$\mathcal{M}$: A Memory Manager for $\mathcal{L}$

by

Andrew Edward Ayers

B. S., Electrical Engineering and Computer Science
University of Colorado, Boulder (1984)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1988

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
November 18, 1987

Certified by [illegible]
Stephen A. Ward
Associate Professor
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

$\mathcal{M}$: A Memory Manager for $\mathcal{L}$

by

Andrew Edward Ayers

Submitted to the Department of Electrical Engineering and Computer Science
on November 17, 1987, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

$\mathcal{M}$ is a memory system for the $\mathcal{L}$ processor which integrates object-oriented virtual memory and ephemeral garbage collection. The garbage collection and virtual memory subsystems of $\mathcal{M}$ were carefully designed to cooperate so as to avoid performance degradations that can occur when garbage collection is built on top of a traditional paged memory systems. Objects in $\mathcal{M}$ live in one of two namespaces: local memory, which corresponds to main memory in a traditional virtual memory system, or permanent memory, which corresponds to secondary memory. The virtual memory side of $\mathcal{M}$ swaps objects into the local memory on demand. All new (and hopefully temporary) objects are created in local memory as well. The ephemeral mark/sweep garbage collector scans just local memory, reclaiming unreachable local chunks, without having to look at permanent chunks. This avoids thrashing the virtual memory side of $\mathcal{M}$. $\mathcal{M}$ and $\mathcal{L}$ have been implemented on a re-microcoded Texas Instruments Explorer I LISP machine. Benchmarks run on this emulation have shown that $\mathcal{M}$ is able to reclaim temporary objects quickly and efficiently.

Thesis Supervisor: Stephen A. Ward

Associate Professor

Acknowledgments

First and foremost, I would like to thank Steve Ward, my advisor, for giving me the inspiration for and the chance to work on $\mathcal{L}$ in general and $\mathcal{M}$ in particular. Thanks also to the people who are or have been working on the $\mathcal{L}$ project: Chris Terman, who got me interested in $\mathcal{L}$, Lamott Oren, for the early version of the EU emulation, Ken Bice, for putting up with an early version of $\mathcal{M}$, and Rick Saenz, for his work on the $\mathcal{L}$ compiler. Thanks to John Pezaris, who built the two-processor $\mathcal{L}_0$ system, John Wolfe, and Ricardo Jenez. A special thanks to Milan Singh, who has managed to put up with working closely with me on $\mathcal{L}$ for the past year. Thanks to Sharon Chang and Mike Matchett for proving that a user community is not necessarily a bad thing. Thanks too to Sharon Thomas.

I would not have made it this far without the support and encouragement of Jim and Susan Avery, who convinced me to go to graduate school, Ben Balsley for giving me the chance to gain some professional maturity and travel a little, and Sally Wang, Ed Dowski, and Dave Tetenbaum, who showed me that being a graduate student wasn't all that bad. Thanks also to Prof. William May, who insisted that I apply for, and the National Science Foundation for generously granting, fellowship support.

Finally, I would like to thank my parents, family, and friends. I'm not sure that this thesis will make it any clearer to them exactly what it is I have been doing the past few years, but they are partially to blame for its existence.

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014-83-K-0125 and N00014-84-K-0099.

Contents

List of Figures

List of Tables

$\boxed{1}$ Introduction

1.1 The $\mathcal{L}$ Project

Research in functional languages, logic languages, and object-oriented languages has led to the evolution of computational models which traditionally designed computers do not implement efficiently. Spurred by the desire for high-performance implementations of these models, computer architects have begun investigating many non-traditional architectures: PROLOG inference machines, data-flow machines, LISP machines, and multiprocessors of all different shapes and sizes. One such non-traditional architecture, $\mathcal{L}$, is currently under development by the Real-Time Systems Group at the Laboratory for Computer Science.

Modern programming languages include many features that deviate from the traditional programming model: SCHEME [Rees86] has *first-class* procedures and *continuations*; COMMON LISP [Steele84] has `catch` and `throw`, which allow non-local exits; MULTILISP [Halstead84] provides *futures* to allow users to exploit parallelism in functional programs; SMALLTALK [Goldberg83] has an object-based, message-passing view of processing. Because these languages include features that are not easily implemented on a traditional computer, high-performance implementations, if they exist at all, are found primarily on special-purpose workstations (e.g. LISP machines).

The goal of the $\mathcal{L}$ project is to look for alternative machine architectures that can support new language features, like those listed above, more naturally than traditional computers do, without giving up the ability to efficiently execute programs written in traditional languages. Our current proposal for an $\mathcal{L}$ machine involves two primary architectural features: an object-oriented memory system, and an inherent ability to exploit fine-grain parallelism. This thesis describes the design and implementation of $\mathcal{M}$, a prototype for the memory system.

1.2 Overview

This thesis is divided into 7 chapters. Chapter 2 explores the differences between $\mathcal{M}$ and more traditional memory systems and presents the interface between $\mathcal{M}$ and $\mathcal{L}$. Chapter 3 examines the founding principles behind $\mathcal{M}$, and Chapter 4 discusses the design of $\mathcal{M}$. Chapter 5 presents the details of an implementation of $\mathcal{L}$ and $\mathcal{M}$ on Texas Instruments Explorer LISP machines, together with some preliminary performance data. Chapter 6 examines future areas of investigation. Finally, Chapter 7 presents some conclusions drawn from this work.

$\boxed{2}$ $\mathcal{M}$ and $\mathcal{L}$

The non-traditional nature of the $\mathcal{L}$ processor is most evident in its object-oriented memory system. This chapter first examines the difference between an object-oriented memory and a traditional memory. It then takes a general look at the $\mathcal{L}$ processor, and concludes with a formalization of the interface between $\mathcal{M}$ and $\mathcal{L}$.

2.1 Memory Models

The *memory model* of a processor is defined by the interface between the processor and its memory system. This interface is characterized by two sets: a set of operations and a set of objects. The following sections describe the processor/memory interfaces of the traditional memory model and the object-oriented memory model in terms of the respective sets of objects and operations.

2.1.1 Flat Address Space Model

From the viewpoint of the processor in a traditional (von-Neumann) computer architecture, memory is organized as a large array of equally accessible words. During the execution of code, the processor reads from the memory array by producing addresses and writes to the memory array by producing addresses and data. There is nothing present in this model that distinguishes addresses from data; addresses are simply bit strings. The processor can thus meaningfully modify an address to obtain another address. For example, consider the program counter (PC) and stack pointer (SP) registers found in many processors. After an instruction fetch the PC (an address) is incremented (an arithmetic operation) so that the new PC contents are the address of the next instruction. Stack pushes and pops use the SP in a similar fashion. Processors can create addresses from arbitrary pieces of data by using register indirect addressing modes. The traditional processor/memory interface is built upon two operations and two objects:

Operations:

(Write address value): stores value at address.

(Read address): retrieves the value previously stored at address.

Objects:

address: an arbitrary fixed-length bit-string

value: an arbitrary fixed-length bit-string

Both processor and memory are free to take measures (caching, for example) to improve performance so long as the interface is not compromised. We will use the term ***flat address space*** to describe the memory model characterized by the operations above.

In reality, there are a few other important operations present in a flat address space memory system because of exceptional conditions and protection schemes. For example, the response of a Write operation might be a "non-existent memory" error, a "writing to read only storage" error, or a protection violation error. These operations mainly serve to detect malfunctioning programs or to prevent hostile programs from accessing protected information.

The flat address space model has many virtues. It is very simple to implement. Addresses can be operated upon by the processor's ALU and can be stored in the same registers as data. This simplicity and uniformity leads naturally to good performance. The addressing modes available to the processor allow for convenient and efficient implementation of many common operations. For example, local variables and arguments to a procedure can be accessed by adding an offset to the stack frame pointer to produce a new address and subroutine calls and returns can be handled by using register indirect addressing. On the memory side, locality of reference in the address stream can be exploited by using caches and virtual memory systems to improve response time. Flat address space machines are good at supporting programming in languages like FORTRAN, where much of the memory allocation is either done statically, by a compiler, or dynamically, via a stack; both give good locality of reference. This leads to good memory system performance and hence good program performance.

In more dynamic programming models, the efficiency of less structured dynamic allocation and deallocation becomes extremely important. Stacks are optimum for a block-structured, single thread of control model of computation, but more general control models need more flexibility, and must use heap allocation. On a processor with a flat address space and a typical paging system, efficient garbage collection is a serious problem. The garbage collection process accesses memory with little locality of reference, causing the paging system to thrash, reducing performance drastically.

The fundamental problem in implementing a dynamic system on a flat address space is that there are two conflicting carriers of locality: the pages, visible to the virtual memory system, and the objects, visible to the garbage collector. One solution, discussed further in chapter 3, is to carefully design the garbage collector so that it works well with a traditionally designed virtual memory (this is the solution adopted by LISP machines). $\mathcal{M}$, on the other hand, attempts to solve this problem by integrating the virtual memory and garbage collection systems: in $\mathcal{M}$ the only carrier of locality is the object. This leads naturally to an object-oriented model of memory.

2.1.2 Object-Oriented Model

An alternative to a flat address space is to create a processor/memory interface in which addresses appear as atomic objects to the processor. This can be thought of as a strongly typed version of the flat address space interface — addresses are now typed objects, and the set of operations that the processor can perform on addresses is limited. In this model, addresses can be used in read and write operations, but cannot be modified by the processor. Since the processor cannot modify addresses, the memory system can interpret addresses more freely. In particular, addresses can now refer to *objects* of possibly varying size (hence the *object-oriented* model). The underlying representation of memory is still likely to be a flat address space, but by restricting the processor-memory interface, the memory can now hide the low-level implementation details.

Objects typically have some internal structure which must be accessible to the processor. Each object is composed of some number (zero or more) of elements, some of which can take on addresses as values. As part of the interface, the memory needs to provide *element-read* and *element-write* operations. Since the low-level details of storage are hidden from the processor, the memory also needs to provide a new-object operation. Thus, an object-oriented memory/processor interface is characterized as follows:

Operations:

(Elt object-ptr field): reads the field element of object.

(SetElt object-ptr field value): establishes value as the value of the field element of object.

(NewObj type): creates a new object based upon type. Returns the address of the new object.

Objects:

object-ptr: an immutable pointer to a block of storage.

field: a numeric offset into the body of an object.

type: an object or scalar that describes a kind of object

There are several schemes that can be used to guarantee the immutability of address objects. For example, the memory system can associate a tag with each element in an object indicating whether or not it is an address. This tag can be checked by processor hardware to prevent any proscribed operations on address objects. As with the flat address space model, it is possible to have malformed operations — (Elt 0 7) for example. The address tag can be used to detect some of these errors (in this case hardware can tell that 0 is a scalar and not an address).

In a typical object-oriented memory system, Elt and SetElt require address translations similar to those performed by a virtual memory system of a flat address space processor. The address of an object is typically used to produce an index to a mapping table, which provides the actual address of the object. With special purpose hardware, Elt and SetElt should therefore be about as fast as Read and Write. However, none of the object-oriented memories described in this thesis use special object addressing hardware; for these systems, Elt and SetElt suffer from additional levels of indirection in the critical read and write paths, and are somewhat slower than Read and Write. This slowdown is an artifact of implementation and not an inherent property of object-oriented memories.

Object-oriented memory systems have some distinct advantages. The level of indirection in object access gives the memory management system the freedom to dynamically relocate

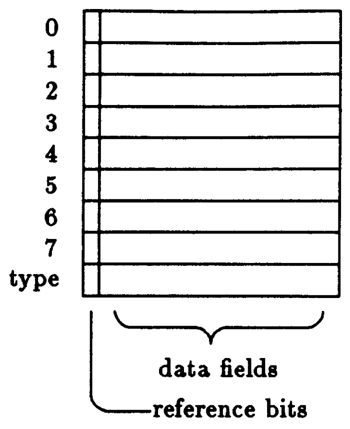


FIGURE 2-1: A CHUNK

objects in order to enhance performance. The memory system can perform grouping of objects into larger objects that can be treated as cohesive units by lower levels of the storage hierarchy. Object addresses can be cached: successive Elts or SetElts on the same object do not all need to suffer the full address translation penalty. Object-oriented memory systems can perform garbage collection on their own without the need for processor support or intervention, if object references are made easily identifiable (by either their location in an object, or by tag bits). Because of the unforgeability of object pointers, object-oriented systems can form the substrate for secure capability-based information sharing [Fabry74].

