (next_to_fill-1) */
               for (index=0;index<next_to_fill;index++)
                 if(processElement_rd(queNum,index)!=-1)
                   slots_emptied++;
             }
    deleteSlotsBuffer_rd(queNum,slots_emptied);

    releaseLock_rd(MYTHREAD,queNum);
    releaseLock_nord(queNum,MYTHREAD);
  }
releaseServiceLock();}
```

A.5 ProcessElement Module

```
#include "upc.h"
#include "buff_api.h"

/* function prototypes */
void setData(tag tg,unsigned long datum,ValueType type,int idfrom);
void mem_addId(Address adr,int idfrom);
cache_setState(Address adr,CacheState Clean);


/*****************************************************************************/
int processElement_nord(int queNum, int index)
{
  ValueType vltype;
  Address adr;
  ProtocolType ptype;
  CacheState state;

  element* el=&(global_space_rx->queues[MYTHREAD][queNum].nord.elements[index]);

  switch (el->type) {

  case PURGE_RESPOND:
    getPthreadLock();
    setStatusArrived_nopthreadlock(putTagBuffer,el->tagg);
    deletePutTag_nopthreadlock(el->tagg);
    adr.va=el->address_to;
    ptype=el->ptype;
    releasePthreadLock();
    adr.thread=MYTHREAD;
    state=((ptype==CleanB)? CleanB : CleanW);
    cache_setState(adr,state);
    break;

  case UPDATE_RESPOND:
    getPthreadLock();
    setStatusArrived_nopthreadlock(putTagBuffer,el->tagg);
    deletePutTag_nopthreadlock(el->tagg);
    adr.va=el->address_to;
    ptype=el->ptype;
    releasePthreadLock();
    adr.thread=MYTHREAD;
    state=((ptype==CleanB)? CleanB : CleanW);
    cache_setState(adr,state);
    break;

  case PURGE_MEM_GET_RESPOND:
    other_thread=getOtherThreadId(MYTHREAD);
    adr.thread=other_thread;
    adr.va=(unsigned long)el->address_from;
    cache_setState(adr,CacheB);
    setGetDatum(el->tagg,el->datum);
    setStatusArrived(getTagBuffer,el->tagg);
    break;

  case PUT_ONE_BYTE_RESPOND:
  case PUT_TWO_BYTE_RESPOND:
  case PUT_FOUR_BYTE_RESPOND:
  case PUT_EIGHT_BYTE_RESPOND:
  case PUT_FLOAT_RESPOND:
  case PUT_DOUBLE_RESPOND:
```

```
    getPthreadLock();
    setStatusArrived_nopthreadlock(putTagBuffer,el->tagg);
    deletePutTag_nopthreadlock(el->tagg);
    /* Wb acknoweldement - need to update cache*/
    adr.va=(char *) el->address_to;
    ptype=el->ptype;
    releasePthreadLock();
    adr.thread =queNum;
    state=((ptype==CleanB) ? CleanB : CleanW);
    cache_setState(adr,state);
    break;

  case PUT_BLOCK_RESPOND:
    setStatusArrived(putTagBuffer,el->tagg);
    break;

  case GET_ONE_BYTE_RESPOND:
    setData(el->tagg,el->datum,OneByte,queNum);
    break;

  case GET_TWO_BYTE_RESPOND:
    setData(el->tagg,el->datum,TwoByte,queNum);
    break;

  case GET_FOUR_BYTE_RESPOND:
    setData(el->tagg,el->datum,FourByte,queNum);
    break;

  case GET_EIGHT_BYTE_RESPOND:
    setData(el->tagg,el->datum,EightByte,queNum);
    break;

  case GET_TEMP_BLOCK_RESPOND:
    getBlock_nord(queNum,index);
    break;

  case GET_BLOCK_RESPOND:
    getBlock_nord(queNum,index);
    setStatusArrived(getTagBuffer,el->tagg);
    break;
  }
}
/*****************************************************************************/

void setData(tag tg,unsigned long datum,ValueType type,int idfrom)
{
  Address adr;
  int status;
  CacheState state;

  /* update cache */
  adr = shaddress_loadl;
  cache_setValue(adr,datum,type);

  state=((sh_ptype==Base) ? CleanB:CleanW);
  cache_setState(adr,state);

