The C-LEMMA Memory Interface on the Cray T3D

Christopher D. Walton and Bruce J. McAdam
Department of Computer Science
University of Edinburgh

Abstract

The challenges associated with the implementation of a concurrent programming language on a parallel machine are often independent of the language itself, such as memory-management. This report is a detailed description of a language-independent memory-platform for the Cray T3D supercomputer, called the C-LEMMA interface. The interface is closely based on the definition of the LEMMA interface used in the DP/ML (Distributed Poly/ML) compiler. However, the actual implementation is quite different, due to some severe limitations imposed by the Cray architecture. The memory-platform provides a distributed shared virtual memory (i.e. a single shared virtual address space distributed across processors) with global garbage-collection.

This report is a detailed description of the implementation of the C-LEMMA interface. It is also intended as a reference manual for those who wish to implement the run-time system of a concurrent programming language using the interface.

1. Introduction

The C-LEMMA interface, described in this report, is intended to provide the first stage of an ongoing project to implement a compiler for the Standard ML programming language [Miln90] on the Cray T3D Supercomputer. The practical advantages of such a compiler are twofold. On the one hand, it would allow programmers already using the Cray T3D to program in a language capable of expression a wide range of complex problems. On the other hand, it would enable programmers already familiar with the Standard ML language to considerably increase the performance of existing programs and/or construct larger programs with greater resource requirements. The Cray T3D implementation of Standard ML is intended to be closely based on the successful Distributed Poly/ML (DP/ML) compiler for networked clusters of workstations [Matt91]. While this compiler already provides a significant performance increase, the Cray's 30Gflop/peak performance and large address space would potentially provide a huge performance boost for a range of applications.

At its lowest level, the DP/ML compiler is based around a memory-management interface called LEMMA [Matt95a], and [Matt95b]. A description of an implementation of this interface for the Cray T3D forms the body of this report. Although the LEMMA interface contains a number of optimisations for the DP/ML compiler, it was designed to be a general-purpose interface capable of supporting a wide-range of concurrent programming languages. As a result, the content of this report has a far wider applicability than simply a back-end for the Standard ML compiler.

The implementation of the LEMMA interface is based around a technique known as Distributed Shared Memory (DSM) [Nitz91], and [Colo94]. DSM provides a single globally-addressable virtual address space, shared between processes executing on different processors. The advantages of DSM over conventional message-passing libraries (e.g. MPI [Mpif94]) include ease of programmability and portability, achieved through the use of the shared-memory programming paradigm. At the present time, the C-LEMMA interface is the only DSM system for the Cray T3D. In addition to a distributed shared memory, LEMMA also provides a global garbage-collection facility for the shared address space.

The following report is a detailed description of the implementation of the LEMMA interface on the Cray T3D, together with an explanation of the interface for those who may wish to use it to implement a concurrent programming language on the Cray.

1.1 Terminology: LEMMA, C-LEMMA, W-LEMMA, and RTS

The LEMMA interface, as described in [Matt95a], does not make any assumptions regarding implementation, thereby leaving open the possibility of alternative implementations on different concurrency platforms with the same interface. Indeed, the previous distributed-workstation version of the LEMMA interface [Matt95b] is quite different to the one described in this report. In order to avoid confusion, the Cray T3D version of the LEMMA interface will be referred to as *C-LEMMA*, while the distributed workstation version will referred to as *W-LEMMA*. The term *LEMMA* used in isolation will refer to the LEMMA interface definition without regard to a particular implementation.

Although the LEMMA interface is designed to be language-independent in that it treats all program data as generic *objects*, there are places in the garbage-collector where is necessary to examine the data contained inside these object (e.g. to find the pointers). Consequently, the LEMMA interface relies on a small language-dependent *Run Time System* (RTS) for this information. Therefore, a different implementation of the RTS is required depending on the target language. An example RTS for DP/ML is described later in this report. It is worth noting that the interface definition treats LEMMA as a library of functions which are called by the RTS, as the RTS may be required to perform marshalling on the data objects before they are processed by LEMMA.

2. The LEMMA Interface Definition

The LEMMA memory interface [Matt95a] defines a *memory-platform* for supporting concurrent programming languages. The model of computation assumed by LEMMA is a single *client* machine, and one or more *servers*. The servers provide only computing power and maintain state only as an extension of the client. The client is responsible for initiating and terminating execution, in addition to providing facilities such as I/O.

The LEMMA interface provides most of the memory management facilities required for a concurrent programming language. The remaining language-specific memory management facilities are provided in a separate interface known as the run-time system (RTS). LEMMA and RTS are inter-related (although their boundaries are carefully defined) and rely on each other behaving correctly. The LEMMA interface effectively provides the RTS with the following three facilities:

1. Setting up communication channels between clients and servers.
2. Allowing distributed objects to be shared through virtual addressing.
3. Allocating memory without requiring explicit deallocation (garbage-collection).

In contrast to the majority of programming languages, Standard ML can make a compile-time distinction between *immutable* and *mutable* data objects. Immutable objects are those whose value is fixed from the time of allocation, while mutable objects may change their value at any time during program execution. Reference-types (`ref`) are the only mutable data objects in Standard ML. Consequently, immutable data allocation is far more common than mutable data allocation, provided a largely functional programming style is used. This assumption is used to perform several optimisations in LEMMA. Other programming languages, such as C, have only mutable data objects (variables) and will therefore require only the mutable data functions of LEMMA.

The LEMMA interface definition [Matt95a] consists of a large number of functions, enabling it to support a range concurrent languages. The possibility of alternative implementations of LEMMA with the same interface is left open. It is also possible to produce a working run-time system without implementing the full LEMMA interface.

2.1 Distributed Shared Memory and Garbage-Collection

Distributed Shared Memory (DSM) allows a process to access memory locations which reside on remote machines as if they were part of its own local address space. The DSM technique employed by LEMMA, uses the fact that a typical machine has a very large virtual address space available (e.g. 64-bit for the DEC Alpha), but uses only a small portion for its physical memory. LEMMA statically partitions the virtual address space between machines and allows each machine to allocate only within its partition. When a machine accesses an area of virtual memory outside of its partition (i.e. used by another machine) a page-fault signal is generated by the OS. This signal is trapped by LEMMA and a copy of the page is fetched from the remote machine. This situation is illustrated in the following diagram. In this case, machine 1 has attempted to access an area of memory managed by machine 2.

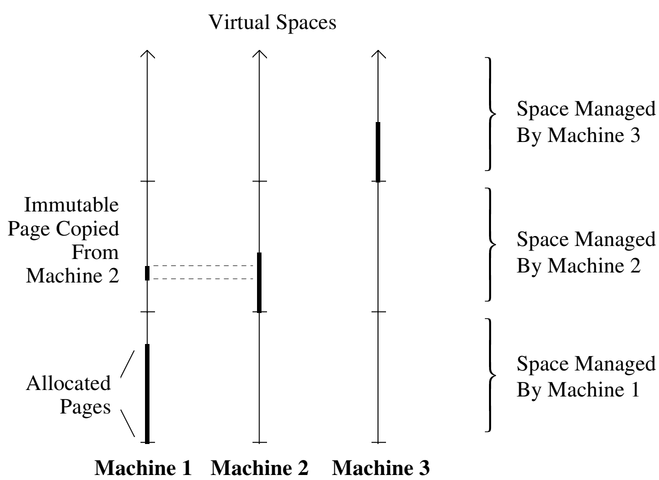


Figure 2.1 - The LEMMA Memory Model

The complete DSM technique in LEMMA involves some additional complexity. The above diagram shows memory accesses performed using a page granularity, yet an actual data object may be much larger (or indeed smaller) than a page. Also, immutable and mutable objects must be treated separately. Consequently, LEMMA allocates mutable and immutable objects in separate areas as described below.

2.1.1 Immutable Data Objects

Immutable objects are packed into pages by LEMMA. Pages containing immutable objects are never invalidated and therefore can be freely cached between machines without the need for coherency checking. A cached immutable object always appears in exactly the same virtual memory location as the original object. It is simple to discover the manager of an immutable object from its address, since the address space is statically partitioned.

2.1.2 Mutable Data Objects

Mutable objects may freely change (by definition). This can lead to two problems if we simply employ the same technique used for immutable objects:

1. If mutable data is packed into pages, the well-known *false-sharing* problem may occur as we can only invalidate complete pages (a page may contain more than one object). One possible solution, is simply to allocate one object per page, resulting in memory wastage. The solution used by the W-LEMMA and C-LEMMA implementations is to abandon the use of page protection to control access and allocate a separate header for each mutable object. This header is checked whenever the object is accessed.
2. The other problem concerns the *caching* of objects on remote machines. Clearly a coherency protocol will be required for mutable data. A number of suitable protocols are described in [Serg94].

2.1.3 Global Garbage-Collection

The LEMMA interface definition also provides garbage-collection of the shared address space via an extension of the *two-space copying* approach [Matt95b]. For reasons explained later, the C-LEMMA interface uses an alternative technique, based around a more traditional *mark and sweep* algorithm.

3. The Cray T3D Supercomputer

In order to understand the implementation of the C-LEMMA interface it is necessary to know a little about the Cray T3D architecture and its memory-management facilities. A detailed description of the Cray T3D configuration used in this report can be found in [Boot95], and an overview of the Cray T3D system architecture can be found in [Cray93]. The Cray T3D is an impressive Massively-Parallel-Processing (MPP) computer capable of providing huge (30Gflop/peak) computation rates. The T3D configuration used in the description of the C-LEMMA system is as follows:

"… the T3D array comprises 256 nodes each with 2 processing elements (PEs) for a total of 512 PEs. Each PE consists of a DEC Alpha 21064 processor running at 150MHz, supporting 64-bit integer and 64-bit IEEE floating point operations and delivering 150 64-bit Mflop/s. The peak performance of the T3D array itself is 300Gflop/s. The DEC Alpha 21064 includes an 8Kbyte direct-mapped data cache and an 8Kbyte instruction cache. Each processor has 64Mbyte of RAM, giving an aggregate memory of 32Gbytes. The nodes are arranged in a three-dimensional torus, with each of the six links from each node simultaneously supporting hardware transfer rates of up to 300Mbyte/s. Hardware support for a single address space across the array is provided …"

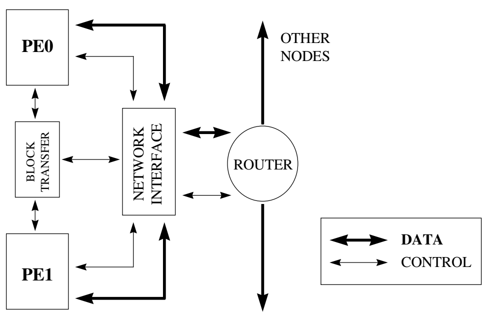


Figure 3.1 - A T3D Processing Node

3.1 Programming Model and Memory Addressing

Programming for the T3D is performed (for the purposes of C-LEMMA) using an extension of the C language [Cray94]. The T3D uses a Single-Program-Multiple-Data (SPMD) programming model which means that the same program is run on each processor simultaneously, though these programs may identify which processor they are executing on and act accordingly (i.e. multiple data).