2.2 The $\mathcal{L}$ Memory

The $\mathcal{L}$ memory system is object-oriented and fits roughly into the framework described above. Memory in $\mathcal{L}$ is divided into individually accessible, identically sized objects called ***chunks***, composed of a few hundred bits of storage. Each chunk (see figure 2-1) is subdivided into nine visible ***slots***, each of which is at least large enough to hold a chunk identifier (ID); each slot has a one bit tag (the reference or ***ref*** bit) indicating whether that particular slot holds a chunk ID or scalar data. One of the visible slots of each chunk is reserved for holding type information, which provides a framework for interpreting the contents of the other slots of the chunk. The other eight visible slots hold data — either chunk IDs or scalar values. Chunks also contain some hidden attribute bits and invisible slots whose use will be described later.

As befits addresses in an object-oriented memory system, chunk IDs do not inherently imply anything about where the corresponding storage is located. IDs are not pointers in the strict sense, but a low-level abstract data type. The processor can only perform a restricted set of operations on IDs: dereferencing (the Elt and SetElt operations), comparison (are these IDs the same?), destruction, and change of properties (Lock, Unlock). The processor cannot create IDs or modify existing IDs. The integrity of chunk IDs is guaranteed below the machine instruction level by the existence of the ref bit. The interpretation of IDs is left strictly up to the memory management system.

2.2.1 Chunk Size

The nine slot chunk (eight data slots and one type slot) was chosen as a compromise between several conflicting desires. Having the number of data slots be a power of two is useful in creating easy-to-access chunk structures. For example, arrays in $\mathcal{L}$ are mapped onto trees of chunks. The access path to a given element of the array can be extracted from the binary representation of the index by masks and shifts, if the branching factor of the tree is a power of two. Figure 2-2 sketches an array accessing operation based upon this idea. If each chunk carries type information, run-time type checks can often be performed without explicitly passing type information. Typed chunks are also convenient for inspectors and debuggers. Thus, a chunk with $2^N + 1$ visible slots seems useful.

Small chunks, with fewer slots, would impose too many levels of indirection when used in implementing medium or large sized objects. Since instruction streams must be packaged into sequences of chunks, small chunks would require frequent inter-chunk jump instructions, which would be inefficient. The amount of overhead required for chunk-based memory management (the invisible slots and hidden bits) is independent of chunk size, so using smaller chunks would increase the percentage of memory space devoted to this overhead. To provide support for the exploitation of fine-grain parallelism, we also want chunks to be large enough to encapsulate the state of a thread of control.

Large chunks would suffer from internal fragmentation when implementing small objects. Most objects in object-oriented systems are fairly small. Mean object size is 70 bytes in the SOAR implementation of SMALLTALK-80 [Ungar84], and 100 bytes in the Explorer LISP system [Courts87].[1] Besides being a waste of memory space, internal fragmentation reduces

[1] When measuring object size, the *median* is probably more meaningful than the mean, since there tend to be a small number of enormous objects, like bitmaps, that distort the mean.

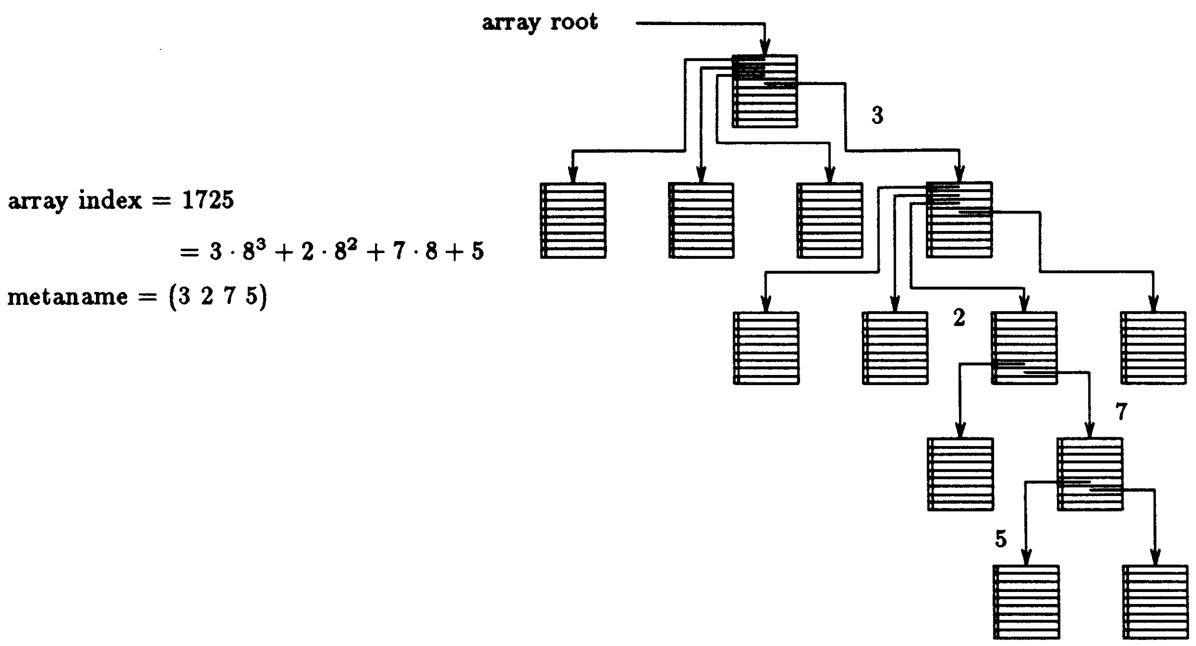


FIGURE 2-2: ARRAY INDEXING IN A CHUNK TREE

the useful object density in memory, leading to reduced efficiency in the object management system, since empty slots in useful objects take up space that could be occupied by other useful objects. Thus, it is important to have a good match between chunk size and object size.

2.2.2 Building Objects Out of Chunks

Chunks provide a uniform substrate for the implementation of objects in $\mathcal{L}$. In particular, STATE chunks represent active and suspended processor states. A collection of STATE chunks (or simply STATEs) describes a set of control threads, each of which contains all of the information needed to resume its execution. All other data objects are also built out of various chunk structures: arrays, environments, code streams, lists, etc. Most objects built from chunks are created from a *template*. The template, which is typically stored as part of the type information, provides a framework for interpreting the object.

To access elements of multi-chunk objects, we frequently need to specify a sequence of fields to use in successive Elt operations. For example, figure 2-2 shows one way to produce the sequence of field names needed to perform an array reference if the array is mapped onto an 8-way branching tree of chunks. The sequence of Elt instructions required is:

(Elt (Elt (Elt (Elt root-chunk 3) 2) 7) 5)

where root-chunk is a pointer to the top chunk in the array tree. As a shorthand for this sequence of Elt operations, we concatenate the field names (or numbers) into a list. Such a list of field names, (3 2 7 5) in this example, is called a *metaname.*

The Elt and SetElt operations can be generalized into MetanameFetch and MetanameStore, which take metanames as operands. The sequence of Elts above becomes:

(MetanameFetch root-chunk (3 2 7 5)).

A MetanameStore operation is composed of a sequence of Elts followed by a SetElt. To write a new value V into the 1725th element of the array in figure 2-2, we need to perform the following sequence of operations:

(SetElt (Elt (Elt (Elt root-chunk 3) 2) 7) 5 V).

This is written in terms of MetanameStore as

(MetanameStore root-chunk (3 2 7 5) V).

Because of the difference in the number of Elts, metanames play a slightly different role in the MetanameFetch and MetanameStore operations.

As noted before, all objects in $\mathcal{L}$ are built from chunks. Figure 2-3 shows an $\mathcal{L}$ implementation of a control stack for a block structured language, with one STATE chunk per "stack frame." In this example, each STATE contains four pointers: a pointer (C) to the code associated with that state, a pointer (P) to a subordinate state (a subroutine), a dynamic link (D), and a static link (S). Control resides in the lowest STATE. When it has finished its computation, it will awaken its successor STATE by using its dynamic link. Non-local variables are accessed through MetanameFetch and MetanameStore operations, where the metanames are constructed so as to follow the static links.

2.3 The $\mathcal{L}$ Processor

The $\mathcal{L}$ processor can be logically divided into three portions, each concerned with a different fragment of the processing task: the execution unit, the state management unit, and the memory management unit ($\mathcal{M}$). This section provides some general information about the execution and state management units.

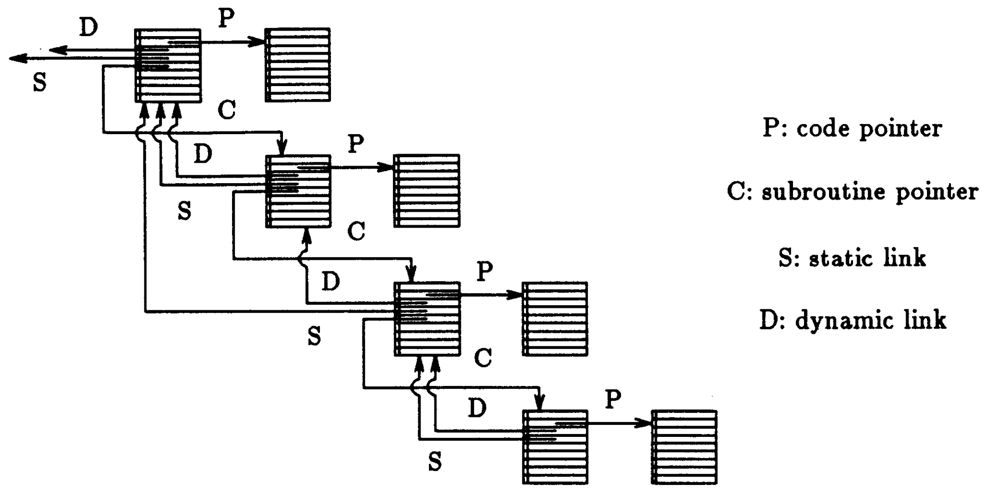


FIGURE 2-3: CONTROL STACK BUILT OUT OF CHUNKS

2.3.1 Execution Unit

The execution unit EU of the $\mathcal{L}$ processor is a simple finite-state machine that accepts the ID of a STATE as input, reads an instruction out of the STATE, and causes the operation specified in the instruction to be performed. The operations range from performing arithmetic on values in the STATE to creating a new runnable STATE. The execution unit of the $\mathcal{L}$ processor has no explicit internal state save for the single ID of the currently executing STATE; all machine state is explicitly stored in STATE chunks. The execution unit is thus only responsible for the actual execution of instructions.

Part of the state information stored in a STATE chunk is a pointer to a chunk full of EU instructions, called a *code* chunk, and a slot (the *microstate*) that records, among other things, which slot of the code chunk contains the next instruction to execute. The code chunk and microstate thus specify the control state of the STATE chunk. Conceptually, the EU can fetch instructions from the code chunk by the following sequence of operations:

(MetanameFetch (MetanameFetch state (1)) state (MetanameFetch state (0)).

Here state is the ID of the current state chunk, and we have followed the convention that the microstate is kept in slot 0 of the STATE, and a pointer to the code chunk is kept in slot 1.

Table 2.1 lists some EU instructions. This table is by no means complete, but provides some examples of the kinds of instructions the EU can execute. These instructions are based upon Metaname operations. For example, the EU instruction

Instruction	*Description*
`(move (ms) (md))`	move
`(movei val (md))`	move immediate
`(add (ms) (md))`	arithmetic operation
`(xor (ms) (md))`	logical operation
`(jump offset)`	intra-code-chunk jump
`(jump chunk offset)`	inter-code-chunk jump
`(jgt chunk offset)`	conditional jump
`(call (m))`	inter-state control transfer
`(return (m))`	inter-state control transfer
`(activate (m))`	create new runnable state
`(lock (ms) (md))`	gain exclusive access
`(test (m))`	set condition codes
`(alloc (m))`	allocate a new chunk

`(m)`:	operand metaname
`(ms)`:	source operand metaname
`(md)`:	destination operand metaname
`chunk`:	chunk ID
`offset`:	instruction number
`val`:	immediate value

Table 2.1: SOME EU INSTRUCTIONS

```
(move (3 2) (3 3))
```

is equivalent to

(MetanameStore state (3 3) (MetanameFetch state (3 2))).

All metanames in EU instructions have a default root chunk — the current STATE chunk.