  /* the regular operation */
  setGetDatum(tg,datum);
  setStatusArrived(getTagBuffer,tg);
}
#include "upc.h"
#include "buff_api.h"
#include "cache.h"
```

```
/* function prototypes*/
TimesDiff *getTimesDiff(long size);
void updateMemAndCache(local_pointer, ProtocolType, unsigned long);
void updateMem_get(local_pointer address_from)

/*****************************************************************************/
int processElement_rd(int queNum, int index)
{
  /* Effects: returns -1 in case there were not enough slots in nord to fit
   *          the block message
   */
  /* NOTE: All locks needed should already be received by that point */

  int times;
  unsigned long datum;
  TimesDiff    *temp;
  local_pointer address_from,address_to,address_from_ptr,address_to_ptr;
  Tag tagg;
  int slotsNum, i, offset;
  long size,diff;
  unsigned long temp_datum;
  CacheRecord *rec;
  Address adr;
  int other_thread;
  ProtocolType ptype;

  element* el=&(global_space_rx->queues[MYTHREAD][queNum].rd.elements[index]);

  que*     rx =&(global_space_rx->queues[queNum][MYTHREAD].rd);
  que*     tx =&(global_space_tx->queues[queNum][MYTHREAD].rd);

  slotsNum = BUFFER_SIZE - getCount_nord(queNum,MYTHREAD);
  if (slotsNum==0)
    return -1;

  switch (el->type) {

  case PURGE_REQUEST:
    other_thread=getOtherThreadId(MYTHREAD);
    adr.thread=other_thread;
    adr.va=(unsigned long)el->address_to;
    if(el->ptype==Base)
      cache_deleteRecord(adr);
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum, MYTHREAD,PURGE_RESPOND);
    setPtype_nord(queNum, MYTHREAD,el->ptype);
    setAddressTo_nord(queNum, MYTHREAD,el->address_to);
    setTag_nord(queNum, MYTHREAD,el->tagg);
    return 0;

  case UPDATE_REQUEST:
    other_thread=getOtherThreadId(MYTHREAD);
    adr.thread=other_thread;
    adr.va=(unsigned long)el->address_to;
    cache_setValue(adr,datum,FourByte);/* ValueType does not matter */
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum, MYTHREAD,UPDATE_RESPOND);
    setPtype_nord(queNum, MYTHREAD,el->ptype);
    setAddressTo_nord(queNum, MYTHREAD,el->address_to);
    setTag_nord(queNum, MYTHREAD,el->tagg);
    return 0;


  case PURGE_MEM_GET_REQUEST:
    other_thread=getOtherThreadId(MYTHREAD);
```

```
  adr.thread=MYTHREAD;
  adr.va=(unsigned long)el->address_from;
  mem_deleteId(adr,other_thread);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD,PURGE_MEM_GET_RESPOND);
  setAddressFrom_nord(queNum, MYTHREAD,el->address_from);
  setDatum_nord(queNum, MYTHREAD,el->datum);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  return 0;

case GET_ONE_BYTE_REQUEST:
  /* update memory structure */
  if(el->ptype==Writer)
    updateMem_get(el->address_from);

  address_from = el->address_from;
  datum = *((unsigned char *)address_from);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, GET_ONE_BYTE_RESPOND);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  setDatum_nord(queNum,MYTHREAD,datum);
  return 0;

case GET_TWO_BYTE_REQUEST:
 /* update memory structure */
  if(el->ptype==Writer)
    updateMem_get(el->address_from);

  address_from = el->address_from;
  datum = *((unsigned short *)address_from);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, GET_TWO_BYTE_RESPOND);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  setDatum_nord(queNum,MYTHREAD,datum);
  return 0;

case GET_FOUR_BYTE_REQUEST:
 /* update memory structure */
  if(el->ptype==Writer)
    updateMem_get(el->address_from);

  address_from = el->address_from;
  datum = *((unsigned int *)address_from);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, GET_FOUR_BYTE_RESPOND);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  setDatum_nord(queNum,MYTHREAD,datum);
  return 0;

case GET_EIGHT_BYTE_REQUEST:
  /* update memory structure */
  if(el->ptype==Writer)
    updateMem_get(el->address_from);

  address_from = el->address_from;
  datum = *((unsigned long *)address_from);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, GET_EIGHT_BYTE_RESPOND);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  setDatum_nord(queNum,MYTHREAD,datum);
  return 0;

case GET_BLOCK_REQUEST:
  size = el->block_size;
  temp = getTimesDiff(size);
```