The Cray T3D provides a global address space. In theory, it is possible to treat the entire available memory space as a single object. While this appears ideal for supporting distributed shared memory, things are not nearly as straightforward. The Cray T3D uses only 64-bit addresses. All local memory accesses are performed using only the lower 26 address bits (i.e. 64Mb). Remote accesses are controlled by bits 32-36 in the address. These bits index into a 32-entry look-up table called the DTB-annex. This table contains some processor specific information including a processor number[1]. Only the first 16 of the table entries are available to the user. This means that, in-fact, the *global* address space only covers a maximum of 16 PEs. The reason for this seemingly arbitrary limitation is the fact that the DEC Alpha 21064 processors used do not implement the full 64-address lines on the chip package.

To perform full global addressing, it is necessary to dynamically alter the PE mapping in the DTB-annex table. An optimised programming library, called SHMEM (SHared MEMory) is provided to transparently alter these table mappings. The C-language version of the SHMEM library [Barr94] provides the following two functions (among others) for accessing remote memory:

```
void shmem_get(long *target, long *source, int nlong, int pe);
void shmem_put(long *target, long *source, int nlong, int pe);
```

- `shmem_get` copies a block of memory, of length `nlong` 64-bit words, starting at address `source` on processor `pe`, to address `target` on the calling PE.
- `shmem_put` writes a block of memory, from address `source` on the calling processor, of length `nlong` 64-bit words, into address `target` on processor `pe`, without notification. This function does not wait for the operation to complete.

Although the SHMEM library may be used to perform global addressing, it is not satisfactory for DSM as a data object does not appear at the same memory address to each processor. Consequently, a form of address translation is also required.

The documentation [Boot95b] contains three heuristics to consider when designing shared memory applications:

1. Remote memory access is slower than local memory access.
2. The further apart the PEs (numerically), the slower the communication.
3. Remote read operations are slower than remote write operations.

3.2 Operating System

The T3D is hosted by Cray Y-MP and Cray J-90 computers running UNICOS-MAX (a UNIX-like OS). These machines are used as the front-end for editing and compiling T3D code as well as serving I/O requests from the T3D. Each T3D processor runs a cut-down micro-kernel of UNICOS-MAX, which forwards I/O requests to the hosts and manages global memory addressing. Unfortunately, this micro-kernel does not provide the full UNIX

[1] There is a slight complication involving logical and real PE numbers. However, only the logical PE number will be used in this report: PEs will be numbered from 0 upwards.

functionality required for the C-LEMMA interface. Similar problems hampered a previous attempt to implement DP/ML on the Meiko CS2 Transputer system [Bhoe92].

1. Although page fault signals are generated when an unallocated (or illegal) area of memory is accessed, these signals cannot be caught by an individual PE. Instead, the entire process is terminated. Signals are used in the garbage-collection (user signals) and distributed shared virtual memory (page fault signals) sections of W-LEMMA.
2. It is not possible to allocate a portion (page) of memory at a specific virtual address. Indeed, it appears that pages and virtual addresses are not even used. The only way of allocating memory is to use the heap (e.g. using `malloc` or declaring an array).
3. The system is limited to one process per processor. Although not a serious limitation, this does somewhat limit the extensibility of the eventual DP/ML implementation.

4. The Distributed Virtual Memory Emulator (DVME)

In order to overcome the limitations of the Cray T3D architecture described in the previous section, a separate library was constructed to *emulate* the distributed virtual memory functionality required by the LEMMA interface. Memory accesses *must* be passed through this emulator or they will be invalid. Clearly, trapping all memory accesses will considerably reduce the performance of system, but the primary concern at this stage of the design was functionality. Efficiency will hopefully be dealt with in a future project, possibly by re-writing some of the emulator as C-macros (thereby eliminating function-call overhead). There are four main tasks performed by the *Distributed Virtual Memory Emulator* (DVME) on behalf of the C-LEMMA interface:

1. Provision of a distributed virtual memory address space.
2. Caching of immutable page data.
3. Allocation and de-allocation of contiguous blocks of pages.
4. Communication of short messages between processors.

An added benefit of the emulator is that it gathers all of the OS and networking dependent sections of the C-LEMMA interface into one module, enhancing the overall portability of the system. In this case, the DVME is based on the Cray SHMEM library, but it could equally be implemented using UNIX and TCP/IP or even on a message-passing system such as MPI. It is worth noting that the PVM and MPI message-passing systems on the Cray are also implemented using the SHMEM library. The following sections describe the four main algorithms used in the DVME library.

4.1 A Distributed Virtual Address Space

The DVME uses a special *virtual* address format. All addresses used throughout the above layers must use this format. Only the emulator (with a few small exceptions for performance reasons) is permitted to use *real* memory addresses. The DEC Alpha processor used in the Cray has a 64-bit architecture: words are defined to be 64-bits in length and all addresses are word length. Only half of the available 64Mb of memory on each PE is used as the main data area (explained below). Therefore, only the lower 25 bits (32Mb) of the 64-bit virtual address are used for addressing purposes. The next 8 bits are used to indicate which (of a maximum 256)[2] PEs the address applies to. The remaining bits are left

[2] A single user is only permitted (on the system used in this report) to use a maximum half of the total number of processors at any time.

undefined (for future addressing needs). With a little thought, it is clear that this style of address effectively provides a contiguous virtual address space partitioned across the PEs in 32Mb segments. The address format is illustrated graphically below:

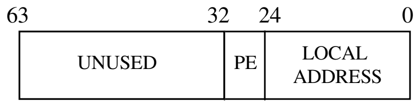


Figure 4.1 - DVME Virtual Address Format

On the Cray it is not possible to allocate a portion of memory at a specific address. However, data declared in the global region of a C program is guaranteed to be allocated at the same address on every PE (recall that the same program runs on every PE). This fact can be used to declare a large contiguous region (i.e. array) of memory on the heap with a known global starting address. Sections of this pre-allocated memory can be used, as if allocating a portion of memory at a specific address. A *page-table* is used to perform the address translations between virtual and real memory addresses. There are two such areas of memory used in the DVME:

1. A large 32Mb data area is reserved on each processor and partitioned into 8Kb pages (this page size is defined by the DEC Alpha hardware). Note that the data area also contains the user processes.
2. Another contiguous 16Mb region is used as a cache for remote immutable pages.

The remaining memory is used to hold the page tables and the remaining run-time system. A possible arrangement for the various memory regions is shown below.

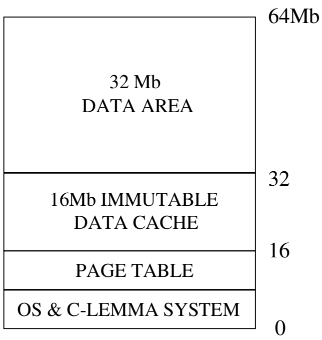


Figure 4.2 - Possible Memory Map

The page-table is used to convert virtual addresses into real addresses. A one-to-one mapping is used between the entries in the page table and the 8Kb pages in the data area. That is, the first entry in the page table corresponds to the first page in the data area, etc. This mapping simplifies address translations in the emulator. Each PE has its own separate page table relating to its local 32Mb data area (4096 entries). Each page table entry (PTE) occupies a 64-bit word. The format of these entries is shown below:

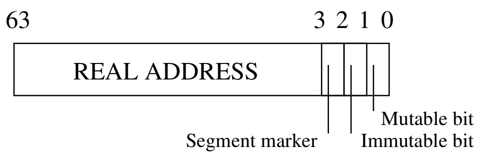


Figure 4.3 - A Page Table Entry (PTE)

- The *mutable* and *immutable* bits indicate the type of data stored in the corresponding page. If neither bit is set, the page is empty (to have both set would be invalid).
- The *segment marker* bit is used in the page allocation algorithm.
- The *real address* field is used to store a pointer to the actual memory address of the page. At this stage, this address is redundant because of the one-to-one mapping, but will be used in the page caching mechanism. The real addresses are aligned to 8Kb page boundaries which permits the usage of the lower bits for other purposes.

The following diagram illustrates the address translation process (from a virtual to real addresses) inside the emulator:

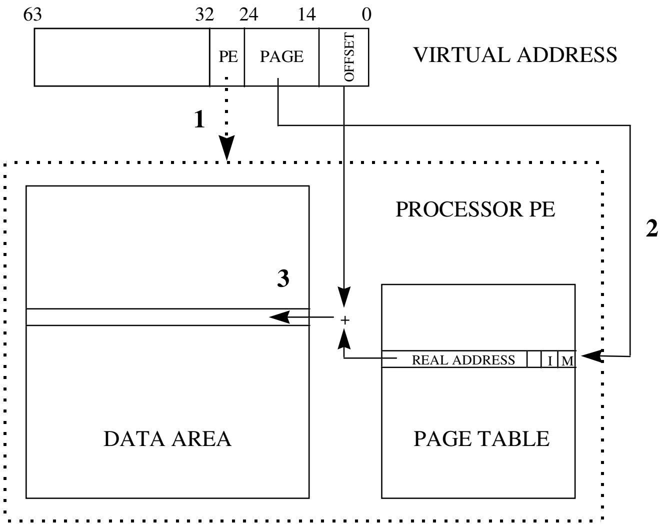


Figure 4.4 - Virtual Address Translation

1. The PE field in the virtual address is used to determine on which processor the data resides, and hence, which page table and data area to examine.
2. The upper bits of the virtual address (i.e. the *page*) are used to index into the page table on this processor. The mutable and immutable bits of the PTE are checked to ensure a valid page mapping.
3. Lastly, the lower bits of the virtual address (i.e. the *offset*) are added together with the real address in the PTE to obtain the actual address of the required data.

4.2 Immutable Data Caching

The virtual address translation mechanism described above is clearly inefficient when it comes to accessing data on a remote processor. In order to improve this performance, a form of data caching is required. The caching of mutable data requires a coherency protocol, which adds overhead and significantly increase the complexity of the system (currently atomic writes are used, which ensure consistency but remain inefficient). However, as explained earlier, immutable data can be freely cached without the need for coherency checking as it is never invalidated.