2.3.2 State Management Unit

The $\mathcal{L}$ processor's state management unit (SMU) keeps track of STATEs. The SMU remembers which STATEs are runnable via a set of ***prerequisites*** associated with each STATE. These prerequisites indicate either that the STATE is runnable or that the STATE is potentially runnable after some set of events (for example, unlocking of a shared resource, importation of a shared object by $\mathcal{M}$, or arrival of a hardware interrupt). Instructions processed by the execution unit can cause modifications in the prerequisites of the executing STATE or other STATEs. For example, the `activate` instruction is actually a message from the EU to the SMU which informs the SMU that there is a new runnable STATE.

The execution unit executes instructions from its current STATE until either the current STATE becomes non-runnable or exhausts its processing quantum. STATEs can become non-runnable for a variety of reasons — they may complete their processing task, be blocked by access to a locked shared resource, or cause a fault or exception. Exceptions are raised by type violations (taking the Elt of a non-chunk ID, for example) or by object faults (the object-oriented equivalent to page faults) while attempting an Elt. When the current STATE becomes non-runnable, the SMU passes the execution unit the ID of a new runnable STATE, and execution commences in this new STATE.

Since all state information is explicitly represented in chunks, context switches are straightforward. The state of a process does not include the state of the memory management unit, which is independent. Thus context switches conceptually involve changing just one pointer — the ID kept in the EU. Of course, the EU is free to cache the contents of chunks so long as it does not violate the memory interface, so actual context switches may involve flushing and reloading these chunk caches.

2.4 The $\mathcal{L}$ — $\mathcal{M}$ Interface

The interface between an $\mathcal{L}$ processor and the $\mathcal{M}$ system is basically the same as the one described for an object-oriented memory system in section 2.1.2. The interface has been extended somewhat to allow the processor to specify chains of accesses by using metanames. In addition, the interface has been extended to include operations that will help $\mathcal{L}$ support synchronization of multiple threads of control. These operations are based upon chunk *attributes* (some of the hidden bits mentioned above) that are modifiable only in certain controlled ways. Mutual exclusion is accomplished by the Lock and Unlock operations.

The details of the $\mathcal{L}$ processor/memory interface are summarized below:

Operations:

(MetanameFetch chunk-id metaname): read the value described by the metaname mn starting from initial chunk chunk-id.

(MetanameStore chunk-id metaname value): write a value to the chunk slot described by metaname and chunk-id.

Lock(chunk-id): set the lock attribute in the chunk pointed to by chunk-id.

Unlock(chunk-id): clear the lock attribute.

NewChunk(type): allocate a new chunk with type type, and return its ID.

Objects:

metaname: a sequence of field indices.

chunk-id: an immutable chunk address object.

type: a scalar or the ID of a type object.

The **Lock** operation takes a chunk as an operand. It sets the lock attribute of the chunk and returns a key as a result. Future accesses to that chunk will be blocked unless the accessor has used the key. The key can be freely copied and distributed. Eventually, a STATE holding a key will **Unlock** the chunk, at which time all keys revert to normal IDs, and all STATEs blocked previously are restarted.

$\boxed{3}$ The Foundations of $\mathcal{M}$

One of the goals of the memory system is to provide the processor with the illusion of near-infinite capacity at reasonable speeds. Since the actual capacity of the memory system is finite, the memory system must manage its limited resources carefully. Two important memory management tools are ***virtual memory***, the management of interesting and uninteresting objects or locations, and ***garbage collection***, the management of accessible and inaccessible objects or locations. The memory system is also responsible for providing the processor (and hence the programmer) with a clean and correct memory model, whether a flat address space or an object-oriented model.

$\mathcal{M}$, an object-oriented memory system, is designed to fulfill the above goals of performance and correctness. This chapter provides a look at the foundations of $\mathcal{M}$: concepts from virtual memory, ephemeral garbage collection, and object-oriented storage.

3.1 Virtual Memory

Modern machines can address large amounts of memory, but only a few locations at a time. Execution of a typical instruction may require 2 or 3 memory operations. Thus, over the span of a few instructions, the processor can only refer to a small number of distinct addresses. Virtual memory techniques rely upon the ability of the memory system to predict the addresses of future memory accesses based upon the addresses of past accesses.

A virtual memory system usually has two types of memory resource at its disposal: a fast, medium-sized ***primary*** memory, and a slow, large ***secondary*** memory. By mapping the set of addresses expected to be accessed into the primary memory, the virtual memory system can simulate a memory as large as the secondary memory with a speed close to that of the primary memory. The actual apparent memory speed depends upon the fraction of the time a memory request can be satisfied from primary memory. This fraction is known as a *hit ratio*.

3.1.1 Locality

In most circumstances, the address stream produced by the processor is far from random. The short-term address density distribution is highly non-uniform. The tendency of accesses to be clustered together is usually described in terms of two characteristics of the address stream: *spatial locality* and *temporal locality* [Smith82].

Spatial locality, or *locality of reference*, refers to the tendency of addresses in an address stream to be clustered in small ranges. In a flat address space system, where numeric operations on addresses have meaning, locality of reference implies that the distribution of addresses is concentrated around a small number of locations. The unit of locality in most virtual memory flat address space systems is a *page*, made up of some number of consecutive virtual memory locations. In an object-oriented system, locality of reference can be measured by the spread in the number of levels of indirection between objects. Given an initially reachable set of objects, accesses will tend to be concentrated on those objects, and objects referred to by those objects, etc. In an object-oriented system, objects are the unit of locality.

Temporal locality refers to the fact that the "hot spots" of memory access change relatively slowly with respect to the number of accesses. In other words, addresses used for current accesses are likely to be used again in the near future, or, from an object-oriented viewpoint, the objects currently being accessed are likely to be accessed again in the future. Temporal locality allows the virtual memory system to extrapolate likely addresses of future memory accesses from a sequence of past addresses.

3.1.2 Working Sets

The set of pages[1] specified by an address stream constitute the *working set* of the process that the address stream is associated with. The pages in the working set are the ones that should occupy primary memory, since they are likely to be the target of accesses by the processor. In flat address architectures, a set of *page tables* is used to record which pages

[1]The discussion of virtual memory concepts that follows will describe flat address space systems, and hence use the term *page* to describe the carrier of locality, but the ideas presented are equally applicable to objects and object-oriented systems.

are actually present in the primary memory. Object-oriented memories use various schemes (described further in section 3.4) for recording the same information.

Because of temporal locality, the working set changes over time. Reads or Writes to addresses not paged in cause *page faults*; at this time the memory system must update the working set to include the new page. If the primary memory is not full, handling a page fault is simple: the page in question is simply copied in from the secondary memory, and the page table is updated. If the primary memory is full, then some kind of page replacement algorithm must run to decide which pages should be returned to the secondary memory to create space in the primary memory. The goal of the replacement algorithm is to determine which page or pages currently in primary memory can be removed with the smallest impact on performance. This is typically done by keeping track of the access history of each page. Temporal locality dictates that the best page to replace is the one least recently accessed, and this forms the basis for the LRU page replacement algorithm. There are many other page replacement algorithms — FIFO, random, and clock, to name a few [Baer80].

Page replacement algorithms need to keep a small amount of extra information about each page; this information can be bundled up with the page itself. For example, each page has a *dirty bit*, indicating whether or not the contents of the page have changed. A page's dirty bit is set when a Write operation is performed to an address on that page. Page replacement algorithms typically prefer to pick clean pages for replacement, since the contents of a clean page do not have to be written back to secondary memory before the primary memory allocated to the page can be reused.

3.2 Garbage Collection

The goal of virtual memory is to populate the primary store with objects that the processor might access. The goal of garbage collection is to populate memory with objects that the processor *can* access (or conversely, to reclaim storage that is inaccessible so that it can be used over). Proper garbage collection requires that the collector *prove* that an object is inaccessible before reclaiming the space used by the object (or else the object-oriented storage abstraction will break down). Garbage collection can be painfully expensive, especially in the large address spaces provided by virtual memory. Steele [Steele75] indicates that large LISP programs can spend as much as 40% of their execution time in garbage collection.

Garbage collection is only necessary in languages that allow for dynamic object allocation (the primary example being LISP). Static programming models have no need for garbage collection, because there is no way for a location initially addressable to become unaddressable. Virtual memory thus suffices as the sole memory management policy for static languages. In more dynamic models, objects eventually become inaccessible, because pointers to objects get destroyed. Garbage collection is the process of discovering and reclaiming these dead objects. As we will discuss in detail in section 3.3, garbage collection and virtual memory interact very strongly in implementations of dynamic programming languages. First, however, we will launch into a discussion of the techniques of garbage collection.

3.2.1 Traditional Methods

Common to all object-oriented systems (whether at the language level, or lower) is the notion of an object's *accessibility*. An object is accessible if a pointer to that object resides in some other accessible object. This recursive definition bottoms out because a certain root set of objects are accessible by default. To simplify matters, we can imagine that all the root set objects are pointed to by a single object, the *root* object, which may or may not actually exist. An object is thus accessible if the object is reachable from the root object through some sequence of pointer dereferences.

Mark/Sweep GC

By starting at the root object, traversing the graph of accessible objects, and coloring (or marking) each as we pass through, we can visit all accessible objects. This simple process is called a *mark*. The mark traces out a tree of objects (whose root is the root object), since we use coloring to keep from following cycles in the object graph. After a mark has been performed, we can then examine all objects in existence (a *sweep*). Those that are unmarked are inaccessible, and hence garbage. These two simple passes make up Mark/Sweep garbage collection. A simple LISP description of this algorithm, in which the control stack is used to keep track of the state of the mark, is given in Figure 3-1. By defining a stack-valued variable to hold the mark state, the mark procedure can be made iterative.

The traversal of the object graph may require a block of temporary storage about as

```
(defun mark (root)
   (set-mark-bit-in root)
   (do-field (f root)
       (and (pointer? (elt root f))
            (unmarked? (elt root f))
            (mark (elt root f)))))

(defun sweep ()
   (do-all-objects (obj)
       (when (unmarked? obj) (reclaim obj))))
```

FIGURE 3-1: MARK/SWEEP ALGORITHM

large as the number of objects (or, in a recursive implementation, stack space large enough to hold about as many stack frames as there are objects).[2] This is because each object can typically contain many object references. For example, a chunk can contain 9 other chunk IDs. As we initially visit objects, the number of objects on the mark stack can grow quickly. Eventually, most of the object references in an object will be to objects that are already marked, and the mark stack will stop growing.

The mark/sweep algorithm makes no intrusions on the processor model; it depends only upon the existence of the object graph. Mark/sweep garbage collection is typically done in "batch mode," run when the storage system runs low on space. To implement mark/sweep garbage collection, each object must be able to accommodate a mark bit.

Reference Counting

As an alternative, we can try to keep track of the copying of pointers to objects by associating a count of the number of pointers to a given object with that object. Every time a pointer is copied, the pointer is dereferenced, and the object at the other end has its count incremented. Before a pointer is destroyed, it is dereferenced, and the object at the other end has its reference count decremented. When the reference count of an object reaches zero, the object is inaccessible, and its storage can be reclaimed. Reclamation involves first decrementing the reference count of any objects referred to by the newly deceased object (which, if it reduces any of their respective reference counts to zero, can lead to *recursive freeing*). Then the storage used by the deceased object can be reused. A simple

[2]There are some elaborate marking algorithms that hide the mark stack in the object graph by reversing pointers, and use only a small amount of extra storage. See [Cohen81] for details.

```
(defun setelt-with-ref-count (new-thing in-obj in-field)
   ;; first, see if we are destroying a pointer
   (when (pointer? (elt in-obj in-field))
      (decrement-ref-count-of (elt in-obj in-field)))
   ;; now, see if we are creating an extra copy of a pointer
   (when (pointer? new-thing)
      (increment-ref-count-of new-thing))
   ;; actually install value
   (setelt new-thing in-obj in-field))

(defun decrement-ref-count-of (object)
   ;; we may have already reclaimed this object
   (and (not-reclaimed object)
        (zerop (decf (ref-count-of object)))
      (do-fields (f object)
         (when (pointer? (elt object f))
            (decrement-ref-count-of (elt object f))))
      (reclaim object)))
```

FIGURE 3-2: REFERENCE COUNTING ALGORITHM

implementation of reference counting is given figure 3-2.

As opposed to the non-intrusive nature of mark/sweep garbage collection, reference counting requires a "hook" into a basic operation — SetElt. As with the mark, the decrement reference count procedure can be made iterative if the set of objects that need their reference counts decreased is kept in a single variable instead of in the control stack. Reference counting cannot reclaim circular structures like the doubly-linked control-stack of figure 2-3, because even if the entire stack itself is not pointed to from the outside, the inter-stack references give each chunk a non-zero reference count. In contrast to mark/sweep, reference counting is a reasonably incremental garbage collection method — objects are freed as soon as possible instead of accumulating until space runs low. To implement reference counting, each object must now be able to accommodate a count field several bits wide.