```
address_to  = el->address_to;
address_from= el->address_from;
diff        = temp->diff;
times       = temp->times;
tagg        = el->tagg;

if ( (slotsNum>times)||
           ((slotsNum==times)&&(diff=0)))
  {
           for (i=0;i<times;i++)
             {
               assert(addElement_nord(queNum,MYTHREAD)!=-1);
               setType_nord(queNum, MYTHREAD,GET_TEMP_BLOCK_RESPOND);
               offset =  i*BLOCK_SIZE;
               address_to_ptr   = address_to   + offset;
               address_from_ptr = address_from + offset;
               setAddressTo_nord(queNum,MYTHREAD,address_to_ptr);
               setBlockSize_nord(queNum,MYTHREAD,BLOCK_SIZE);
               setBlock_nord(queNum,MYTHREAD,address_from_ptr,BLOCK_SIZE);
             }
           if (diff>0)
             {
               assert(addElement_nord(queNum,MYTHREAD)!=-1);
               offset =  times*BLOCK_SIZE;
               address_to_ptr = address_to     + offset;
               address_from_ptr = address_from + offset;
               setAddressTo_nord(queNum,MYTHREAD,address_to_ptr);
               setBlockSize_nord(queNum,MYTHREAD,diff);
               setBlock_nord(queNum,MYTHREAD,address_from_ptr,diff);
             }
           setType_nord(queNum, MYTHREAD,GET_BLOCK_RESPOND);
           setTag_nord(queNum,MYTHREAD,tagg);
           return 0;
  }

/* i.e slots =times and diff>0 */
if (slotsNum==times)
  {
           for (i=0;i<times;i++)
             {
               assert(addElement_nord(queNum,MYTHREAD)!=-1);
               setType_nord(queNum, MYTHREAD,GET_TEMP_BLOCK_RESPOND);
               offset =  i*BLOCK_SIZE;
               address_to_ptr = address_to     + offset;
               address_from_ptr = address_from + offset;
               setAddressTo_nord(queNum,MYTHREAD,address_to_ptr);
               setBlockSize_nord(queNum,MYTHREAD,BLOCK_SIZE);
               setBlock_nord(queNum,MYTHREAD,address_from_ptr,BLOCK_SIZE);
             }
           /* update the "index" element in rd */
           offset=times*BLOCK_SIZE;
           address_from_ptr = address_from + offset;
           address_to_ptr   = address_to   + offset;
           setAddressFromIndex_rd(MYTHREAD,queNum,index,address_from_ptr);
           setAddressToIndex_rd(MYTHREAD,queNum,index,address_to_ptr);
           setBlockSizeIndex_rd(MYTHREAD,queNum,index,diff);
           return -1;
  }

if (slotsNum<times)
  {
           for (i=0;i<slotsNum;i++)
             {
               assert(addElement_nord(queNum,MYTHREAD)!=-1);
               setType_nord(queNum, MYTHREAD,GET_TEMP_BLOCK_RESPOND);
```

```
                offset =  i*BLOCK_SIZE;
                address_to_ptr = address_to     + offset;
                address_from_ptr = address_from + offset;
                setAddressTo_nord(queNum,MYTHREAD,address_to_ptr);
                setBlockSize_nord(queNum,MYTHREAD,BLOCK_SIZE);
                setBlock_nord(queNum,MYTHREAD,address_from_ptr,BLOCK_SIZE);
              }
            /* update the "index" element in rd */
            offset=slotsNum*BLOCK_SIZE;
            address_from_ptr = address_from + offset;
            address_to_ptr   = address_to   + offset;
            setAddressFromIndex_rd(MYTHREAD,queNum,index,address_from_ptr);
            setAddressToIndex_rd(MYTHREAD,queNum,index,address_to_ptr);
            size = size - offset;
            setBlockSizeIndex_rd(MYTHREAD,queNum,index,size);
            return -1;
    }


case PUT_ONE_BYTE_REQUEST:

  /* update memory and cache structures */
  ptype=el->ptype;
  updateMemAndCache(el->address_to,ptype,el->datum);

  address_to = el->address_to;
  temp_datum = el->datum;
  *((unsigned char *)address_to) = *((unsigned char *)&temp_datum);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, PUT_ONE_BYTE_RESPOND);
  setPtype_nord(queNum, MYTHREAD,ptype);
  setAddressTo_nord(queNum, MYTHREAD,address_to);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  return 0;

case PUT_TWO_BYTE_REQUEST:
  /* update memory and cache structures */
  ptype=el->ptype;
  updateMemAndCache(el->address_to,ptype,el->datum);

  address_to  = el->address_to;
  temp_datum = el->datum;
  *((unsigned short *)address_to) = *((unsigned short *)&temp_datum);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, PUT_TWO_BYTE_RESPOND);
  setPtype_nord(queNum, MYTHREAD,ptype);
  setAddressTo_nord(queNum, MYTHREAD,address_to);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  return 0;

case PUT_FOUR_BYTE_REQUEST:
  /* update memory and cache structures */
  ptype=el->ptype;
  updateMemAndCache(el->address_to,ptype,el->datum);

  address_to  = el->address_to;
  temp_datum = el->datum;
  *((unsigned int *)address_to) = *((unsigned int *)&temp_datum);
  assert(addElement_nord(queNum,MYTHREAD)!=-1);
  setType_nord(queNum, MYTHREAD, PUT_FOUR_BYTE_RESPOND);
  setPtype_nord(queNum, MYTHREAD,ptype);
  setAddressTo_nord(queNum, MYTHREAD,address_to);
  setTag_nord(queNum, MYTHREAD,el->tagg);
  return 0;
```

```
  case PUT_EIGHT_BYTE_REQUEST:
    /* update memory and cache structures */
    ptype=el->ptype;
    updateMemAndCache(el->address_to,ptype,el->datum);

    address_to  = el->address_to;
    temp_datum = el->datum;
    *((unsigned long *)address_to) = *((unsigned long *)&temp_datum);
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum, MYTHREAD, PUT_EIGHT_BYTE_RESPOND);
    setPtype_nord(queNum, MYTHREAD,ptype);
    setAddressTo_nord(queNum, MYTHREAD,address_to);
    setTag_nord(queNum, MYTHREAD,el->tagg);
    return 0;

  case PUT_FLOAT_REQUEST:
    /* update memory and cache structures */
    ptype=el->ptype;
    updateMemAndCache(el->address_to,ptype,el->datum);

    address_to  = el->address_to;
    temp_datum = el->datum;
    *((float *)address_to) = *((float *)&temp_datum);
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum, MYTHREAD, PUT_FLOAT_RESPOND);
    setPtype_nord(queNum, MYTHREAD,ptype);
    setAddressTo_nord(queNum, MYTHREAD,address_to);
    setTag_nord(queNum, MYTHREAD,el->tagg);
    return 0;

  case PUT_DOUBLE_REQUEST:
    /* update memory and cache structures */
    ptype=el->ptype;
    updateMemAndCache(el->address_to,ptype,el->datum);

    address_to  = el->address_to;
    temp_datum = el->datum;
    *((double *)address_to) = *((double *)&temp_datum);
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum, MYTHREAD, PUT_DOUBLE_RESPOND);
    setPtype_nord(queNum, MYTHREAD,ptype);
    setAddressTo_nord(queNum, MYTHREAD,address_to);
    setTag_nord(queNum, MYTHREAD,el->tagg);
    return 0;

  case PUT_TEMP_BLOCK_REQUEST:
    getBlock_rd(queNum,index);
    return 0;


  case PUT_BLOCK_REQUEST:
    getBlock_rd(queNum,index);
    tagg=el->tagg;
    assert(addElement_nord(queNum,MYTHREAD)!=-1);
    setType_nord(queNum,MYTHREAD, PUT_BLOCK_RESPOND);
    setTag_nord(queNum, MYTHREAD,tagg);
    return 0;
  }
}
/*****************************************************************************/
void updateMemAndCache(local_pointer address_to,ProtocolType ptype,
                                    unsigned long datum)
{
  CacheRecord* rec;
  Address adr;
```

```
  int otherThread;

  adr.thread=MYTHREAD;
  adr.va=(unsigned long)address_to;
  otherThread=getOtherThreadId(MYTHREAD);

  if(ptype==Writer)
    {
      mem_addId(adr,otherThread);
      if(mem_isIdIn(adr,MYTHREAD))
                {
                  /* update cache */
                  cache_setValue(adr,datum,FourByte);/* ValueType does not matter */
                  return;
                }
      return;
    }