Figure 4.2 shows a 16Mb region reserved for caching immutable data. It is trivial to extend the address translation mechanism to fetch an entire page (rather than a word) when it accesses a remote immutable object. Therefore it is only necessary to design a suitable caching algorithm.

A hardware cache is often constructed out of associative memory. This allows all of the cache locations to be examined simultaneously for the presence of a particular tag. In a software scheme we do not have this luxury, neither would it be practical to examine every page in the cache (2048 entries) on every remote memory access. The solution that is implemented extends the page-table on each PE to provide an entry for every possible page location. This generates a rather large (8Mb) page-table on each PE, but ensures a minimum overhead on remote memory accesses. The new page table layout is shown below:

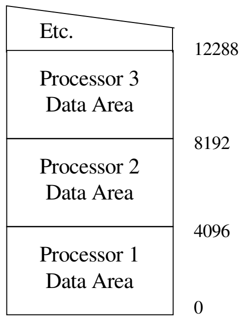


Figure 4.5 - Extended Page Table

4.2.1 Retrieval of Objects Via the Cache

The following pseudo-code shows the steps that the address translation mechanism must take when accessing a remote immutable object:

```
Check page-table on local PE for a valid mapping into the cache.
If none found then
       Fetch remote page containing the required address into the page cache.
       Update the corresponding local PTE to point to the cache.
Return the required word from the page cache.
```

Clearly, the page-table will only be sparsely populated with addresses except in the portion relating to its own data area (note that PTEs always point to valid local addresses). The size of the page-table could be reduced using hashing or set-associativity techniques, but this would adversely affect performance unless carefully implemented.

4.2.2 Insertion of Pages Into the Cache.

The remaining issues involve *page-placement* and *page-invalidation* within the cache. The commonly used hardware technique is to attempt a least-recently-used (LRU) policy. Every time a page is accessed, a *history* bit is set in the corresponding PTE. Periodically, these history bits are cleared. When the cache becomes full, the first page with its history bit clear is overwritten. While this technique looks promising, the cost of clearing the history bits without hardware assistance is too great.

Instead, a simpler *round-robin* policy is used: the cache is filled sequentially until it becomes full, then the cache is overwritten from the beginning, invalidating one page at a time, etc. This leaves one remaining complication. When a page is overwritten, the PTE pointing to the old page must be cleared. This entry will lie in a part of the page-table outside the local portion. To enable the PTE to be cleared easily, a small table of *back-pointers* is maintained for each page in the cache (2048 pages). The entries of this table contain the index of the PTE pointing at the page. The various tables used in this scheme are shown below (this example examines the tables on PE 2).

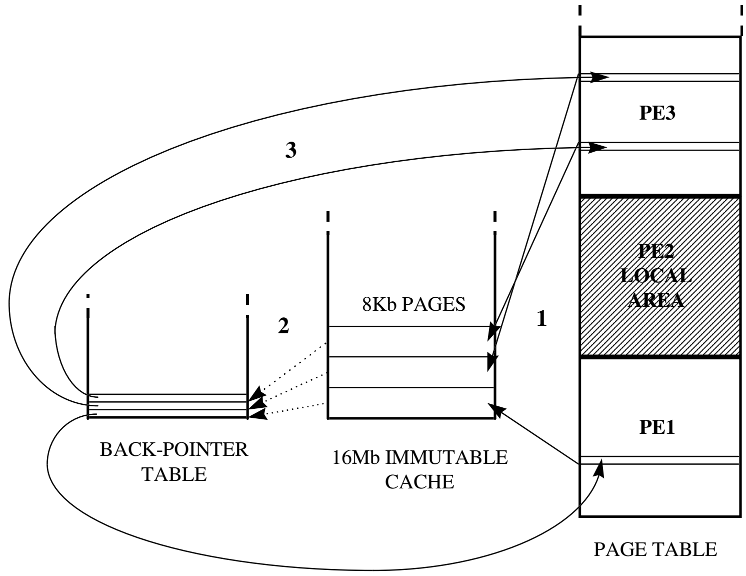


Figure 4.6 - Immutable Page Caching

1. Remote immutable pages are copied sequentially into the local cache area and the local page-table is updated.
2. Each page in the cache has an entry in the back-pointer table with its original address.
3. The entries in the back-pointer table are used to invalidate the page-table when cache is cleared.

4.3 Contiguous Page Allocation and Deallocation

The next area of concern is the allocation (and deallocation) of contiguous pages of memory in the emulator. As stated earlier, each processor is responsible for the management of memory within its own data area. Most of the time it doesn't matter at what address a page is allocated. However, there are times when data must be placed at a certain memory location. (It is reasonable to assume that these areas will be allocated first). Clearly, allocating pages of memory sequentially or in a round-robin fashion will be inadequate. A solution to the memory allocation problem must be able to satisfy the following requirements.

1. Allocation of data at a specific memory location.
2. Allocation of data at an unspecified memory location.
3. De-allocation of data at a specific memory location.

It is also highly desirable to reduce memory fragmentation as far as possible.

4.3.1 Allocation and De-Allocation Algorithm

The algorithm described below performs the required allocation and de-allocation operations, as well as attempting to minimise fragmentation. This may not be the most efficient algorithm available, but it performs the job adequately assuming the frequent allocation of short segments of memory. For the purposes of this algorithm, a *segment* is defined as a block of one or more contiguous pages.

The 32Mb data area on each processor consists of 4096 pages. These pages will be indexed sequentially from 0 to 4095. The algorithm maintains 13 ordered lists (numbered 0 to 12) used for storing the indices of unallocated segments. Only the index of the first page in the segment is stored in the list. Each of these lists stores segments that are a different (and increasing) power of two in length. For example, list 5 stores the indices of segments that are 32 (2^5) pages in length. The indices are stored in increasing order in each list.

Initially, none of the 4096 pages are allocated, therefore, list 12 ($2^{12} = 4096$) contains a single entry 0 (the index of the first page). This could equivalently be represented as two segments of 2048 pages in which case, list 11 ($2^{11} = 2048$) would contain the entries 0 and 2048 in that order. However, in practice the algorithm maintains the largest contiguous segments possible.

The allocation of a segment of memory at an arbitrary memory location proceeds as follows. (Note that allocation corresponds to the removal of items from the lists). The required segment length is rounded-up to the nearest power of two. The lists are then searched from this power of two upwards (e.g. 32 = list 5 upwards) until a non-empty list is found. If no non-empty lists are found, the allocation fails. Otherwise, the first item is removed from this list and the index returned. The remaining pages (due to the rounding-up) are then de-allocated. The following pseudo-code illustrates this allocation procedure:

```
Allocate (length) :
        Round 'length' up to nearest power of two.
        Search lists from this power upwards until a non-empty list is found.
        If a non-empty list can't be found, fail.
        De-allocate (rounded-length - starting length).
        Return first item from non-empty list.
```

De-allocation of a segment of memory is done recursively. The length of the segment to be de-allocated is rounded-down to the nearest power of two. The pages corresponding to the rounded length are removed from the segment and inserted into the list associated with this power of two. The remaining pages are repeatedly fed back into the de-allocation routine until there are no pages left. De-allocation will always terminate as the first list corresponds to a segment of length one. The following pseudo-code illustrates this de-allocation procedure:

```
De-Allocate (length) :
        Round 'length' down to the nearest power of two.
        Place a segment of rounded length into the corresponding list.
        If (starting length - rounded length > 0)
                De-Allocate (starting length - rounded length).
```

As it stands, the algorithm makes no attempt to reduce memory fragmentation. The eventual consequence is that all of the free pages will end up in small segments and a request for a large segment will fail despite the fact that there are enough contiguous pages to satisfy the request. This problem is partially rectified by a simple addition to the de-allocation routine. If two segments are positioned one after the other, they can be combined to produce one segment, a power-of-two greater in length. This fact is used to merge segments wherever possible during de-allocation.

Recall that the lists are organised in increasing order. The modification to the de-allocation routine begins once the insertion point in the appropriate list has been found but prior to actually inserting the pages. The indices of the segments before and after the insertion point are examined to see if either could be merged with the new segment to produce a larger segment. If not, insertion proceeds as before. If a merge candidate *is* found, this segment is removed from the list and merged with the new segment. The resulting larger segment is then inserted into the list corresponding to its new length. (This insertion could also lead to a merge). Finally the remaining pages are de-allocated as before. The following diagram illustrates a list insertion with merging:

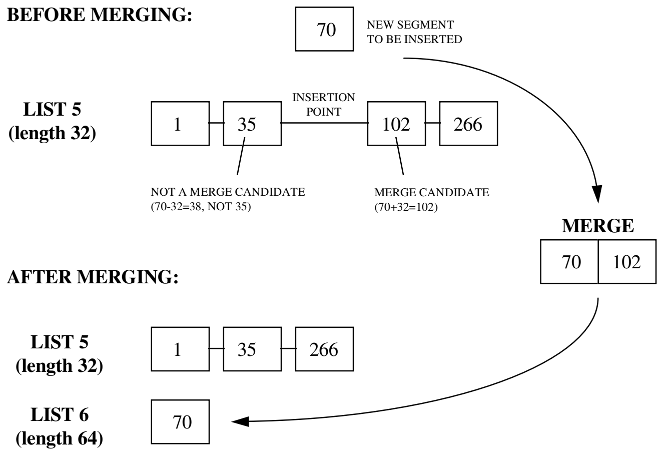


Figure 4.7 - List Insertion and Merging

4.3.2 Allocation at a Specific Memory Location

The allocation of memory at a specific address can also be achieved, although the solution is less intuitively appealing. The technique used attempts to allocate all of the pages between two indices by piecing together empty segments containing the required pages. The allocation routine proceeds as follows:

The lists are searched in turn until a segment containing the starting index is found. This segment is removed and the pages within the segment, before the starting index are returned to the lists. The lists are then repeatedly searched for segments with starting indices directly after the previously removed segment. If such a segment cannot be found, there must already be a previously allocated segment between the two indices and so the allocation fails. Otherwise, the segment is removed and the search and removal continues until a segment containing the final index is removed. Spare pages after the final index are returned to the lists. The following diagram illustrates this technique:

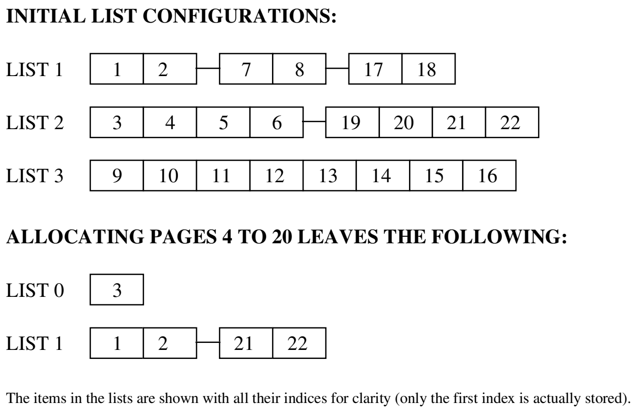


Figure 4.8 - Allocation Between Two Indices

As it stands, the page allocation/de-allocation algorithm is entirely self contained (i.e. it does not require any external data to operate). Indeed, it is implemented as a separate module to allow the algorithm to be easily changed and/or modified. A few more steps are therefore required to link the procedures into the main DVME interface.