3.2.2 Ephemeral Garbage Collectors

As with virtual memory, which keeps track of which pages are likely to become uninteresting in the near future, there are garbage collection methods which keep track of which objects are likely to become garbage in the near future: *ephemeral* or *volatility-based* garbage collectors [Liberman83].

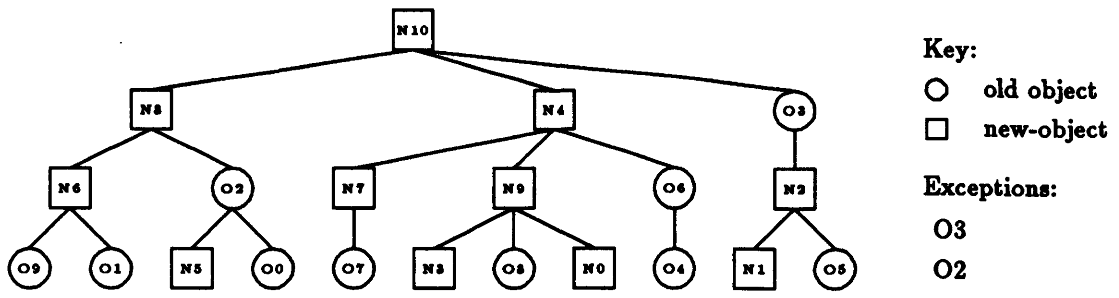


FIGURE 3-3: SEGREGATED OBJECT TREE

Intuition, heuristics, and statistical studies of dynamic programs have shown that the objects most likely to become garbage are newly allocated objects. In addition, new objects are much more likely to point to old objects than old objects are to point to new ones.[3] Thus, new objects occupy, for the most part, the upper portions of the object tree that would be traced out by a mark. There are a few exceptions, that is, a few new objects in the lower portions of the tree, which are pointed to by old objects. Pointers from old to new objects are called backwards pointers, since they point in the opposite direction from the majority of pointers. As long as there aren't many backwards pointers, their existence can be noted in a special exception (or entry) table.[4] Figure 3-3 illustrates a hypothetical object tree with two classes of objects, old and new, and an associated exception table.

Because the new objects occupy the upper portions of the object tree, we can perform a limited-depth mark which will be assured of marking most of the new objects. By consulting the exeception table, the new objects that are low in the tree can be marked as well. A limited-breadth sweep over the new object area can then confidently reclaim any unmarked new objects as garbage. Because of the segregation of objects, we can collect new objects without having to traverse the entire object tree. The proof of garbage collector correctness here relies on a volatility assertion: no old object refers to a new object except as noted in the exception table.

The example given above has only two volatility levels, but the idea of segregated storage can easily be extended to multiple levels. In this case, as we progress down the object tree, we pass through zones (or generations) of increasingly more stable (less likely to become

[3]This observation comes from LISP, where the basic object creating function is cons. Cons cells start out with initial contents, which must necessarily be at least as old as the cell itself. Pointers from old to new objects are created by mutation functions like rplaca or rplacd.

[4]This table can be distributed among special backwards pointer objects, as is done by the TGC on the TI Explorer (see section 3.4.1).

garbage) objects. The objects at the top of the tree are temporary, and the objects at the bottom are static storage. Objects in between are of intermediate volatility. To collect objects on a given level, we can set the mark depth and sweep width to a given generation.

As an new object survives successive garbage collection cycles the garbage collection system can begin to suspect that the object may be more permanent than its age would indicate. Aging is the process of updating the volatility gradations in storage. As objects mature, they can be *promoted* or moved to levels of more stable storage. A related issue is where new objects should be created. Most objects should be created on the highest volatility level, but there are some occasions (compilation, for example) where the objects being created are known to be static. This knowledge can be used to save the aging system the work of promoting the objects in question from temporary storage out to static storage.

An implementation of an ephemeral gc algorithm requires a storage check similar to that done in reference counting. When the processor stores an object reference (to object A, for example) inside of another object (B), it must check to see if B is older than A, and if so (a *volatility fault*), place a note of that fact in the appropriate exception table. The volatility of objects can be stored in the objects themselves, but it is usually more convenient to keep different generations of objects in different regions of memory, so that the volatility of an object is deducible from a pointer to that object.

3.3 Interaction between Virtual Memory and Garbage Collection

Garbage collection causes additional memory cycles without advancing the states of any ongoing computations. Unless an implementation is careful in arranging the pattern of these additional cycles, they can cause problems for the virtual memory system. Memory accesses initiated by the garbage collector may cause object faults, displacing objects from the working sets built up by the current processes. These objects will then have to be faulted back into the primary memory when the processes access them later. This type of interaction between the garbage collector and the virtual memory system is called *thrashing*, and it can greatly reduce the efficiency of the memory system.

Consider an example of mark/sweep garbage collection, running on top of a paged virtual memory system. A single process, A, is running on the machine, and issues a request for a new object. The allocation system discovers that there is not enough room, and so initiates garbage collection to free up space. At this point, the primary memory is populated with pages from the working set of A. The mark then begins traversing the object tree. There is little guarantee that the objects of interest to the mark initially (those near the root object) will be paged in, so the mark will cause many page faults. During the mark phase, every accessible object must be in primary memory at some time or another. Each object will be accessed as many times as there are pointers to that object (see the implementation of mark in Figure 3-1) because of the need to check the state of the mark bit. These accesses may occur in various orders depending upon the exact nature of the object graph traversal (i.e. depth first or breadth first) and have little of the temporal or spatial locality that the virtual memory system is depending upon. The memory system thrashes. The sweep phase is more regular, since objects are accessed in a fixed order, presumably in order of increasing address (since, in reality, we are mapping objects onto a flat address space). At the end of a mark/sweep cycle, the set of objects in primary memory will bear little resemblance to A's working set, and A will have to fault its working set back in.

Reference counting has a less violent effect on the working set of the ongoing process, since a typical SetElt operation will result in only a few extra memory cycles. However, destroying the last reference to a large subtree of objects (thereby causing lots of recursive freeing) will cause thrashing of the sort seen under the mark phase of mark/sweep, since the recursive freeing will access every object in the subtree.

3.3.1 Compacting Garbage Collectors

Implementations of object-oriented systems that are built on top of page-oriented virtual memories can suffer from more problems than just thrashing. These problems stem from the fact that a paged, object-oriented system has two possibly conflicting carriers of locality: objects and pages. There is no guarantee that objects on the same page refer to each other.

Suppose that we start with a clean page and allocate a set of objects onto that page. After a while, some of the objects on the page will have become garbage, and the space they were using will be allocated to other objects. If we repeat this gc/allocation cycle a few times, the page becomes populated with objects from various generations (and empty holes) that may have nothing in common with each other (that is, they reside in non-intersecting working sets). Thus, the fraction of the page that is devoted to objects in any one working set decreases over time — the working set becomes spread over a larger and larger base set of pages. Since the primary memory stays a constant size over the same time period, the number of objects in the working set that can be in primary memory at the same time decreases.[5] The decrease in interesting object density causes a rise in page faults, slowing down the memory system.

3.3.2 Simple Compaction

One method of combating this spread of the working set is to garbage collect with compaction. The principal algorithm used is the *semispace* algorithm [Fenichel69]. The semispace algorithm divides memory up into two halves, oldspace and newspace. Initially, all objects reside in oldspace. When oldspace becomes full, *scavenging* begins putting objects into newspace. Scavenging begins by copying the root object from oldspace to newspace. The oldspace copy of the root is replaced by a *forwarding* pointer which points to the newspace copy. A scavenging pointer is set at the base of this new root object, which is presumably full of pointers to objects in oldspace, and a free space pointer is set to just past the root object. A scavenging step consists of examining the field pointed to by the scavenging pointer. If the field contains a pointer, then this pointer's oldspace attribute is checked. If it points to oldspace, then the object pointed to is copied to newspace (unless it is a forwarding pointer), at the location indicated by the free space pointer. The scavenging pointer is then advanced one location, and the free space pointer is advanced past the end of the object just copied. After some number of scavenges, the scavenging pointer will have progressed through all the fields of the of the root object. At this point it will fall off the end of the root object onto the start of the first copied object. The semispace method thus relies upon the mapping of objects on an underlying flat address space to keep track of objects that still need to be scavenged. Eventually, the scavenging pointer will catch the free space pointer. At this point, no more references to oldspace will exist in accessible objects, the roles of oldspace and newspace are flipped, and the accessible objects are packed end-to-end in the pages of newspace.

This algorithm can be run in batch mode as described above, but its typical use is in the Baker real time algorithm [Baker78], which performs garbage collection incrementally. In this case, new object allocation and object accesses (Elts and SetElts) are going on at

[5] See, for example, [White80].

```
(defun new-object (size)
   (let ((new-object-id new-object-pointer))
      (incf new-object-pointer size)
      (do-some-scavenging)
      new-object-id))

(defun transport (object)
   (let ((newobject (copy-object-to-newspace object)))
      (setf (forward-value object) newobject)
      newobject))

(defun elt-for-baker-gc (object field)
   (let ((thing (elt object field)))
      (if (oldspace? thing)
          (then
             (if (forwarded? thing)
                 (then  ;; pointer snap
                    (setelt object field (forward-value thing))
                    (forward-value thing))
                 (else  ;; pointer update
                    (let ((newthing (transport thing)))
                       (setelt object field newthing)
                       newthing))))
          (else thing))))

(defun setelt-for-baker-gc (object field value)
   (if (oldspace? value)
       (then (if (forwarded? value)
                 (then (setelt object field (forward-value value)))
                 (else (setelt object field (transport value)))))
       (else (setelt object field value))))
```

FIGURE 3-4: BAKER REAL TIME GC ALGORITHM

the same time as scavenging. New objects are allocated in newspace at the location of a new object pointer.[6] Elts and SetElts require *transporter* tests to insure that no oldspace pointers are returned or stored in newspace. A pidgin LISP implementation of this algorithm is given in Figure 3-4. By scavenging a few objects each time an object is allocated, the rate of reclamation (as determined by scavenging) can automatically track the rate of allocation.

The Baker algorithm requires an extra attribute bit in each pointer, so that the forwarding pointers can be distinguished from normal object pointers. Otherwise, the storage requirements are similar to those of the semispace algorithm. By installing an intelligent object scavenger, we can adapt this algorithm to handle objects of various sizes and com-

[6]The implementation in 3-4 assumes new objects are created empty. A minor modification is required if, as in the case of a LISP cons cell, objects are created with initial contents.

positions – the scavenger can examine objects on an individual basis to determine where pointers might be stored. This increases the scavenging efficiency on objects like arrays of integers, which will not contain many pointers.

3.3.3 Dynamic Compaction

The above methods may help with the problem of external fragmentation, and make it easy to allocate free space, even in the presence of variable-sized objects, but they do not completely solve the problem of the spread of the working set. The semispace algorithms traverse the *static* object tree, and pay no heed to the *dynamic* patterns of object access. Thus, compacting garbage collectors can increase the fraction of accessible objects on a page, but may not necessarily increase the fraction of objects on a page belonging to a given working set.

There are several methods that can be used to help achieve dynamic compaction: scavenging resident pages of objects preferentially, allocating new objects onto the same page as the objects they refer to, etc. For Baker's incremental garbage collector, White [White80] notes that the principal villain in the degradation of the interesting object density is the scavenger, since it is busy copying accessible (but not necessarily interesting) objects to newspace, where they take up room alongside interesting objects brought together by the transporter. A simple strategy in this case is simply to delay the start of scavenging until the (estimated) working set has been transferred to newspace by calls to the transporter from Elt and SetElt. This provides for dynamic grouping of objects onto pages, and compacts the working set into a small number of pages, even in cases where the working set is made up of large numbers of short-lived objects.

3.3.4 The Object Hierarchy and the Memory Hierarchy

If we step back at this point and examine the distribution of objects created by ephemeral garbage collection, and the distribution of memory in a virtual memory system, there seems to be a very natural correspondence. At the "high" ends of both systems are frequently accessed, dynamic objects; as we move downwards the objects become less interesting and less dynamic. This correspondence is not too surprising, since ephemeral garbage collection and virtual memory are both based upon exploiting properties of non-uniform distributions (object lifetimes or object accesses, respectively). Thus, it seems reasonable that virtual memory and garbage collection could be integrated so that these two hierarchies become more unified. This idea will be explored in more detail in section 4.1.