  /* 1.update memory */

  mem_deleteId(adr,MYTHREAD);
  mem_deleteId(adr,otherThread);

  /* 2.update cache */
  rec=cache_findRecord(adr);
  if(rec!=NULL)
    cache_deleteRecord(adr);
}

void updateMem_get(local_pointer address_from)
{
  Address adr;
  int other_thread;
  adr.thread=MYTHREAD;
  adr.va=(unsigned long)address_from;
  other_thread=getOtherThreadId(MYTHREAD);
  mem_addId(adr,other_thread);
}
```

A.6 UPC Header File

```
/*
 * upc.h
 */
/*****************************************************************************/
/*** include, define *********************************************************/

#include "stddef.h"
#include <sys/imc.h>
#include <sys/types.h>
#include <c_asm.h>
#include <assert.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include "/home/cs/morganb/work/tagBuffer/tagBuff_api.h"
#define GLOBAL_SPACE_SIZE 200000
#define NUMBER_OF_PROCS 2   /* number of nodes                          */
#define BUFFER_SIZE     30  /* 30 length of circular buffer of each node  */
#define EMPTY           0   /* indicates that circular buffer is empty */
#define BLOCK_SIZE      100 /* bytes - size of block to use in splitting th
                                        * transfer block */

#define BARVALUE -999      /* default value of barValue flag */
#define MAX_NUMBER_OF_BARRIERS 100 /* max number of barriers in the program */

enum ProtocolType {Writer,Base};
typedef enum ProtocolType ProtocolType;

/*****************************************************************************/
/** element  que *************************************************************/
typedef  struct {
  int type;
  ProtocolType p type;
  unsigned long datum;
  tag tagg;
  unsigned char* address_to;  /* address to where put the transfered data     */
  unsigned char* address_from;/* adrress from which get the data to transfer  */
  long   block_size;
  unsigned char block[BLOCK_SIZE];
}element;

struct  que{
  element  elements[BUFFER_SIZE];
  int      next_to_fill;
  int      next_to_take; /* is used only for rd que (as circl buff) */
  volatile int count;
} ;
typedef struct que que;
/*****************************************************************************/
/*** procQue  ****************************************************************/
typedef struct {
  que     nord;  /* each element does not require respond */
  que     rd;    /* each element of it requires respond */
  boolean serviceFlag; /* TRUE if service proc is on, and FALSE if off */
} procQue;
/*****************************************************************************/
/*** barrStruct   ************************************************************/
typedef struct {
  int barCount; /* a flag for barrier syncronization  */
  int barValue; /* a flag for barrier syncronization  */
  int barError; /* a flag for barrier error  */
```

```
}barrStruct;
/*****************************************************************************/

/*** global space ************************************************************/

typedef struct {
  procQue queues[NUMBER_OF_PROCS][NUMBER_OF_PROCS];
  volatile int syncCount; /* a flag for syncronization start */
  barrStruct  barArray[MAX_NUMBER_OF_BARRIERS];
} global_space;

global_space* global_space_tx;
global_space* global_space_rx;


/*****************************************************************************/
/*****************************************************************************/
/*
 * init Lock required for syncroniztion and initialization procedure
 */

int initLock;
imc_lkid_t initLock_id;
/*****************************************************************************/
/*
 * lock global variables
 */

imc_lkid_t lock_id_rd;
int locks_rd;

imc_lkid_t lock_id_nord;
int locks_nord;


imc_lkid_t lock_id_barrier;
int lock_barrier;


pthread_mutex_t lock_service_mutex;
pthread_mutex_t lock_cache_mutex;
pthread_mutex_t lock_mem_mutex;
/*****************************************************************************/
/* pointer */

typedef unsigned char* local_pointer;

typedef struct {
  local_pointer l_ptr;
  int procId;
} pointer;
/*****************************************************************************/

/*****************************************************************************/
/* putGet */

typedef struct {
  int times;
  long diff;
} TimesDiff;
/*****************************************************************************/


int MYTHREAD;
```

```
#define FUNC   300
#define THREAD 301
/*****************************************************************************/
 /* MISL definitions */
#define TEST printf("\n test \n");
#define HOME /home/cs/morganb/work
/*****************************************************************************/