4.3.3 Memory-Management in the DVME Interface

On allocation of a segment of memory (either at a specified or unspecified location), the pages within the segment must be marked in the page table as *allocated*. This is done by setting either the mutable or immutable bit (depending on the type of segment allocated) of the PTE for every page in the segment. To facilitate de-allocation, the first page in the segment also has its *segment marker* bit set. All other pages in the segment have their *segment marker* bits clear. Finally, the address field of the PTE is filled in. This is performed via a simple incremental calculation from the starting address as there is a one-one mapping between the PTEs and the data area pages. Recall that it is not necessary to fill in the address field for local pages. Nevertheless, as an optimisation, these addresses are calculated once during allocation and stored in the page-table rather than re-calculating on every memory access. This optimisation also allows a single access mechanism to be used for both local and locally cached pages.

To perform the de-allocation of a segment of memory, the length of the segment must be known. The length is calculated using the page table as follows. The first page in the segment is checked to see if the *segment-marker* bit is set (i.e. a segment does actually start at the specified address). The routine then counts the PTEs following this starting page until either:

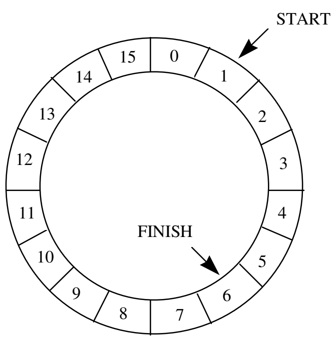
1. An entry containing another set segment marker bit is encountered, or
2. An entry containing an unallocated page is encountered, or
3. The end of the local page table is reached.

De-allocation of the segment can then proceed using this calculated segment length. Finally, the PTEs associated with the segment must be cleared.

4.4 Message Buffering

The DVME also provides a facility for transferring small messages between processes. These messages are always two words in length and typically consist of a result code and an argument. The message is also tagged with the name of the sending process. At present, these messages are only used while garbage-collection.

One of the limitations of the Cray T3D architecture discussed earlier is that signals cannot be caught by a single processor. This means that there is no immediate method of informing a process when a new message arrives. Each process must therefore periodically check for incoming messages at suitable points in the code. A 128 entry circular-buffer is used to store the incoming messages. The explanation of the operation of this buffer is confined to the following diagram as it is a very common data-structure.



START points to the first item in the BUFFER.
FINISH points to the last item in the BUFFER.
LENGTH is the number of items in the BUFFER.

INITIALLY, both START and FINISH point to item 0.

IF START=FINISH the BUFFER is empty.
IF ((FINISH+1) mod LENGTH) = START the BUFFER is full.

ADDING AN ITEM:
CHECK BUFFER not full.
BUFFER[FINISH] = new item.
FINISH = (FINISH+1) mod LENGTH.

REMOVING AN ITEM:
CHECK BUFFER not empty.
item = BUFFER[START].
START = (START+1) mod LENGTH.

Figure 4.9 - A Circular-Buffer

A problem that can arise with this technique is that two (or more) processes may wish to send a message to the same destination simultaneously. It is possible that one message may overwrite the other if one of the processes reads the value of the 'finish' variable before the other has updated it. This problem can be overcome by using a mutual exclusion algorithm. Fortunately the Cray SHMEM library provides an atomic-swap operation which simplifies the task. The atomic-swap operator places a value into a memory location (on any processor) and returns the old value at that location atomically. The following pseudo-code, when executed by every process, ensures that only one process can enter the *critical-section* at a time. The *lock* variable is shared by every process:

```
do
  old_lock = atomic_swap (1, lock);
while (old_lock == 0);
< critical-section >
atomic_swap (0, lock);
```

This solution is not entirely satisfactory as it can lead to starvation (e.g. one process may be running slower and therefore never get a chance to enter the critical section). However, this is unlikely to be a problem in practice.

When using the message transfer operations it is important to be aware of the possibility of deadlock (e.g. one process attempts to write into a full buffer on another processor while this other process is attempting to do the reverse). This can be avoided by checking for incoming messages when the buffer-full condition is detected.

5. The C-LEMMA Memory-Platform

The Distributed Virtual Memory Emulator (DVME) provides a considerable number of the functions necessary to implement the complete C-LEMMA interface. The remaining tasks concern the architecture/network independent sections of C-LEMMA:

1. Controlled allocation of mutable and immutable objects.
2. Extended operations on mutable objects (e.g. mutable freezing).
3. Global Garbage-Collection.

The complete C-LEMMA interface closely follows the definition given in [Matt95a] with a few small exceptions. The most notable exception is that asynchronous operation is not supported. The LEMMA interface makes provision for a number of the function calls to be left to complete in the background while execution continues. Given the single-process-per-processor limitation of the Cray it is not possible to provide this facility. However, the lack of asynchronous operation does not violate any of the LEMMA operating principles as all of these functions may also be called synchronously. The other exceptions involve the new garbage-collection algorithm used for the Cray and will be dealt with in the appropriate section.

As a consequence of using a software emulated virtual memory system, the C-LEMMA interface also contains two additional functions for the reading and writing of immutable objects. Previously, a transparent mechanism using page-fault signals was used to access and cache immutable objects. The new functions simply pass the memory requests directly into the DVME interface.

The following sections describe the main algorithms and techniques used in the implementation of the C-LEMMA interface.

5.1 Controlled Allocation of Mutable and Immutable Objects

Within the DVME, all data is handled in terms of *pages* and *words*. At the LEMMA level, data is treated in terms of arbitrary sized *objects*. Consequently, a small but important part of the C-LEMMA implementation concerns the mapping of data objects into DVME pages. To avoid wasting memory, C-LEMMA attempts to pack objects into pages during allocation. Objects are not permitted to cross page boundaries unless the size of the object is greater than a page. While this restriction is not strictly necessary, it ensures that small immutable objects do not require more than one page of cache and it also simplifies the allocation routine. The data structure shown below is used to keep track of the allocation of objects into mutable and immutable pages:

```
structure allocators
{
  word mutspace;      /* Number of words left in mutable page */
  word immutspace;    /* Number of words left in immutable page */
  word mutptr;        /* Starting address of free mutable space */
  word immutptr;      /* Starting address of free immutable space */
};
```

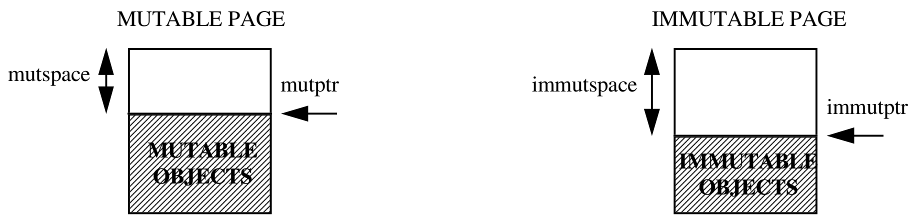


Figure 5.1 - LEMMA Page Allocation Structure

When space is required for a new object, C-LEMMA first checks the allocation structure to see if the object will fit in the remaining space. If not, new pages are obtained from the DVME to hold the object. In either case, the allocation tracking structure is updated to show the remaining space in the page. If the object spans more than one page, the allocation structure records the free space in the final page occupied by the object. Typically, objects are allocated in this manner one at a time. However, C-LEMMA provides the facility for allocating a larger area of memory (a *segment*) to hold several immutable objects at once. The objects within a segment may form circular structures. A segment must be fully filled in before another immutable allocation can take place.

Mutable objects can only be allocated one at a time. A header is appended to the beginning of each mutable object as it is allocated. The format of this header is shown below (figure 5.2). The `magic` word is used to provide a crude mechanism for verifying that an area of memory is actually a mutable object (it is set to the integer 12345). The `length` word stores the size of the object. The remaining fields will be explained in the next section. The header is hidden from the layers above C-LEMMA as the allocation mechanism only returns the starting address of the object itself.

```
structure mutable_header
{
  word magic;                /* Magic number */
  word length;               /* Object length */
  boolean frozen;            /* Frozen ? */
  virtual_address old;       /* Old address */
};
```

Figure 5.2 - Mutable Object Header

After a certain number of pages have been allocated, the garbage-collector is automatically invoked to reclaim any memory no longer in use. Garbage-collection can also be explicitly started at any time.

5.2 Extended Operations on Mutable Objects

The LEMMA interface contains two additional operations that can be performed on mutable objects. Under some circumstances it is known that a mutable object will not have its value changed again after a certain point. The LEMMA interface contains a mechanism for *freezing* such a mutable. This effectively changes a mutable object into an immutable one. At first sight, this would appear to be simply a case of changing the PTE entries for the mutable object into immutable ones at the DVME level. However, objects are packed into pages by C-LEMMA and this operation could also affect other mutable objects. To get around this, mutable objects could simply be allocated in individual pages, but this was considered this to be a waste of valuable memory (there are only 4096 pages available).

The solution adopted by the W-LEMMA interface moves the mutable object into an immutable area of memory. However, a reference to the object may still exist inside any other object anywhere in the virtual address space. A forwarding pointer is therefore placed into the old object location to inform other objects of the new object location. This solution is closely tied to the copying garbage-collection mechanism, used by the W-LEMMA interface, which also moves objects and places forwarding pointers. The C-LEMMA interface uses an alternative garbage-collection mechanism that does not move objects. Rather than deal with the complications of forwarding pointers, an entirely different solution was adopted. The solution avoids any modification to the DVME interface by treating the object as a mutable at the DVME level. Consequently, the object will not be cached, but it is assumed that frozen mutable objects will be sufficiently rare[3]. A Boolean variable in the mutable header is used to indicate that the mutable is frozen. Since all memory accesses are passed through the C-LEMMA interface, it is straightforward to alter the C-LEMMA memory access functions to deal correctly with this special case. The following pseudo-code illustrates the modification to the 'load immutable' function:

```
Load_Immutable (base, offset)
        Query page type at base address.
        If Immutable
                Read data at base + offset.
        Else If Mutable
                Check for 'frozen' condition in mutable header at base address.
                If Frozen
                        Read data at (base + offset) treating as a mutable.
                Else Invalid Address.
        Else Page Fault.
```

Although it has been stated that mutable data is never cached and is always accessed in-place (in C-LEMMA), there is one small exception to this rule. The LEMMA interface definition provides a mechanism for fetching an *exclusive* copy of a mutable object. This ensures that a local copy of the object is available (although not necessarily at the same address). However, no coherency checking is performed and the contents of the object on any other processor is undefined. This operation is provided purely for performance reasons. After the local accesses to the object have been performed, the object is *released* back to its original location.