3.4 Object Oriented Virtual Memories

The last of the three sources of inspiration for $\mathcal{M}$ are the object-oriented virtual memory systems. This section presents four examples. The first is the object-oriented virtual memory system used on the TI Explorer LISP machine. The other three examples are memories designed for various implementations of SMALLTALK, an object-oriented programming system. The first two SMALLTALK systems differ from page-based object-oriented virtual memory systems, like the Explorer system, in that their virtual memory systems deal directly with objects and not pages. Both SMALLTALK and LISP, unlike $\mathcal{L}$, have variable-sized objects. This leads to some additional levels of complexity in the memory system which $\mathcal{M}$ does not have to worry about.

3.4.1 TI Explorer Memory

The temporal garbage collector (TGC) implemented in Release 3 of the TI Explorer LISP system is an ephemeral, dynamically compacting, real-time semispace garbage collector [Courts87]. TGC keeps track of backwards pointers by creating special backwards pointer objects; these special objects are clustered together so that they can be efficiently scavenged.[7] TGC uses 4 volatility levels, numbered from zero (the most volatile) to three (most static). In level three there are actually three different classes of backwards pointer objects — those that point to level zero, those that point to level one, and those that point to level two (similarly, level one has one class and level two has two). Thus to flip level zero storage, TGC needs to scavenge all of level zero's oldspace, level one's level zero backwards pointer objects, level two's level zero backwards pointer objects, and level three's level zero backwards pointer objects. Unused backwards pointer objects can be reclaimed during a flip of the corresponding generation.

Compaction is provided by three mechanisms. The first is due to the traversal of the static object graph by the scavenger. Scavenging is done approximately depth-first (by using

[7]In contrast, the Symbolics 3600 page-oriented object memory [Moon84] handles volatility exceptions by keeping track of which pages contain backwards pointers.

a scavenge stack) which leads to better performance than the breadth first approach of the Baker scavenger. The second compaction technique is an inhibition of the scavenger until a certain amount of newspace has been allocated to new objects or objects moved by the transporter in the course of program execution. This keeps the interesting object density in newspace high. The final mechanism, training, involves keeping track of the activity of each object. This is done by creating a set of activity levels for each generation. When an object is created, or is transported due to program action, it is placed in the highest activity level. When an object is transported by the scavenger, it is moved down one activity level. Objects in less active levels of a generation only need to be brought into memory when that generation is going to be flipped. Training and scavenger action thus weed out accessible but uninteresting objects, increasing the interesting object density on the pages that make up the most active level of a generation.

3.4.2 OOZE

OOZE (Object-Oriented Zoned Environment) is an object-oriented virtual memory for the SMALLTALK-74 and SMALLTALK-76 systems [Kaehler81]. OOZE, implemented in microcode on the Xerox Alto, manages a two-level memory hierarchy using 48K of primary memory and 1M of disk memory. The primary memory is divided into an 8K Resident Object Table (ROT) and 40K of object space.

The ROT keeps track of which objects are currently in primary memory. Object pointers in OOZE contain the disk address of the associated object. To dereference a pointer, a hashing function is applied to the pointer to obtain an entry index into the ROT which is used to obtain an resident object description. This description is compared to the original pointer, with three possible outcomes:

- *Match*: If the pointer part of the resident object description matches the initial pointer, then the address part of the description holds the memory address of the object. The Elt or SetElt can then proceed, using the memory image of the object.

- *Fault*: if the resident object description is invalid, then the object in question is not in primary memory, and an object fault is invoked to bring the object in, and that entry of the ROT is filled with a description of the object and its location in memory.

- *Mismatch*: if the resident object description is in use, but the pointer part does not match, then the hash table lookup has collided. In this case, a rehash function is applied to the original pointer and the lookup is retried at another ROT location.

Eventually, all accesses end in a fault or a match. Since the disk address of the object is encoded in the object pointer, the fault routine need only be passed the pointer to obtain the object.

OOZE divides the set of SMALLTALK objects into pseudoclasses, which are objects of a given SMALLTALK class that all have the same length. Part of the object pointer specifies the pseudoclass of an object. The pseudoclass number is used to index a pseudoclass map which contains the base disk address for the pseudoclass, the actual class of the object, and the length of objects in that pseudoclass. Given the base address, object length, and the instance number (the other bits of the object pointer), the disk address of the object is determined.

OOZE performs garbage collection by reference counting and memory management by the clock algorithm. When an object fault occurs, OOZE scans free memory space for a block of the proper size. If one is found, the object is copied in and and entry is made in the ROT. If no free block large enough is found, OOZE purges some objects from primary memory and invalidates the corresponding entries in the ROT. During purging, the fragmentation of primary memory is checked, and if it exceeds some threshold, then a compaction phase is run. The fault can then be handled. Since pointers specify disk addresses, objects can be freely moved about in primary memory so long as the ROT is suitably updated; this makes compaction simpler.

As mentioned in chapter 2, some object-oriented memories incur performance degradations (with respect to flat address space memories) because of the extra levels of indirection in the Elt and SetElt operations. In the case of OOZE, an object pointer dereference involves hashing followed by table lookup. The OOZE system caches the addresses of frequently accessed objects (the current method, the receiver, and the top of the stack), and is careful about managing collisions in the ROT to try and remove some of the delay from this critical path.

3.4.3 LOOM

LOOM (Large Object-Oriented Memory), the successor to OOZE, was designed for the SMALLTALK-80 system [Kaehler83]. OOZE had demonstrated that swapping objects was viable, but the OOZE address space was too small. LOOM solves this problem by creating a dual namespace: each object id (called an Oop) in LOOM can have has two possibly valid representations, known as the short and long Oops respectively. Short Oops refer to objects that are swapped in. A short Oop is a direct index into the resident object table (as opposed to pointers being hash keys in OOZE) which specifies the object's base address.

Objects that are in main memory can only contain short Oops. In order for one of these objects to refer to an object on secondary storage, one of two mechanisms was used. Objects in secondary storage can be represented in main memory either as a *leaf* or a *lambda*. Leaves are small objects in main memory (hence having a short Oop) that contain the long Oop of the real object and a delta reference count. Lambdas are a reserved short Oop. Leaf pointers can be freely copied or destroyed, but any access to a lambda (or a dereference through a leaf) causes an object fault. In an object fault, the secondary storage copy of the containing object is consulted to recover the appropriate long Oop. This long Oop is then hashed on to produce a short Oop (if the faulting object is a leaf, then the short Oop and ROT entry are already known). This short Oop is used to index into the ROT, and, if the resulting ROT entry is unused, the body of the object is copied into main memory, and the appropriate information is placed into the ROT. Hash collisions are resolved by rehashing until an unused ROT entry is found. Besides the object base address, ROT entries contain a short Oop reference count (which keeps track of how many copies of the short Oop exist), a dirty bit, and an untouched bit.

When an object is copied into main memory, its long Oops are turned into short Oops. This is handled in one of three ways. If objects referred to are already in main memory, then the long Oops are replaced by the proper short Oops. If the objects are not in main memory, then the long Oops are replaced by either short Oops of leaves or the special lambda short Oop. Creation of leaves and lambdas does not require examination of the object in question, and so does not cause further object faults. The decision as to whether to create a leaf or use a lambda is steered by a bit associated with each pointer in the secondary memory copy; this bit records whether the pointer in question was a lambda just before the last time the object was swapped out of main memory. When an object is swapped back in, its pointers are restored to the configuration they had when the object was swapped out. Thus, pointers representing infrequent access patterns are likely to be lambdas, and pointers in more frequent access patterns will be leaves. Faults on leaves are cheaper than faults on lambdas, because faulting in a lambda requires that the secondary memory images of both the containing object and the referred-to object be consulted.

The converse operation to object faulting is object purging. LOOM can free up space in main memory by turning full objects into leaves. When it is known that no more references to the leaf exist (because the leaf has a zero short Oop reference count) then the leaf can be destroyed, and the ROT entry reclaimed. Note that this does not imply that the object itself is garbage, because there may be copies of its long Oop. Like OOZE, LOOM reclaims garbage by reference counting. Thus, the short Oop reference count and dual namespace serve to implement a kind of ephemeral garbage collection — objects in main memory can be reclaimed without examining objects on secondary memory. The LOOM system does not completely follow the ephemeral model, because new objects are given space in secondary storage upon creation. Inspired by LOOM, $\mathcal{M}$ (as we will see in chapter 4) uses similar dual namespaces to implement a more complete ephemeral garbage collection.

3.4.4 Berkeley Smalltalk

As a final example of an object-oriented memory, we take a brief look at Generation Scavenging [Ungar84]. Generation Scavenging is an implementation of a volatility-based batch-mode semispace garbage collector on a Sun workstation running Unix.

Generation Scavenging keeps track of two generations of objects: old and new. Main memory is divided up into four regions:

1. **NewSpace**: a set of wired pages[8] where new objects are created.

2. **PastSurvivorSpace**: a set of wired pages where new objects that have survived a few scavenges are kept.

3. **FutureSurvivorSpace**: an area of equal size to **PastSurvivorSpace** which will play the role of **newspace** during the scavenge.

4. **oldspace**: a set of normal (swappable) pages where more static objects are kept.

Scavenging is done when **NewSpace** becomes reasonably full. Beginning with a set of root objects, objects are transported from **NewSpace** and **PastSurvivorSpace** to **FutureSurvivorSpace**.

[8]The implementation of Generation Scavenging described in [Ungar84] could not wire pages, but the operational assumption was that these pages would be wired.

At the finish of the scavenge, both NewSpace and PastSurvivorSpace are empty; NewSpace is simply reused, and the two survivor spaces swap roles.

As objects survive scavenges, they become candidates for *tenuring*, or promotion to the older generation of objects. This generation is not garbage collected while the system is in operation, but is collected offline, with reorganization that attempts to perform useful compaction. SetElts of new object pointers into objects in oldspace cause an appropriate entry in an exception table (called the remembered set).

3.4.5 Summary

The object-oriented memories discussed above deal with the interaction of virtual memory and garbage collection in one of three ways. TGC tries to pack interesting objects onto pages so that the interesting object density is kept high. Generation Scavenging creates a special class of storage which is exempt from paging. By employing ephemeral techniques, this special storage can then be garbage collected without referring to swappable objects. Both TGC and Generation Scavenging are built on top of traditional paging systems. LOOM and OOZE use an object-oriented virtual memory; for these systems, the virtual memory and garbage collection systems are carefully designed to cooperate. As we will see in chapter 4, $\mathcal{M}$ is based upon this same idea.

$\boxed{4}$ The $\mathcal{M}$ Memory

Given the terminology of the previous chapter, $\mathcal{M}$ can be classified as an object-oriented virtual memory with an ephemeral mark/sweep garbage collector. In other words, the sole units of locality in $\mathcal{M}$ are objects (in this case, chunks); these objects are segregated according to their age; and inaccessible objects are discovered and reclaimed by a mark/sweep garbage collector. Unlike the systems described in chapter 3, $\mathcal{M}$ includes some features that are intended to provide support for a multiple processors, multiple memory managers, and multiple threads of control. These features will be discussed in section 4.6.

4.1 Hierarchies in $\mathcal{M}$

As mentioned in section 3.3, there is a close correspondence between the hierarchies used by virtual memory and ephemeral garbage collection. In the $\mathcal{M}$ system, these two hierarchies are fused into a unified storage hierarchy under which both virtual memory and garbage collection operate.

The memory space directly managed by $\mathcal{M}$ is divided into two sections: local and permanent memory. Local memory is analogous to the primary memory in a traditional virtual memory. Chunks in local memory are directly accessible to the execution unit. Local memory is the home of all new chunks and those chunks with high volatility.

Permanent memory, the next level deeper, has some similarities to secondary memory but is also shared among a number of different $\mathcal{M}$ units. As befits a virtual memory system, this logical hierarchy corresponds to a physical hierarchy; chunks close to the processor are stored in high-speed semiconductor memory, while chunks further from the processor are located on secondary storage devices like disks. Chunks in permanent memory are more stable than chunks in local memory. Table 4.1 gives a summary of the two levels of memory managed by $\mathcal{M}$, as well as possible extensions in both directions. The size of each level

level number	*Storage Level*	*Size (in chunks)*	*Volatility*	*Medium*
0	processor registers	10	high	on-chip memory
1	local memory	10000	high	RAM
2	permanent memory	10000000	moderate	mixed RAM/Disk
3	external memory	10000000000	low	Disk

Table 4.1: POSSIBLE $\mathcal{L}$ STORAGE HIERARCHY

is just an estimate; what is important is that the number of chunks at each level is much greater than that of the previous level.