/* Put Requests */
#define PUT_ONE_BYTE_REQUEST    10
#define PUT_TWO_BYTE_REQUEST    15
#define PUT_FOUR_BYTE_REQUEST   20
#define PUT_EIGHT_BYTE_REQUEST  25
#define PUT_FLOAT_REQUEST       30
#define PUT_DOUBLE_REQUEST      35
#define PUT_TEMP_BLOCK_REQUEST  40
#define PUT_BLOCK_REQUEST       45

/* Get Requests */
#define GET_ONE_BYTE_REQUEST    50
#define GET_TWO_BYTE_REQUEST    55
#define GET_FOUR_BYTE_REQUEST   60
#define GET_EIGHT_BYTE_REQUEST  65
#define GET_BLOCK_REQUEST       70

/* Put Responds */
#define PUT_ONE_BYTE_RESPOND    75
#define PUT_TWO_BYTE_RESPOND    80
#define PUT_FOUR_BYTE_RESPOND   85
#define PUT_EIGHT_BYTE_RESPOND  90
#define PUT_FLOAT_RESPOND       95
#define PUT_DOUBLE_RESPOND      100
#define PUT_BLOCK_RESPOND       105

/* Get Responds */
#define GET_ONE_BYTE_RESPOND    110
#define GET_TWO_BYTE_RESPOND    115
#define GET_FOUR_BYTE_RESPOND   120
#define GET_EIGHT_BYTE_RESPOND  125
#define GET_TEMP_BLOCK_RESPOND  130
#define GET_BLOCK_RESPOND       135

/* Purge */
#define PURGE_REQUEST           140
#define PURGE_RESPOND           150
#define PURGE_MEM_GET_REQUEST   160
#define PURGE_MEM_GET_RESPOND   170
#define UPDATE_REQUEST          180
#define UPDATE_RESPOND          190

struct _UPCRTS_shared_pointer
{ unsigned long va: 43;
  unsigned int phase: 10;
  unsigned int thread: 11;
};

typedef struct _UPCRTS_shared_pointer _UPCRTS_SHARED_POINTER_TYPE;
typedef unsigned long _UPCRTS_context_tag;

const int _UPCRTS_gl_cur_vpid;
const int _UPCRTS_gl_cur_nvp;
int _UPCRTS_gl_forall_depth;

/* added later */
unsigned long shortcut_value; /* global variable to transfer the value of
```

```
                                                        datum in shortcut case when
                                                        MYTHREAD==procId */
/**Tags   ***************************************************************/
#define SHORTCUTTAG -1000

/* shortcuts */
typedef _UPCRTS_context_tag UPCTag;
typedef _UPCRTS_SHARED_POINTER_TYPE Address;

/** Tags: shortcuts for loadl ***************************************/
#define SHTAG_LOADL -2000  /* supposed to mean shortcut tag for loadl   */
unsigned long shvalue_loadl;
Address  shaddress_loadl;
ProtocolType sh_ptype;
/*******************************************************************/
/* Cache and Mem declarations */

enum ValueType {OneByte,TwoByte,FourByte,EightByte};
typedef enum ValueType ValueType;
typedef unsigned long Value;
enum CacheState {CleanW,CleanB,Dirty,WbPending,CachePending,Free}; /* Free state means

* that cache record i

*  not filled

*/
typedef enum CacheState CacheState;

enum AccessType {Load,Store}
typedef enum AccessType AccessType;
```

Bibliography

[1] Active Messages: a Mechanism for Integrated Communication and Computation. von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauser. In P*roceedings of the 19th Int'l Symp. on Computer Architecture,* May 1992, Gold Coast, Australia.

[2] Parallel Programming in Split-C, D. Culler, A.Dusseau, S. C. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick. In *Proceedings of Supercomputing'93,* November 1993.

[3] MEMORY CHANNEL Application Programming Interfaces. *http://www.unix.digital.com/faqs/publications/cluster_doc/cluster_15/PS_MCAPI/TITLE.HTM*

[4] X.Shen, Arvind, and L.Rudolph CACHET:An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems.

[5] X.Shen, Arvind, and L.Rudolph. Commit-Reconcile & Fences (CRF): A New Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, May 1999.

[6] J.K Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[7] S.Eggers and R.H. Katz. Evaluating the Performance for Four Snooping Cache Coherency Protocols. In P*roceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989.

[8] B.Falsani, A.R. Lebeck, S.K.Reinhardt, I.Schoinas, M.D. Hill Application-specific protocols for user-level shared memory. In *Supercomputing*, Nov 1994.