The implementation of this mechanism in the W-LEMMA interface is an extension of the mutable caching policy. The C-LEMMA interface uses another technique whereby a new area of memory is allocated in the local address area to hold the object and the object is

[3] This appears to be a fair assumption as frozen mutable objects are generated in only one place of the DP/ML compiler.

copied word-at-a-time into this area. The *old address* field in the mutable header is updated to point back to the previous location. The local copy can then be accessed as a normal mutable object. When the object is released, the local mutable object is copied back into its original location, overwriting the original object. All of these operations are performed using the existing DVME functions without modification.

5.3 Distributed Garbage-Collection

As mentioned earlier, the distributed garbage-collection algorithm used in C-LEMMA differs from the one used in the workstation version. There are several reasons why it was chosen to reject the two-space copying approach. A two-space algorithm requires a virtual address area twice as large as the actual data area. This implies an impractical 16Mb page-table on each processor (using DVME), and also complicates the one-one mapping between the local page table area and the data area. Moreover, the compacting of objects means that the DVME page allocation algorithm no longer knows which pages are free (unless all the pages are reallocated). Although inconvenient, these problems could potentially be overcome. The remaining reason for rejecting the two-space approach applies to any implementation. Moving objects between spaces changes their address. This renders all cached data (both at the underlying hardware and DVME software levels) invalid and causes a flurry of unnecessary activity while the cache contents are renewed.

The distributed garbage-collector used in C-LEMMA is based on an algorithm presented in [Kord93]. The algorithm in this paper performs all object traversals *in-place*, in the sense that objects are not moved. This approach effectively overcomes all the problems of the two-space approach highlighted above. The technique used is similar to a traditional *mark and sweep* collector, but requires only one pass through the objects. A summary of the basic operation of the algorithm is given below. To bring the description of the algorithm in line with the one described in the paper it is first necessary to define a few terms and their meanings in the context of the C-LEMMA interface:

A *subheap* is defined to be the part of the heap managed by a single *node*. This corresponds to the section of the virtual address space managed by a single processor in C-LEMMA.

The garbage-collector assumes the possibility of *logical migration* of objects. That is, objects belonging to a single subheap can reside at many nodes. In the context of C-LEMMA, this corresponds to immutable objects residing in the cache. It is worth noting that the garbage-collector could also cope with the caching of mutable data should the current access-in-place mechanism be changed.

Garbage-collection is performed on a *per-subheap* basis. Therefore, the co-operation of several nodes may be required to complete the collection of a single subheap. The garbage-collection algorithm relies heavily upon the maintenance of a number of data structures described below:

5.3.1 Garbage-Collection Data Structures

- For each node, a *ParticipantList* data structure is maintained. This list keeps track of all the nodes that have cached objects belonging to the local subheap. All nodes in this list must participate in the collection of the local subheap.
- For every subheap, an *object directory table* (ODT) is maintained. This table contains entries for objects in the local subheap that are referenced from objects belonging to other subheaps (using a conservative estimate). The *roots* for the collection of the local subheap are root_set_for_local_node $\cup$ ODT.

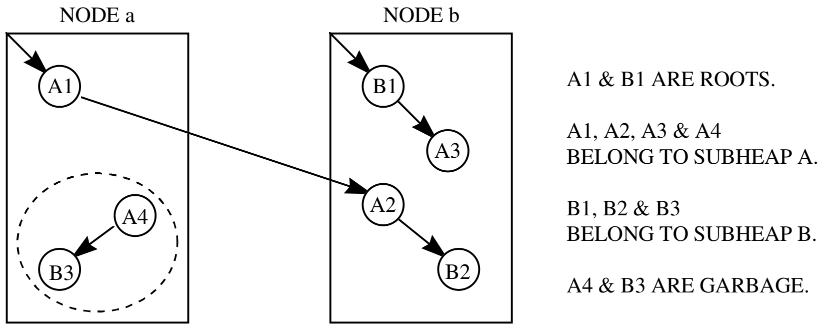
- A *non-local reference table* (NRT) is also maintained for every subheap. This table contains entries for objects in other subheaps that are referenced by objects in the local subheap. An inter-subheap reference always has an entry in both the local NRT and the corresponding remote ODT. To avoid a constant flow of messages between nodes, each node maintains a local copy (an *image*) of the ODT and NRT for every subheap. At the beginning of collection, the union of these images is calculated to produce the complete ODT and NRT.
- Each node is responsible for the traversal (i.e. following the pointers) of objects that it has cached. To facilitate this, an *external-reference table* (ERT) is maintained on each node. If, during the collection of a subheap, a reference to an object on another node is encountered, this reference is placed into the ERT on that node. Traversal continues on this node by arranging each node to traverse all objects with entries in its ERT. Another table called the *pending table* (PT) is maintained on the node responsible for the subheap currently being collected. This table stores all of the objects from which traversals need to be performed (i.e. a local copy of all the entries which have been placed into the ERT of other nodes).
- A *free-list* (FL) on each node keeps track of the unused space in the local subheap. At the start of collection, another *new-free-list* (NLF) is initialised with an optimistic estimate that the entire subheap is empty. As collection proceeds, traversed objects are marked in the NFL. At the end of collection, the NFL is copied into the FL. This process effectively removes all of the garbage objects.
- Finally, each node contains a *GCcount* variable which records the number of times a collection has been performed on the local subheap. ODT and NRT entries are time-stamped with the value of GCcount when they are created. However, the additional complexity associated with time-stamping will not be dealt with in this explanation.

5.3.2 The Garbage-Collection Algorithm

The following algorithm is used to perform the garbage-collection of a subheap. The main work of the algorithm is done by the local node containing the subheap. However, the other participants are required to co-operate at various stages (particularly during traversals).

1. The local node requests the ODT and NRT images from all nodes in the ParticipantList. These images are merged together with the local ODT and NRT entries.
2. All the root object references and ODT entries are copied into the ERT and PT.
3. A recursive object traversal is done on every object with an entry in the ERT (on all participating nodes). Traversal terminates when either, there are no pointers remaining, the next object resides on a different node or, the next object belongs to a different subheap. If the next object resides on a different node, an entry is added to the ERT on that node. An entry is also added to the PT of the node containing the subheap. If the next object belongs to a different subheap, then traversal of this object is not part of the collection of the current subheap and so a entry is added to the NRT. Traversed objects are marked in the NFL. Each node maintains a separate image of the NFL for the subheap until the end of the collection.
4. When the recursive traversal of an object in the ERT has finished, the corresponding entry is removed from the PT on the node containing the subheap. Collection finishes when all of the entries have been removed from the PT.
5. The local node requests all of the NRT and NFL images from the participants. These images are merged with the local NRT and NFL.
6. The garbage objects are identified by comparing the FL with the NFL. Objects that do not appear in the NFL are garbage. These garbage objects are subsequently removed on all nodes.

The following diagram illustrates the garbage-collection process (note that the FL shows the objects which are allocated for clarity):



BEFORE COLLECTION :

ODT(A):	A3	ODT(B):	B2, B3
NRT (A):	B3, B2	NRT(B):	A3
FL(A):	A1, A2, A3, A4	FL(B):	B1, B2, B3

AFTER COLLECTION OF SUBHEAP A (A4 REMOVED) :

ODT(A):	A3	ODT(B):	B2
NRT(A):	B2	NRT(B):	A3
FL(A):	A1, A2, A3	FL(B):	B1, B2, B3

B3 WILL BE REMOVED WHEN SUBHEAP B IS COLLECTED.

Figure 5.3 - Garbage-Collection of Subheap A

Garbage-collection of a subheap may proceed asynchronously with respect to the collection of any other subheap. For example, a node may be garbage-collecting its own subheap as well as participating in the collection of one or more other subheaps. A global garbage-collection corresponds to the collection of all the subheaps. Note that the simplified algorithm presented here does not correctly identify *cyclic* garbage. To collect cyclic garbage, the full algorithm as described in the paper [Kord93] is required.

5.3.3 Garbage-Collection in C-LEMMA

The C-LEMMA interface contains an implementation of the garbage-collection algorithm presented above. However, the garbage-collection routine is currently separate from the C-LEMMA interface as it has yet to be properly tested. Nevertheless, only a small amount of extra work remains to integrate the collector with the C-LEMMA interface.

6. Language-Specific Memory Management

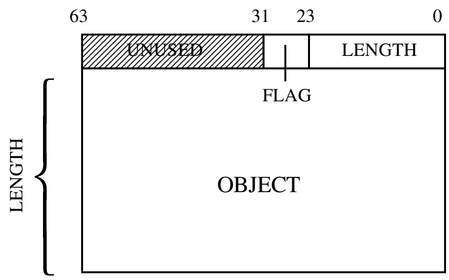
The C-LEMMA interface handles the allocation, sharing, and garbage-collection of generic data *objects*. LEMMA is not permitted to examine the contents of an object as the details of the object representations are specific to the higher-level language implementation. Nevertheless, there are a number of occasions where LEMMA requires specific information about the contents of a particular object (e.g. during garbage-collection traversals). Therefore, a separate language-specific interface, the *Run Time System* (RTS), is also required [Matt95a]. The functions provided by the RTS are as follows:

1. Returning the size of an object given its address.
2. Determining if an object contains constants or addresses, and where.
3. Applying a function to all of the addresses in an object.
4. Supplying the roots of all objects accessible by the machine.
5. Closing an immutable segment of memory.
6. Handling messages and system signals (W-LEMMA only).
7. Moving objects and placing forwarding pointers (W-LEMMA only).

The implementation of the RTS interface on the Cray is very straightforward. The new garbage-collection algorithm used in the C-LEMMA interface means that the most complicated part of the RTS, moving of objects and the placing of forwarding pointers, is no longer required. The following section outlines the construction of an RTS for the DP/ML compiler.

6.1 An RTS for DP/ML

In order to provide an RTS for the DP/ML compiler it is necessary to examine the representation of data objects in DP/ML. A detailed description of their structure can be found in [Matt95c]. The first word in every object contains the length of the object together with a flag indicating the content of the object. The following diagram illustrates the layout of an object:



FLAG BIT DEFINITIONS:

BIT	
0	BYTE SEGMENT
1	CODE SEGMENT
2	CLOSURE
3	STACK SEGMENT
4	RESERVED
5	RESERVED
6	MUTABLE SEGMENT
7	FORWARDING POINTER

Figure 6.1 - DP/ML Object Layout

Clearly, the first two RTS functions can be provided simply by examining the first word in the object. Closing an immutable segment (function 5) corresponds to updating the length field of the header word to the actual length of the data within the segment.