Each of the memory levels of table 4.1 presents a different challenge to the system designer. $\mathcal{M}$ does not manage all of the memory levels of $\mathcal{L}$; it only manages local memory and some aspects of permanent memory. Chapter 6 will present some preliminary ideas for the structure of a permanent memory manager.

4.2 Chunks, Revisited

Since chunks are the basic unit of locality in $\mathcal{M}$, the virtual memory system must deal with them directly. In the current system, every chunk falls into one of three classes:

- *local temporary*: chunks representing temporary objects. They are found only in local memory. Local temporary chunks can contain pointers to all classes of chunks.
- *local permanent*: chunks, found only in local memory, that represent permanent chunks currently swapped into local memory. Like local temporary chunks, local permanent chunks can contain pointers to all classes of chunks.
- *permanent*: chunks in permanent memory. Permanent chunks can only contain pointers to other permanent chunks.

The restrictions on what pointers can be kept in permanent chunks are necessary because $\mathcal{M}$ does not use exception tables to record pointers from permanent to local chunks. To store a local pointer in a permanent chunk, the permanent chunk must be imported into local storage as a local permanent chunk. Backwards pointers are allowed to exist in local storage. These pointers are detected and dealt with during garbage collection.

Every chunk begins life as a local temporary. As it survives garbage collection cycles, it ages. At a certain age threshold, the chunk is promoted. Promotion involves upgrading the local temporary chunk to a local permanent chunk, and allocation of a corresponding permanent chunk. At this point, the chunk is no longer subject to garbage collection (by $\mathcal{M}$), and it now can be swapped out if it becomes uninteresting.

The hidden slots (see section 2.2) of a chunk provide a place for the virtual memory and garbage collection to keep information. In a local chunk, there are four hidden slots (see figure 4-1). The ref bit slot holds the reference bits for the other slots, and the attribute bits of the chunk. There are currently ten attributes, occupying 13 bits. These attributes record various pieces of information, and are of interest to various subsystems of $\mathcal{L}$. In the following list, the principal user of the attribute is listed after the attribute description.

- **mark**: a bit used by the garbage collector during the mark phase (gc).
- **allocated**: a bit set if the chunk is not on a free list (gc).
- **read-only**: a bit set if it writing into the chunk should cause an exception (execution model).
- **dirty**: a bit set if the chunk contents have been modified (gc,vm).
- **volatility**: a bit field encoding the volatility of the chunk.
- **age**: a bit field that records the number of garbage collection cycles the chunk has survived (gc,vm).
- **cache**: a bit that is set when the chunk in question is actually kept in the processor chunk cache (vm).
- **lock**: a bit that indicates that both Elts and SetElts using this chunk should cause exceptions (synchronization).
- **forward**: a bit indicating that the chunk is actually an invisible forwarding pointer; the real chunk ID is then found in the forward-pointer slot (synchronization).
- **export**: a bit set when other memory managers have requested ownership of the chunk (load balancing).

The other three hidden slots of a local chunk are the permanent name slot, the link slot, and the forward slot. The permanent name slot is used in local permanent chunks to record

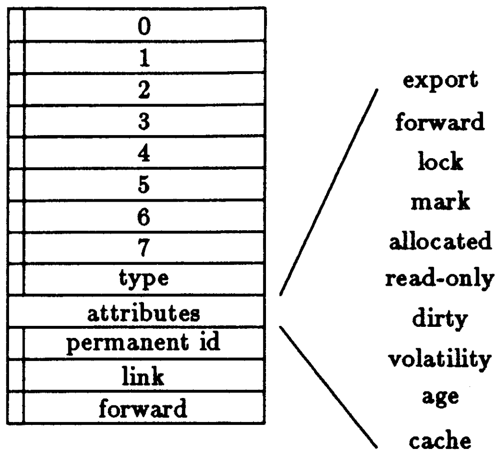


FIGURE 4-1: A LOCAL CHUNK

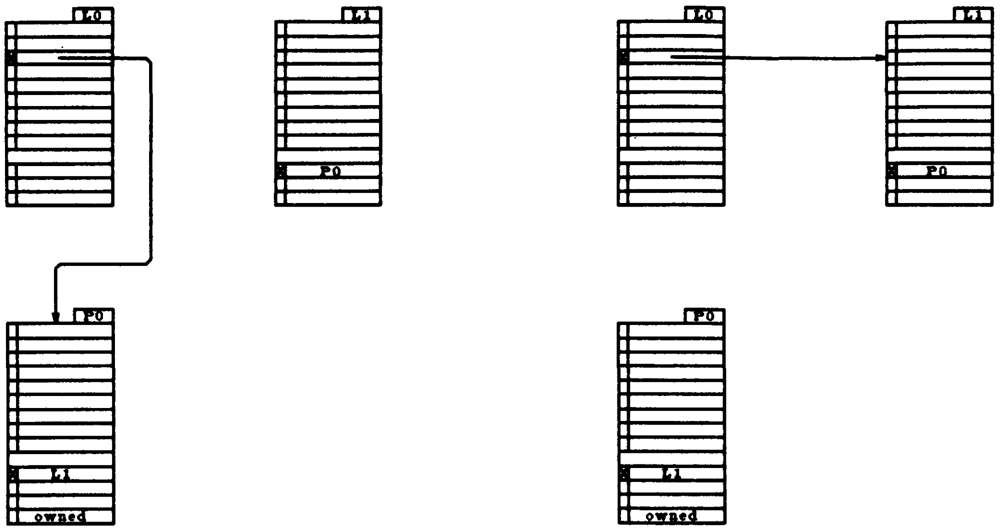
the associated permanent chunk's ID. It is invalid if the chunk is local temporary. The link slot is used by the garbage collector in marking and maintaining free lists. The forward slot is used to implement invisible forwarding pointers needed in our implementation of the Lock and Unlock synchronization primitives.

Permanent chunks have a similar set of hidden slots and the same attributes (the cache attribute is not used). There are four hidden slots in a permanent chunk. The link and forward slots serve the same purposes as their local chunk counterparts. The other two slots are the local name slot, which records the local ID that the chunk had the last time it was swapped into a local memory (or the current ID if the chunk is currently swapped in), and the owner slot, which records the identity of the local memory that the chunk is swapped into (and is set to zero if no local memory has the chunk swapped in).

In the following sections, we will use special ID tags to identify chunks. Local chunks will be identified by the letter L followed by a number, and permanent chunks will be identified by the letter P and a number. Thus L0 is a local chunk, and P12345 is a permanent chunk.

4.3 Virtual Memory in $\mathcal{M}$

The virtual memory part of $\mathcal{M}$ is based upon three operations: ***importation***, ***exportation***, and ***pointer updating***. Importation is the process of swapping a permanent chunk into the local memory to satisfy an object fault. Exportation is just the opposite — swapping local permanent chunks back out to permanent memory. Pointer updating is a method for estimating which of the swappable local permanent chunks actually fall in the working set.



Before: chunk L1 initially contains the permanent chunk ID P0. P0 is owned by this memory system.

After: L0 contains the equivalent local ID L1, obtained from the local name slot of P0.

FIGURE 4-2: TRIVIAL IMPORTATION

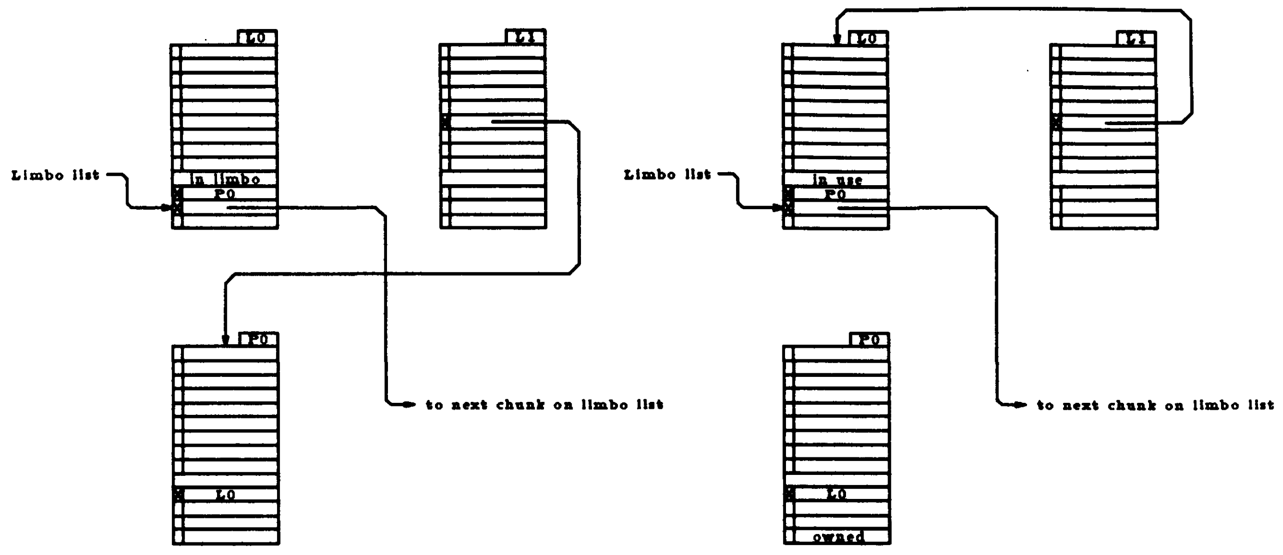
Each of these is explained in more detail below.

4.3.1 Importation

When a Metaname-Fetch or Metaname-Store operation attempts to dereference the ID of a permanent chunk, an object or importation fault takes place. The steps in handling an importation fault are:

1. *ownership check*: $\mathcal{M}$ first checks to see which memory manager owns the permanent chunk. There are three possibilities here. Either no manager owns the chunk, $\mathcal{M}$ owns the chunk, or some other manager owns the chunk. If the chunk is owned by another manager, then the importation is blocked, and $\mathcal{M}$ must negotiate with that manager to resolve the situation. If $\mathcal{M}$ owns the chunk, then there is already a copy of that chunk in local memory. In this case, the importation is *trivial*: $\mathcal{M}$ consults the local name slot of the permanent chunk to discover the proper local ID to return (see figure 4-2). If there is no owner, then $\mathcal{M}$ acquires ownership of the chunk and proceeds to the next step.

2. *cheap importation check*: after acquiring ownership, $\mathcal{M}$ looks to see if the permanent chunk has a valid local copy. This may be possible if the permanent chunk was



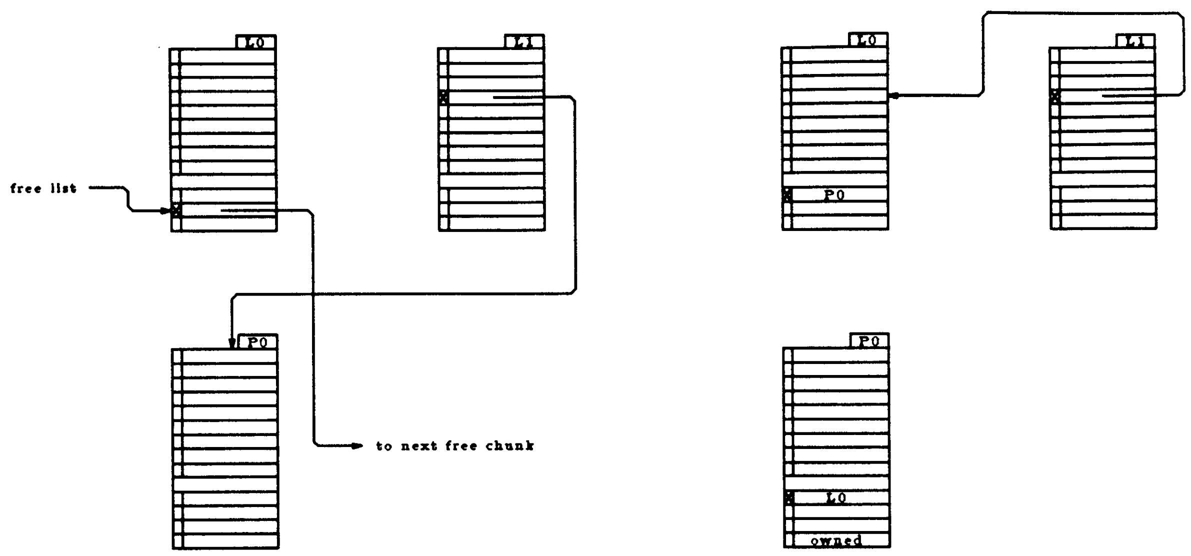
Before: chunk P0, unowned, contains a record of a former local equivalent, chunk ID L0. L0 is on the limbo list. L1 points to P0.