The RTS (function 4) supplies all the root objects for garbage-collection. The root objects are determined using a registry structure maintained at a higher-level. During a collection, it is necessary to traverse all of the pointers inside an object. The RTS provides a function (3) which applies another function (supplied as an argument) to all of the addresses in an object. With a little thought, it is clear that a recursive pointer traversal can be performed by supplying the recursive traversal function itself to this RTS function. The RTS function determines all of the addresses inside an object as follows. The object is *scanned* word-at-a-time for potential addresses. That is, all values inside the object that correspond to valid memory locations. To avoid misinterpreting data items as addresses, the DP/ML interpreter allocates such items in *byte-segments* which cannot be scanned. The interpreter op-codes can be distinguished as they do not correspond to valid addresses.

7. Programming with the C-LEMMA Interface

This section contains a description of the C-LEMMA interface, from a programmers perspective, using the implementation of the Distributed Poly/ML (DP/ML) byte-code interpreter on the Cray T3D [McAd97] as an example of the style and techniques to use. This section is intended to help application programmers who wish to use C-LEMMA without delving too deeply into details of its implementation, though we advise that your read other relevant material such as the LEMMA interface definition [Matt95a]. We start with some remarks about the architecture of the DP/ML system, then look at how to program different types of memory use with LEMMA. In particular, we will look at how different *layers* of C-LEMMA are used at different times for different tasks.

7.1 Architecture of C-LEMMA Programs

The use of C-LEMMA has a large effect on the logical structure of programs. The structure of the DP/ML system is described below.

7.1.1 Structure of DP/ML

The complete DP/ML system [Matt95c] consists of three separate parts: the *persistent database*, the *byte-code interpreter*[4], and the *run-time module* (part of which is the LEMMA interface):

- The *persistent database* is essentially a memory dump of a running DP/ML system.
- The *byte-code interpreter* (written in C) which emulates an abstract machine; ML programs are compiled to byte-code then executed by the interpreter.
- The *run-time support system* (also written in C) is used by the interpreter to communicate with the hardware and the operating system.

The C-LEMMA interface forms part of the run-time support system. The byte-code interpreter is an application program using C-LEMMA, and the persistent database is loaded into LEMMA memory, so C-LEMMA is central to the program architecture. An idealised view of how the system is constructed is a hierarchy of memory controllers each offering approximately the same functionality but with a different level of abstraction:

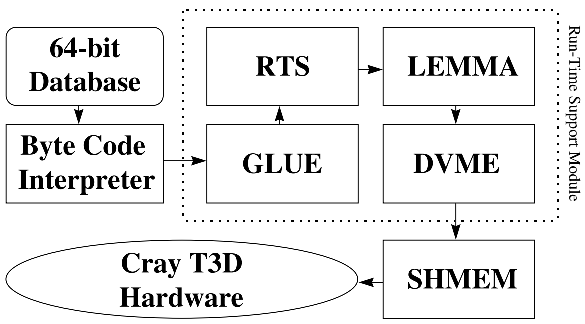


Figure 7.1 - Layers in DP/ML

[4] The native machine-code generating version will not be considered due to the additional complexity involved.

In reality, the interplay between the various layers is significantly more complex than indicated above. For the purpose of efficiency, layers may communicate directly with lower layers, rather than only the layer directly below them. For example, the byte-code interpreter contains sections which deal directly with memory hardware without involving the intermediate layers.

7.1.2 Linking Programs to C-LEMMA

DP/ML, and any other program using C-LEMMA, has the C-LEMMA system linked into the executable file (when it is compiled). Programs use C-LEMMA's facilities by calling its functions.

The T3D has a Single Program Multiple Data (SPMD) architecture (refer to section 3.1). When you compile your C programs and link them to C-LEMMA there is a single executable file which is loaded onto every processor in your partition (a fixed number of processors used exclusively by your program). Each instance of the program (on its own processor) must establish which processor it is running on and act accordingly. This will be explained in more detail later.

7.1.3 GLUE and C-LEMMA

The GLUE layer provides an interface between the byte-code interpreter and the C-LEMMA interface. The majority of DP/ML memory interactions are performed using the GLUE functions. These functions offer similar facilities to programs as does LEMMA, but at a higher level of abstraction. GLUE is designed for use only with DP/ML, so other programs must use C-LEMMA functions directly. The following descriptions deal with both GLUE and C-LEMMA functions.

7.2 System Initialisation

All C-LEMMA programs must *initialise* the system before performing any memory interactions. The functions to call when initialising depend on whether programs are using GLUE or LEMMA.

7.2.1 Initialisation using GLUE

Every processor must call the function `GLUE_initialise_memory` when it starts. The client must then set up the garbage-collector using the `GLUE_register_gc_proc(GCprocedure, LEMMA_GC_Kind_Strong)` function. The client should then check for servers with the function `GLUE_find_server` as we will see in the next section.

7.2.2 Initialisation using LEMMA

Programs using C-LEMMA directly should call `LEMMA_initialise`, then set up the garbage collector. The LEMMA interface definition contains details of how this is done.

7.2.3 Servers and Clients

Earlier sections have described the differences between the *client* and the *servers*. It is important that you make the client act differently from servers. One task of the client should is to locate servers when it starts. The client is always processor number 0. The processor number is stored in the global value `LEMMA_server_no` which is set when you initialise C-LEMMA. Apart from the restriction that the client should use 'find server' and that the servers cannot call 'find server', there are few restrictions on what the client and servers should do.

7.2.4 Loading a File into LEMMA Memory

An important task of DP/ML is to initialise itself by loading a database (memory image) file. The database is to be shared by all the processors in the partition and, therefore, must be placed in LEMMA memory. The database contains LEMMA/RTS objects, so when it is loaded objects are created without directly using the C-LEMMA functions.

Here, we give a short description of how to load a database into LEMMA memory. The description applies to loading any image containing LEMMA objects into LEMMA memory. Typically, loading a memory image into virtual memory would use the (UNIX) operating system `mmap` function, however, as this function is unavailable on the T3D we must use DVME:

The loader function must declare these variables:

```
/* Number of pages required, and size of segments in bytes */
int immutPages, mutPages, immutSize, mutSize;
DVME_addr immutDVME, mutDVME;     /* Virtual addresses to allocate */
byte *immutReal, *mutReal;        /* Real addresses to load to */
```

The program must set the number of pages required, sizes of the segments and the addresses they are to be loaded to. C-LEMMA will later give values for the real addresses. After setting the variables above, use these lines to allocate virtual memory:

```
LEMMA_ASSERT(
  (DVME_allocate_at_address(immutDVME, immutPages, DVME_immutable) ==
  DVME_success), "DVME_allocate immut failed in LEMMA_load");
LEMMA_ASSERT(
  (DVME_allocate_at_address(mutDVME, mutPages,DVME_mutable) ==
  DVME_success), "DVME_allocate mut failed in LEMMA_load");
```

After allocating virtual memory, use these lines to find the real addresses of the memory:

```
DVME_virtual_to_real((DVME_addr)immutDVME, &immutReal);
DVME_virtual_to_real((DVME_addr)mutDVME, &mutReal);
```

Next you must set up the file `db` so that it is ready to load immutable data. Then load it using the line:

```
fread(immutReal, 1, immutSize, db);
```

Finally set up `db` so it is ready to load mutable data, then load it:

```
fread(mutReal, 1, mutSize, db);
```

The code above makes use of DVME functions and real memory operations which will be described later.

7.3 Allocation of Data Objects

7.3.1 Allocation in GLUE

When creating an object using GLUE, the object type (mutable or immutable) and size must be known. The `GLUE_alloc` function takes a single word parameter which is the size of the object to allocate with flags indicating the type of object. The parameter becomes the header of the object. For example, to allocate an eight word mutable object:

```
RTS_addr newObject = GLUE_alloc(RTS_F_mutable | 8);
```

GLUE does not offer exactly the same functionality as LEMMA, so LEMMA must be used directly for more complex allocation and memory manipulation such as *freezing* an object (converting a mutable object into an immutable one).

7.3.2 Allocation in LEMMA

In addition to `LEMMA_allocate_mutable` and `LEMMA_allocate_immutable`, LEMMA offers a number of other functions for allocation and object manipulation. Refer to the LEMMA interface definition for details on how to use these.

7.4 Accessing Objects

Because memory is being managed by a software library, rather than hardware and the operating system, objects cannot be accessed using conventional means. Instead function calls must be used to read and write memory.

7.4.1 Access with GLUE

GLUE provides a simple way for programs to access bytes within LEMMA objects. It provides functions to read and write bytes within mutable and immutable objects, taking object address and offsets as parameters:

```
LEMMA_word GLUE_get_mutable(RTS_addr addr, LEMMA_word offset);
void GLUE_set_mutable(RTS_addr addr, LEMMA_word offset,
                      LEMMA_word value);
LEMMA_word GLUE_get_immutable(RTS_addr addr, LEMMA_word offset);
void GLUE_set_immutable(RTS_addr addr, LEMMA_word offset,
                        LEMMA_word value);
```

The meanings of these functions should be self explanatory. The `GLUE_set_immutable` function should only be used immediately after creating the object. These functions lack efficiency but are ideal for prototyping when designing programs.

7.4.2 Access with LEMMA

As explained in section 2.1, W-LEMMA implements a DSM system transparently using page-fault signals. The alternative technique used in C-LEMMA requires all memory accesses to be explicit. Consequently, two extra functions are required for reading and writing immutable objects. These functions are analogous to the `LEMMA_load_mutable` and `LEMMA_assign_mutable` functions already present in the LEMMA interface.

```
LEMMA_result LEMMA_load_immutable(int is_address, LEMMA_addr,
               int offset, LEMMA_word *result, LEMMA_synch_type,
               int *synchtoken);
LEMMA_result LEMMA_assign_immutable(int is_address, LEMMA_addr,
               int offset, LEMMA_word value, LEMMA_synch_type,
               int *synchtoken);
```

Programmers are advised to read the next section on using real addresses to make their code more efficient.