After: L0 has been re-allocated, and L1 points to it.

FIGURE 4-3: CHEAP IMPORTATION

previously imported, and then was partially exported (limboized) because it seemed to have fallen out of the working set. Limboization removes the ownership but keeps the local ID in the permanent chunk and the permanent ID in the local chunk. If these IDs are consistent (that is, if the local ID of the permanent chunk is the local chunk ID), the local chunk can be removed from limbo in a *cheap* importation that does not involve copying the chunk contents (see figure 4-3. If cheap importation is not possible, the importation continues with the next step.

3. *local chunk allocation*: given that neither blocking, trivial importation, or cheap importation has happened, the importation fault handler next makes a request for a new local chunk. This allocation request can trigger garbage collection, if the number of free local chunks is running low. In this case, the importation will be retried after garbage collection finishes. If all has gone smoothly up to this point, then $\mathcal{M}$ has both ownership of a permanent chunk, and a new, empty local chunk. Next, the contents of the visible fields of the permanent chunk are copied directly into the local chunk. Some of the attributes (**read-only, forward, lock, volatility**) are inherited from the permanent chunk, and the others are initialized to the correct values. The permanent and local IDs are stored in the local and permanent chunks, respectively,



Before: chunk P0 is unowned and has no record of a former local equivalent. L0 is on the free list. L1 points to P0.

After: L0 has been allocated as the local equivalent to P0, and L1 points to it.

FIGURE 4-4: FULL IMPORTATION

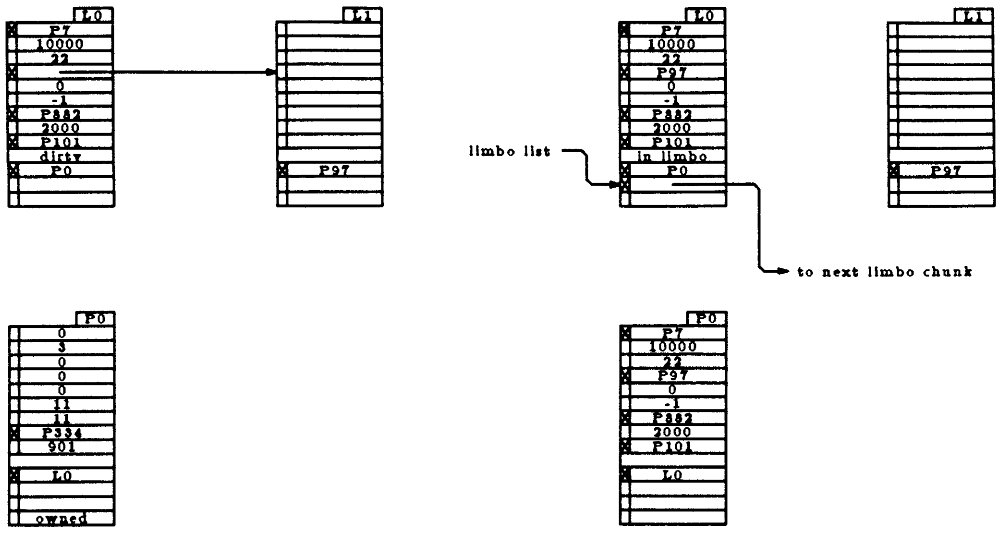
and the importation completes. This *full* importation is illustrated in figure 4-4.

Limboization of chunks adds a lot of complexity to $\mathcal{M}$. Section 5.5.4 discusses whether or not the additional complexity, both here and in the sweep phase of garbage collection, is justified by the potential speed advantages of cheap importation.

4.3.2 Pointer Updating

When a chunk is imported, it is full of permanent chunk IDs. If one of these IDs is used in an Elt or SetElt, the corresponding chunk will be imported. At this point, two valid IDs exist for the chunk: the original, permanent ID, and the newly created local permanent ID. Pointer updating is the process of replacing the permanent ID with the local permanent ID in the original chunk. This both prevents importation faults (which would be handled via trivial importation) on future uses of the chunk ID, and provides $\mathcal{M}$ with a simple way to estimate the working set.

The information about which chunks are included in the working set is recorded as follows. Local temporary chunks are always in the working set. Local permanent chunks are in the working set if they are accessible by a chain of local IDs. Permanent chunks are never in the working set. Thus, the act of pointer updating after an importation serves to



Before: chunk L0 and chunk P0 are a local-permanent pair. L0 was not marked by the garbage collector.

After: L0 has been put onto the limbo list. P0 is unowned but still remembers its association with L0. L0 and P0 have identical data fields.

FIGURE 4-5: LIMBOIZATION

place the imported chunk into the working set. Pointer updating is undone by the garbage collector.

4.3.3 Exportation

The converse of importation is exportation, the removal of accessible objects from local storage. Exportation is actually done during the sweep phase of garbage collection. The mark phase of garbage collection marks all local permanent chunks that are considered to be in the working set. An unmarked local permanent chunk is therefore considered to be out of the working set.

At this point, the first phase of exportation, *limboization*, takes place. If the chunk was dirty, the contents of the local permanent chunk are written back to permanent storage. The ownership of the permanent chunk is released, but the permanent chunk retains the local ID, and the local chunk retains the permanent ID. The local chunk is then put on a special *limbo* list, threaded through the link slot, and its allocation bit is cleared. Figure 4-5 illustrates this process.

After limboization, one of two things can happen to the local permanent chunk: it can either be reallocated as new storage, or reclaimed by cheap importation. Reallocation of the local chunk completes the exportation process: the old local and permanent IDs are erased. Any future references to the permanent chunk will have to be handled by a full importation. If the local chunk is reclaimed by cheap importation, it is restored to regular local permanent status, and there is no need to copy the data fields from permanent to local memory.

4.4 Garbage Collection in $\mathcal{M}$

$\mathcal{M}$ has a simpler task of garbage collection than the object-oriented memory systems of section 3.4. Since chunks are the unit of locality, and chunks are all the same size, $\mathcal{M}$ does not have to worry about compacting objects onto pages. While a compacting garbage collector is vital in a paged object-oriented system, compaction is not an issue for $\mathcal{M}$. Because $\mathcal{L}$ implements control stacks from chunks, $\mathcal{M}$ must provide for extremely cheap temporary storage. Finally, $\mathcal{M}$ is intended for high performance, and so cannot afford to become too complicated. The important factors in selecting a garbage collection algorithm for $\mathcal{M}$ are therefore simplicity of implementation, the cost of temporary storage, and overall performance.

The requirement of cheap temporary storage is met by ephemeral garbage collection. Non-ephemeral methods would have difficulty coping with the rapid allocation and deallocation rates of temporary chunks. Although most existing implementations of ephemeral garbage collection are based upon semispace collectors, ephemeral techniques are compatible with other methods of garbage collection.

Semispace methods were not used in $\mathcal{M}$ for several reasons. The first reason is that they waste space. The classical semispace algorithm can only use 50% of its address space for objects. The local memory of $\mathcal{M}$ is mapped directly to physical memory, so a loss of 50% is a serious drawback. The second reason is that the primary value of the semispace methods comes in a paged virtual memory environment. $\mathcal{M}$ swaps objects, not pages. To achieve compaction, the semispace methods copy live objects. A straightforward implementation of a non-copying algorithm may be more efficient when paging and compaction are not issues.

The non-compacting garbage collection methods fall into two classes — variants of mark/sweep and reference counting. Since reference counting cannot reclaim circular structures, and such structures (like our doubly-linked control stack) will be commonplace in

$\mathcal{L}$. The $\mathcal{M}$ garbage collector is based upon the mark/sweep algorithm. This algorithm is intrinsically simple, is compatible with ephemeral garbage collection, and can potentially be run in parallel with execution (via an adaptation of the Djikstra-Lamport on-the-fly mark/sweep method [Dijkstra78]).

4.4.1 Details of $\mathcal{M}$ Garbage Collection

Garbage collection in $\mathcal{M}$ is triggered by the local chunk allocator when the total number of chunks on the free list and limbo list falls below a certain threshold. Since garbage collection does not require any extra space, this threshold can be zero.

Garbage collection is done by a modified mark/sweep algorithm that contains three phases. The first phase is a limited-depth mark, starting from a root object, following only local IDs. This marks all local chunks reachable from the root object via local pointers. Because no volatility checks are performed during execution, and this first mark phase follows only local pointers, it is possible for some reachable local chunks to be bypassed. These are chunks that would normally have been marked through the exception tables. This mark is performed using no additional storage by using the link slot to hold the mark stack.[1]

To mark these chunks properly, $\mathcal{M}$ performs a volatility-checking sweep. This sweep examines all unmarked local permanent chunks to see if they point to unmarked local temporary chunks. Whenever one of these backwards pointers is found, an auxiliary mark is started from the pointer. The local temporary object pointed to is then promoted to local permanent. After this second phase, all accessible local temporary chunks have been marked, and all interesting local permanent chunks have been marked.

The third phase of garbage collection resembles a traditional sweep. Unmarked local temporary chunks are threaded onto a free list through the link slot. The allocation bit is used here to keep from putting a chunk on the free list if it was already there before gc started. Marked local temporary chunks are aged, and those that exceed the age threshold are promoted. Marked local permanent chunks have all of their pointer updates undone — that is, all of the local permanent IDs in the chunk are replaced with equivalent permanent IDs. Unmarked local permanent chunks are limboized.

[1]This is possible because the mark will touch only accessible objects, and the link slot is only normally used by inaccessible objects.

4.4.2 Garbage Collection Issues

There are a few subtle issues here which were glossed over in the preceding section. The volatility-checking sweep does not detect all backwards pointers, but only those that exist in unmarked chunks. These chunks are considered to be out of the working set. It is therefore unlikely that their contents will become garbage in the near future merely because the attention of the processor will be directed somewhere else. $\mathcal{M}$ would like to export these uninteresting chunks; but exportation of a chunk is only possible if the chunk contents are expressible in terms of permanent IDs and scalars. Thus, the local temporary chunks pointed to by this local permanent chunk are promoted. In effect, $\mathcal{M}$ decides that the temporary objects at the ends of the detected backwards pointers are not very temporary any more.

$\mathcal{M}$ only carries this promotion out to one level; that is, only the chunk referred to by the backwards pointer is promoted. If this promoted chunk itself contains local temporary IDs, then the promotion creates more backwards pointers. The original version of $\mathcal{M}$ recursively promoted, removing all backwards pointers, but this strategy proved to be too aggressive. $\mathcal{M}$'s decision that chunks at the end of detected backwards pointers are no longer temporary is not always a good one, and recursive promotions only compounds the effects of a bad decision. Restricting promotions to one level reduces the amount of unnecessary promotion without greatly increasing the amount of time it takes to legitimately promote large structures. For example, in a treed array, the number of promotions can increase by a factor of 8 each garbage collection pass. $\mathcal{M}$'s aging policy also helps to ease the difficulty of turning a large temporary structure into a permanent structure.

Backwards pointers can exist in marked chunks for as long as is necessary. The aging mechanism will eventually promote the temporary object at the end of the backwards pointer, and the backwards pointer will then no longer be backwards.

The actions of the second sweep are straightforward, except for the replacement of local names by permanent names in local permanent chunks (also known as pname updating). Pname updating is a crude mechanism for breaking up local accessibility, so that the next garbage collection pass can put as many local permanent chunks on limbo as possible. Without some kind of local name replacement, it would be possible for local memory to fill with a locally accessible network of local permanent chunks, none of which were garbage; at this point, $\mathcal{M}$ would not be able to limboize anything, and would be unable to free up any local storage. Thus, $\mathcal{M}$ replaces all local permanent IDs in local permanent chunks.

```
procedure push-end (list : lnode; new_item : integer);
 begin
  while list^.cdr <> nil do
   list := list^.cdr;
  list^.cdr := new(lnode);
  list^.cdr^.car := new_item;
  list^.cdr^.cdr := nil
 end;
```

FIGURE 4-6: EXAMPLE PROGRAM

Those local permanent chunks which were truly in the working set will have their local IDs resurrected by trivial importations. Those that were not will be limboizable at the next garbage collection cycle. Because of this one cycle delay in the effect of pname updating, it is possible that the garbage collector may require more than one cycle to free up storage (this is very unlikely, however). Section 6.1 will discuss ways in which the working set can be better estimated, so that local permanent chunks in the working set are not subjected to this cycle of pname updating followed by trivial importation.