7.5 Using Real Memory Addresses

It is often useful to bypass C-LEMMA functions and access objects directly using *real* memory addresses. This allows code to be more like conventional C code (allowing, for example, array indexing) and can make programs more efficient by reducing the overheads of function calling and repeated address resolution. To use normal memory accesses with data in C-LEMMA memory, DVME must first be used to find the real address of the data, this is a simple operation, a short fragment of code is shown in below:

Declare DVME and real address forms of the program counter:

```
DVME_addr pcDVME;
byte *pc;
word instr;
```

Given a virtual address, the corresponding real address is calculated as follows:

```
LEMMA_ASSERT(DVME_virtual_to_real(pcDVME, &pc)==DVME_SUCCESS,
      "Failed to convert PC to real address");
```

The real address can then be used as a conventional C pointer:

```
instr = *(pc++); /* read next instruction
                    and increment program counter */
```

7.5.1 When to use Real Addresses

An important consideration in the design of a C-LEMMA program is when to use real addresses instead of virtual ones:

1. As the C-LEMMA garbage collector does not move objects, once a program has a real pointer to some data it is safe to keep this pointer for later accesses. Real pointer should not, however, be used inside objects in the LEMMA memory as they are not safe for use by other processors.
2. As it is not possible to convert a real address back into a virtual address, avoid converting DVME addresses then performing arithmetic operations on the resulting pointer. Doing this would prevent you from sending the result to another processor.
3. DP/ML stores the real addresses of the stack and current code as processes do not migrate from one process to another, this saves processor time by avoiding use of C-LEMMA functions where they are not necessary.

Therefore, it is advisable to use real addresses where possible to keep the code simple and make programs more efficient, provided it can be determined that these addresses will not migrate to another processor where they will be invalid.

7.6 Message Passing Communication

Although C-LEMMA is primarily a shared-memory system, it also offers a few simple message passing primitives for convenience. Message passing can be used to send simple signals from one processor to another, in particular it is useful for client to server communication.

Messages are described as *RTS messages* as they are primarily designed for synchronising run-time operations such as garbage collection. DP/ML uses these messages so that the client can signal a server to start executing a function. However, it is not advisable to use these primitives to write programs with in a message passing style. A specialised message passing interface such as MPI [Mpif94] is more suitable for this.

7.6.1 Message Passing Primitives

A *message* consists of two words. The first word is an identifier giving the message type, the second is data associated with this (e.g. an address). Functions are provided for asynchronous send and receive (`LEMMA_send_rts_message` and `LEMMA_get_rts_message`) and synchronous receive (`GLUE_wait_for_message`). Synchronous receive simply busy-waits until a message arrives. Several values for the first word (message type) are reserved, these can be found in the appendix (section 9.4). Additional values may be added to this file for user-defined messages.

8. Further Work and Concluding Remarks

The purpose of C-LEMMA is to support the implementation of concurrent languages on the Cray T3D. There are two main achievements associated with the new C-LEMMA interface over the previous workstation-based version.

1. The construction of the *Distributed Virtual Memory Emulator* (DVME) library. This library implements a paging distributed virtual memory sub-system. The C-LEMMA interface is based on the functionality provided by this library. However, the library could also be used by other Cray applications requiring a distributed virtual address space. An added advantage of this library is that it separates the OS and network dependencies from the C-LEMMA interface providing an overall more portable system. In this project, the library is an entirely software-based due to the limitations of the Cray hardware. However, the possibility of alternative implementations on different architectures is available.
2. The implementation of a new *in-place* global garbage-collection algorithm. This algorithm should provide a performance advantage over the *two-space copying* approach used in W-LEMMA, as it avoids the inefficiencies and complications of moving objects and placing forwarding pointers.

The distributed shared memory system is working and has been thoroughly tested. A detailed study of the Cray T3D shared memory system revealed that the performance of the C-LEMMA implementation will almost certainly be better than the W-LEMMA implementation, although a direct comparison has yet to be performed.

8.1 Improving the C-LEMMA Interface.

A number of techniques for improving the performance and/or efficiency of C-LEMMA were suggested earlier in this report. It is hoped that these will be implemented in a future project.

1. A large number of the C-LEMMA functions could be rewritten as C-macros, thereby eliminating the associated function-call overhead.
2. The 8Mb page-table inside the DVME interface could be implemented using an efficient hashing technique to save a considerable amount of memory.
3. An analysis of the Cray's shared memory performance revealed that a large increase in performance could be obtained through the caching of mutable objects. This would also enhance the appeal of the C-LEMMA interface for imperative languages which contain only mutable data objects. The mutable caching scheme used in the W-LEMMA interface is detailed in [Serg94]. The scheme adaptively selects between three different caching policies on an object-by-object basis by examining the access patterns to the object. This technique potentially provides far greater performance than a single system-wide caching policy.

8.2 Improving the Cray T3D Architecture

Despite the predicted increase in performance, it is clear that the full potential of the Cray T3D is not being realised. A significant amount of the performance is wasted due to the necessity of the software-based virtual address translation scheme. A number of improvements to the Cray T3D architecture, each of which would improve overall performance of the DVME interface, are outlined below.

1. If individual processors could catch operating system signals, most immutable data accesses would not have to be passed through the emulator. Instead, *page-fault* signals could be handled on remote memory accesses as in the W-LEMMA system. This would significantly improve access times to locally resident immutable objects. User signals could also be used to inform processors of arriving messages, avoiding the necessity of repeatedly checking for incoming messages, thereby improving the performance of the new garbage-collection routine. Signals could also be used to provide a time-slicing mechanism, permitting more than one process per processor.
2. The overheads associated with the Cray SHMEM library come from the dynamic alteration of the DTB-annex table to provide the illusion of a globally addressable memory. If the Cray T3D provided hardware support for a global address space, then there would be little need for the SHMEM library. This would provide a significant boost in performance to the DVME interface and the other Cray message-passing systems. It is difficult to see why the Cray designers did not improve performance in this way since every parallel process is affected. However, it is understood that the new Cray T3E incorporates this improvement.
3. The most radical proposal would be to provide caching of remote data on each node. This would significantly simplify the DVME interface and almost certainly improve performance over a software managed cache. However, a complicated distributed cache-invalidation scheme would be required and would probably limit the overall scaleability of the Cray architecture.

9. Appendix - C-LEMMA Interface Prototypes

9.1 DVME Interface Header File.

```
/**************************************************************************/
/* Title : Cray T3D Distributed Virtual Memory Emulator (DVME) v1.05      */
/* File  : dvme.h - Functions, types and variables exported from DVME     */
/**************************************************************************/

/* Word length */
typedef long DVME_addr, DVME_word;      /* Word size is 64-bit */

/* Inter-Processor Message Format */
struct dvme_message
{
  DVME_word data0, data1;
  int sender_pe;
};
typedef struct dvme_message DVME_message;

/* Exit conditions */
typedef enum
{
  DVME_success,
  DVME_buffer_full,                     /* Message buffer conditions */
  DVME_buffer_empty,
  DVME_fail                             /* i.e. Page Fault */
} DVME_result;

/* Status descriptions */
typedef enum
{
  DVME_mutable,                         /* Mutable */
  DVME_immutable,                       /* Immutable */
  DVME_unallocated                      /* Free Space */
} DVME_page_status;

/* Globals */
extern int DVME_mype;                   /* Processor number */
extern int DVME_numpes;                 /* Total number of processors */

void DVME_initialise(void);             /* Initialisation */

/* Virtual Address Manipulation */
int DVME_processor(DVME_addr virtual_address);
DVME_addr DVME_make_virtual(DVME_word page_number, int pe);
DVME_result DVME_virtual_to_real(DVME_addr virtual_address,
                                 DVME_word **real);

/* Memory Access */
DVME_result DVME_read_mutable_word(DVME_addr virtual_address,
                                   DVME_word *value);
DVME_result DVME_read_immutable_word(DVME_addr virtual_address,
                                     DVME_word *value);
DVME_result DVME_write_mutable_word(DVME_addr virtual_address,
                                    DVME_word newvalue,
                                    DVME_word *oldvalue);
DVME_result DVME_write_immutable_word(DVME_addr virtual_address,
                                      DVME_word value);

/* Memory Allocation/De-Allocation */
DVME_result DVME_allocate_segment(DVME_addr *virtual_address, int *length,
                                  DVME_page_status type);
DVME_result DVME_deallocate_segment(DVME_addr virtual_address,
                                    DVME_page_status stat);
DVME_result DVME_allocate_at_address(DVME_addr virtual_address, int length,
                                     DVME_page_status stat);
DVME_page_status DVME_query_page_status(DVME_addr virtual_address,
                                        int *pe_no);
```

```
/* Message Passing */
DVME_result DVME_send_message(int dest_pe, DVME_message *message);
DVME_result DVME_get_message(DVME_message *message);

/* Error Handling */
void DVME_crash(char *message);
#define DVME_ASSERT(x,s) ((!(x)) ? (void)(DVME_crash(s)) : (void)0)
```

9.2 DVME Memory Allocation Routine Header File.

```
/**********************************************************************/
/* Title : Cray T3D Distributed Virtual Memory Emulator (DVME) v1.02      */
/* File  : dvme_heap.h - header file for heap management functions        */
/**********************************************************************/

void HEAP_initialise(int length, DVME_addr start);
void HEAP_dealloc_contig_pages(int length, DVME_addr offset);
DVME_result HEAP_alloc_contig_pages(int length, DVME_addr *offset);
void HEAP_allocate_at_address(int length, DVME_addr offset);
```

9.3 C-LEMMA Interface Header File.

```
/**********************************************************************/
/* Title   : Cray T3D Distributed ML Memory Interface (C-LEMMA) v1.02     */
/* File    : lemma.h - exports from LEMMA interface                       */
/**********************************************************************/

/* Word sizes and address formats */
typedef long word;
typedef word LEMMA_word, LEMMA_addr, RTS_addr;

/* Synch types (only LEMMA_SYNCH supported) */
typedef enum
{
  LEMMA_SYNCH,
  LEMMA_ASYNCH
} LEMMA_synch_type;

/* Result codes */
typedef enum
{
  LEMMA_Failed,
  LEMMA_Succeeded,
  LEMMA_InProgress,
  LEMMA_TooMany
} LEMMA_result;

/* Message format */
typedef struct
{
  LEMMA_word rtsm_data0, rtsm_data1;
  int rtsm_sender;
} LEMMA_rts_message;

/* Server information */
extern int LEMMA_server_no;              /* This servers ID */
extern int LEMMA_servers;                /* Total number of servers */

/* Interrupt reasons */
typedef enum
{
  LEMMA_Synch_request,
  LEMMA_RTS_Msg_arrived,
  LEMMA_RTS_Clear_to_send,
  LEMMA_Immutable_complete,
  LEMMA_Mutable_complete
} LEMMA_interrupt_reason;
```

```
/* Garbage-Collection types */
typedef enum
{
  LEMMA_GC_Kind_Strong,
  LEMMA_GC_Kind_Weak
} LEMMA_GC_Kind;
#define PAGE_CREDIT 2048     /* Local pages allocated before GC begins */

/* Allocation structure (for keeping track of allocation inside pages) */
struct allocators
{
  word       mutspace;       /* # words left in current mutable page     */
  word       immutspace;     /* # words left in current immutable page   */
  LEMMA_word mutptr;         /* pointer into current mutable page        */
  LEMMA_word immutptr;       /* pointer into current immutable page      */
};

/* Mutable object header */
typedef struct
{
  word magic;                  /* magic number for checking headers */
  word frozen;                 /* page frozen (0=no) */
  word length;                 /* length of mutable */
  LEMMA_addr old_addr;         /* old address (if get_exclusive used) */
} LEMMA_mutable_header;
#define LEMMA_MAGIC 12345
#define MUTABLE_OVERHEAD (sizeof(LEMMA_mutable_header)/sizeof(LEMMA_word))

/* Error Handling */
#define LEMMA_ASSERT(x,s) ((!(x)) ? (void)(DVME_crash(s)) : (void)0)

/* Initalisation */
LEMMA_result LEMMA_initialise_memory(int is_server, int token,
              LEMMA_addr *interface_addr, unsigned int interface_size);

/* Mutables */
LEMMA_result LEMMA_allocate_mutable(LEMMA_addr *, int);
LEMMA_result LEMMA_load_mutable(int is_address, LEMMA_addr, int offset,
               LEMMA_word *result, LEMMA_synch_type, int *synchtoken);
LEMMA_result LEMMA_assign_mutable(int is_address, LEMMA_addr, int offset,
               LEMMA_word value, LEMMA_word *result, LEMMA_synch_type,
               int *synchtoken);
LEMMA_result LEMMA_check_mutable_operation(int synchtoken,
               LEMMA_word *result);
LEMMA_result LEMMA_freeze_mutable(LEMMA_addr);
LEMMA_result LEMMA_get_exclusive_copy(LEMMA_addr, LEMMA_addr *);
LEMMA_result LEMMA_release_exclusive_copy(LEMMA_addr);
LEMMA_result LEMMA_is_mutable(RTS_addr, int *);

/* Immutables */
LEMMA_result LEMMA_allocate_immutable(LEMMA_addr *, int);
LEMMA_result LEMMA_allocate_immutable_segment(LEMMA_addr *, int *);
LEMMA_result LEMMA_load_immutable(int is_address, LEMMA_addr, int offset,
               LEMMA_word *result, LEMMA_synch_type, int *synchtoken);
LEMMA_result LEMMA_assign_immutable(int is_address, LEMMA_addr, int offset,
               LEMMA_word value, LEMMA_synch_type, int *synchtoken);

/* Message buffering */
LEMMA_result LEMMA_get_rts_message(LEMMA_rts_message *);
LEMMA_result LEMMA_send_rts_message(int, LEMMA_rts_message *);

/* Garbage-collection */
LEMMA_result LEMMA_force_garbage_collection(void);

/* IO handling */
LEMMA_result LEMMA_ensure_object_is_accessible(RTS_addr, LEMMA_synch_type);
LEMMA_result LEMMA_connect_to_server(char *server_name, int *server_address,
               int token);
```

9.4 C-LEMMA Messages

```
/***********************************************************************/
/* Title : C-LEMMA Message Formats                                     */
/* File  : messages.h                                                  */
/***********************************************************************/

/* Messages to be handled by LEMMA_{get,send}_rts_message */

typedef enum
{
  GC_PARTICIPATE,
  GC_DEPART,
  GC_COUNT_A,
  GC_REQ1,
  GC_ODT1,
  GC_NRT1,
  GC_ACK1,
  GC_TRAVERSE,
  GC_ADD,
  GC_UPDATE_A,
  GC_UPDATE_B,
  GC_REQ2,
  GC_NFL2,
  GC_NRT2,
  GC_ACK2,
  GC_COUNT_B,
  GC_REMOVE,
  GC_FINISH,
  SV_START          /* Server, start a process */
} GC_MESSAGE;
```

9.5 RTS Header File.

```
/***********************************************************************/
/* Title : Cray T3D Run-Time System (RTS) v1.02                        */
/* File  : rts.h                                                       */
/***********************************************************************/

void RTS_for_addresses_in_object(LEMMA_addr, void (*)(RTS_addr,
                                 LEMMA_addr, int));
void RTS_apply_to_address(LEMMA_word *p, RTS_addr (*op)(RTS_addr));
void RTS_apply_to_rts_message(LEMMA_rts_message*,
                              RTS_addr (*op)(RTS_addr));
void RTS_crash(char *);
void RTS_close_segment(void);
void RTS_apply_to_roots(RTS_addr (*op)(RTS_addr), LEMMA_GC_Kind strength);
LEMMA_addr RTS_start_of_object(RTS_addr);
int RTS_object_length(LEMMA_addr);
int RTS_object_is_mutable(LEMMA_addr);
void RTS_fill_unused_space(LEMMA_addr, int);

/* RTS object header masks */
#define RTS_Flag_field      0xff000000        /* hi byte of lengthword    */
#define RTS_Length_field    0x00ffffff       /* low 3 bytes of lengthword */

/* ---------------------------- RTS Flags ------------------------------- */
/* F_bytes means the data is bytes and must not be searched for pointers  */
#define RTS_F_bytes         0x01000000
/* F_code means that the data is a code segment with a literal segment at */
/* the end.  Only this must be searched for pointers.                     */
#define RTS_F_code          0x02000000
/* F_first_ptr means that the first word contains the address of the next.*/
/* This is an optimisation of a closure which only contains the address   */
/* of the code and has no free variables. (Only with F_bytes or F_code).  */
#define RTS_F_first_ptr     0x04000000
/* F_stack means the segment is a stack segment. (F_mutable is also set)  */
#define RTS_F_stack         0x08000000
/* F_mutable means that the object is mutable i.e. a variable or a vector */
```

```
#define RTS_F_mutable       0x40000000
/* These next two bits are not looked at by the store management system   */
#define RTS_F_user1         0x10000000  /* No longer used */
#define RTS_F_user2         0x20000000  /* No longer used */
/* F_gc is used by the garbage-collector to indicate an object which has  */
/* been moved to a new address. The rest of the word is the new address.  */
#define RTS_F_gc           0x80000000   /* No longer required */
/* If this bit is set then the address is a code pointer */
#define RTS_CODE_BIT        2

#define RTS_ASSERT(x,s) ((!(x)) ? (void)(DVME_crash(s)) : (void)0)
```

9.6 GLUE Header File.

```
/**************************************************************************/
/* Title : Cray T3D Run-Time System Glue v1.00                            */
/* File  : rts_glue.h                                                     */
/**************************************************************************/

typedef long word;
#define TAGSHIFT        1
#define TAG             1
#define TAGGED(i)       (((i) << TAGSHIFT) + TAG)

/* Initialisation */
void GLUE_initialise_memory(void);
LEMMA_result GLUE_find_server(int pe);

/* Memory allocation and access */
RTS_addr GLUE_alloc(RTS_word size);
LEMMA_word GLUE_get_mutable(RTS_addr addr, LEMMA_word offset);
void GLUE_set_mutable(RTS_addr addr, LEMMA_word offset, LEMMA_word value);
LEMMA_word GLUE_get_immutable(RTS_addr addr, LEMMA_word offset);
void GLUE_set_immutable(RTS_addr addr, LEMMA_word offset, LEMMA_word value);

/* Garbage-collection */
void GLUE_register_gc_proc(void (*op)(RTS_addr (*)(RTS_addr)),
       LEMMA_GC_Kind);

/* Messaging */
void GLUE_wait_for_message(LEMMA_rts_message *m);
```

10. Bibliography.

[Barr94] Barriuso and A. Knies, *"SHMEM User's Guide for C"*, Cray Research Inc., Revision 2.2, 1995.

[Boot95] S. Booth, J. Fisher, P. H. Maccallum, and A. D. Simpson, "*Introduction to the Cray T3D at EPCC"*, EPCC, September 1995.

[Bhoe92] R. A. F. Bhoedjang, *"Porting Concurrent Poly/ML to the Computing Surface"*, Report EPCC-SS92-02, EPCC, September 1992.

[Colo94] G. Colouris, J. Dollimore, and T. Kindberg, *"Distributed Systems: Concepts and Design"*, (Second Edition), Addison Wesley, 1994, Chapter 17.

[Cray93] Cray Research Inc., *"Cray Research MPP Software Guide"*, Cray Research Inc., SG-2508, Revision 1.0, 1993.

[Cray94] Cray Research Inc., *"Cray Standard C Reference Manual"*, Cray Research Inc., SR-2506, Revision 4.0, 1994.

[Hogg94] J. Hogg, *"Poly/ML on Alpha"*, MSc. Project Report, University of Edinburgh, September 1994.

[Kord93] R. Kordale, M. Ahamad and J. Shilling, *"Distributed/Concurrent Garbage Collection in Distributed Shared Memory Systems"*, IEEE Computer, 1993.

[Matt91] D. C. J. Matthews, *"A Distributed Concurrent Implementation of Standard ML"*, Report ECS-LFCS-91-174, LFCS, Edinburgh University, August 1991.

[Matt95a] D. C. J. Matthews and T. Le Sergent*, "LEMMA Interface Definition*", Report ECS-LFCS-95-316, LFCS, Edinburgh University, January 1995.

[Matt95b] D. C. J. Matthews and T. Le Sergent, *"LEMMA: A Distributed Shared Memory with Global and Local Garbage-Collection"*, Report ECS-LFCS-95-325, LFCS, Edinburgh University, June 1995.

[Matt95c] D. C. J. Matthews, *"Papers on Poly/ML"*, Report ECS-LFCS-95-335, LFCS, October 1995.

[McAd97] B. J. McAdam, *"Poly/ML on the Cray T3D: A Byte-Code Interpreter"*, 4th Year Project Report, Department of Computer Science, Edinburgh University, May 1997.

[Miln90] R. Milner, M. Tofte, and R. Harper, *"The Definition of Standard ML"*, MIT press, 1990.

[Mpif94] Message Passing Interface Forum, *"MPI: A Message-Passing Interface Standard"*, May 1994.

[Nitz91] B. Nitzberg and V. Lo, *"Distributed Shared Memory: A Survey of Issues and Algorithms"*, IEEE Computer, August 1991.

[Serg94] T. Le Sergent and D. C. J. Matthews, *"Adaptive selection of protocols for strict coherency in distributed shared memory"*, Report ECS-LFCS-94-306, LFCS, September 1994.