4.5 An Example

This section presents an example of the workings of $\mathcal{M}$. To keep the example tractable, we have reduced the local memory size to five chunks, L0 – L4. To simplify some of the accompanying figures, we use two different notations for chunk IDs. Explicit pointers (drawn as arrows originating in a slot of a chunk, and terminating at another chunk) are used for most IDs. The association between permanent chunks and local chunks is denoted by placing the chunk ID tag in the proper slot. To help distinguish between scalars and pointers, slots containing pointers have their reference bits checked.

The example program, written in pidgin PASCAL (see figure 4-6), is one that adds a new element onto the end of a non-empty list. Here we have created a `lnode` object which contains two fields called `car` and `cdr`. In this example, a `lnode` will be represented as a chunk. The `car` will be stored in slot 0 and the `cdr` in slot 1. The special pointer `nil` will be represented as a scalar value 0. This is not the most space-efficient chunk-based implementation of lists, but it will do for this example. In this case the initial list will be (0 1 2 3) and we will use this routine to add new elements 4 and 5.

```
label-0 (test (3 1))              ;examine list's cdr
        (jnr label-1)             ;jump if nil (not a reference)
        (move (3 1) (3))          ; else list := list^.cdr
        (jump label-0)            ; and loop back
label-1 (alloc (3 1))             ;make new lnode
        (move (4) (3 1 0))        ; and set its fields
        (movei 0  (3 1 1))        ; cdr is nil
```

FIGURE 4-7: EU INSTRUCTIONS FOR THE PROGRAM

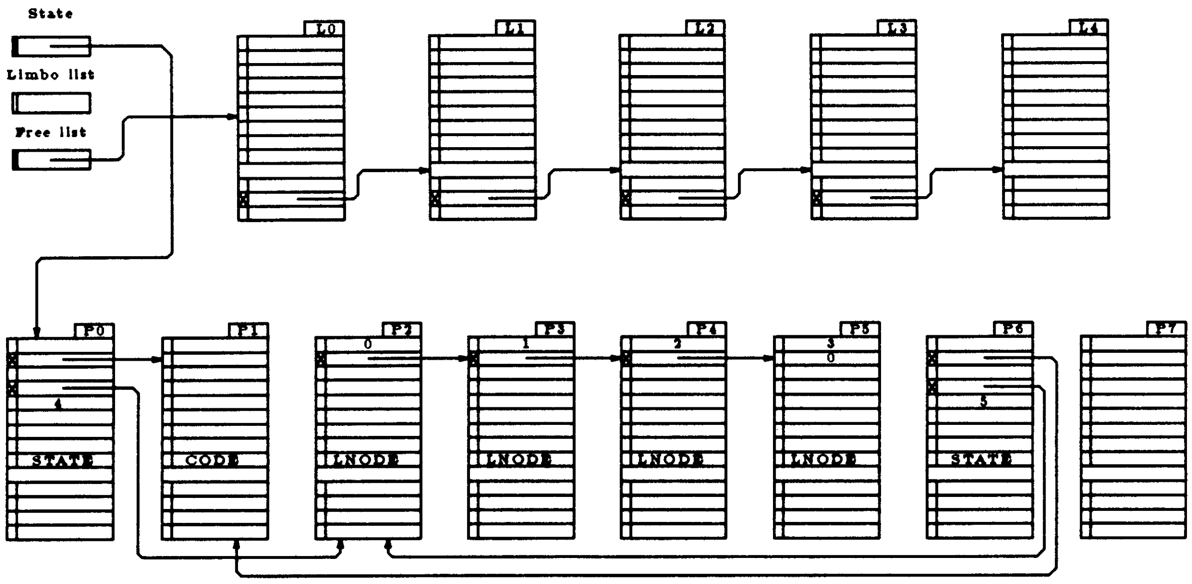


Figure 4-8 : INITIAL SETUP OF MEMORY

The initial state of the local and permanent memories is shown in figure 4-8. Local memory is empty; all the local chunks are threaded onto the free list through their link slots. The information required to execute **push-end** has been stored into chunks in permanent memory. In particular, P0 is a runnable STATE representing the first call to **push-end**; arguments for this call have been placed inside P0. Slot 3 of P0 points to the first **lnode** (P2) of the list, and slot 4 contains the value to be inserted in the list, 4. As in the control stack example from chapter 2, slot 1 of P0 points to a code chunk (P1) and slot 0 contains the microstate (the detailed structure of the microstate, which contains the condition code, state status and instruction offset bits, is not shown in the figures). P6 is the STATE chunk for the second call to **push-end**. Figure 4-7 lists the EU instructions that implement **push-end**. The code chunk P1 holds the machine language equivalents of these instructions.

Execution begins when the SMU passes the ID P0 to the EU. The EU prepares for fetching

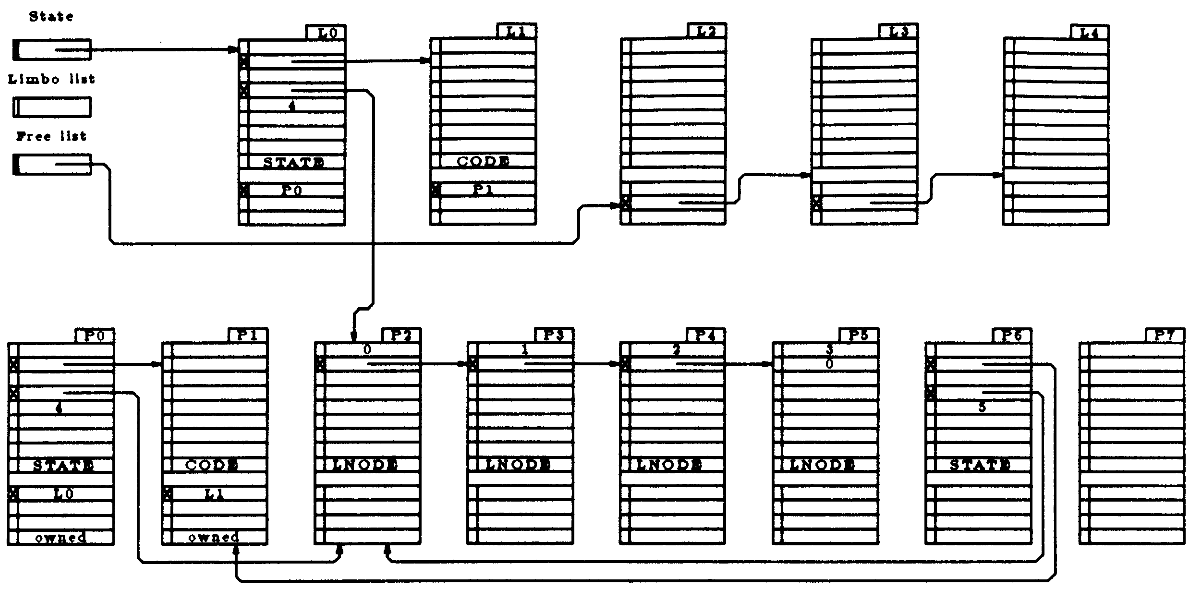


FIGURE 4-9: MEMORY AFTER IMPORTING STATE AND CODE CHUNKS

the first instruction by trying to read the microstate and code chunk pointer from P0. P0 is a permanent chunk, so the **MetanameFetch** used to get the microstate causes an importation fault. Since local memory is empty, chunk P0 is imported fully: chunk L0 is popped off the free list to serve as the local permanent copy, the fields of P0 are copied into L0, and the EU updates its STATE pointer from P0 to L0. After the importation is handled, the faulting **MetanameFetch** completes. The EU is then able to fetch the code pointer from L0 without any importation fault. By this point, the EU is able to fetch the correct instruction out of the code chunk; in doing this, the code chunk needs to be fully imported. The corresponding state of memory is shown in figure 4-9. Note that the second importation fault has updated the ID of the code chunk, so that future instruction fetches will not suffer importation faults.

The first machine instruction in push-end is `(test (3 1))`. This instructs the EU to fetch the object that has metaname `(3 1)` with respect to the STATE, and (among other things) examine the reference bit. If the reference bit is set, then the `r` condition code bit is set in the microstate. Execution of this instruction causes another importation fault, bringing chunk P2 into the local memory as L2. Since the first `lnode` of the list has P3 as its `cdr`, the EU sets the `r` (reference) condition code bit. The next instruction, `jnr`, examines the `r` bit and jumps when it is clear; this time it is set, so the EU does not jump, but goes on to update the value of `list`, arriving at the state shown in figure 4-10. Note that chunk L2 is no longer locally accessible, and is thus a candidate to be put onto the limbo list, since

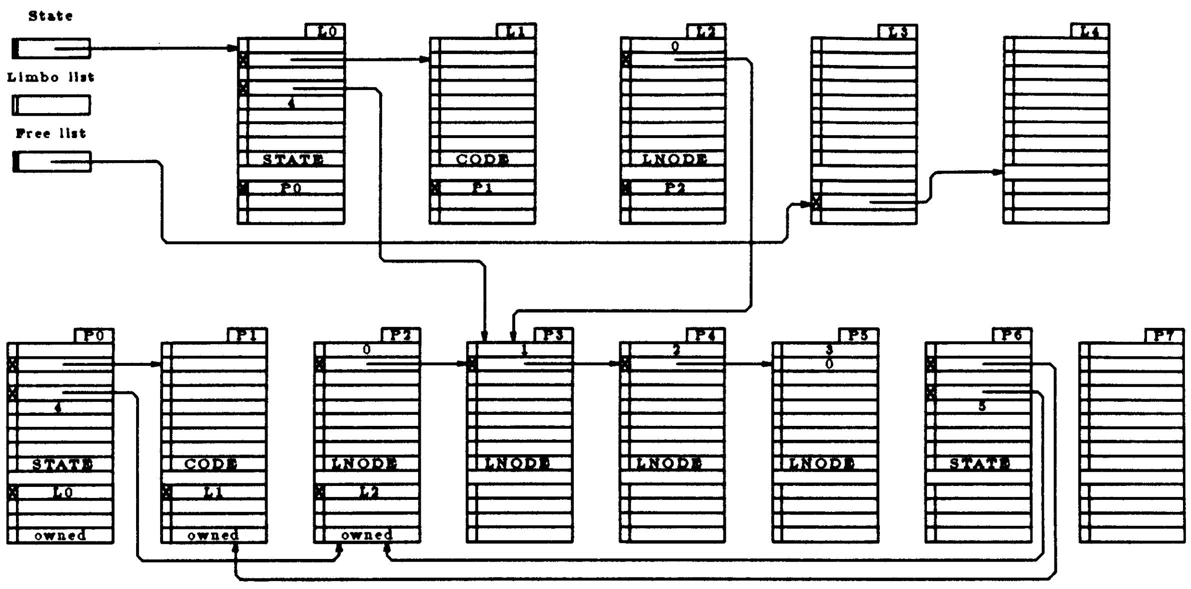


FIGURE 4-10: MEMORY STATE AFTER THREE INSTRUCTIONS

it is a local permanent chunk.

The next two lnodes are traversed in the same fashion, both causing importation faults. After the third (move (3 1) (3)) instruction has faulted in P4, there are no more free chunks; local memory is completely allocated. In this example, the garbage collector runs just after the instruction that allocates the last free chunk. Because execution of an instruction can cause multiple importation faults (as the first instruction did), a somewhat more sophisticated mechanism is needed to handle or prevent the case where the garbage collector needs to run in the middle of an instruction. Figure 4-11 shows the state of memory just before invoking the garbage collector.

The root object in this example is the STATE pointer of the EU. The mark phase thus starts at chunk L0, and, following local IDs only, marks chunks L1 and L4. These chunks have no local IDs, so the mark phase finishes. The first sweep phase then examines all of the local chunks. Since there are no backwards pointers, the first sweep phase does nothing. The second sweep phase notices that chunks L2 and L3 are unmarked and local permanent, so they are put on the limbo list. Since neither chunk was dirty, the data in those chunks does not need to be written back out. The second sweep also replaces local IDs with permanent IDs in the other three local chunks, and clears the mark bits of all the chunks. Garbage collection is then finished, and has freed up two chunks. The memory state at this point is shown in figure 4-12.

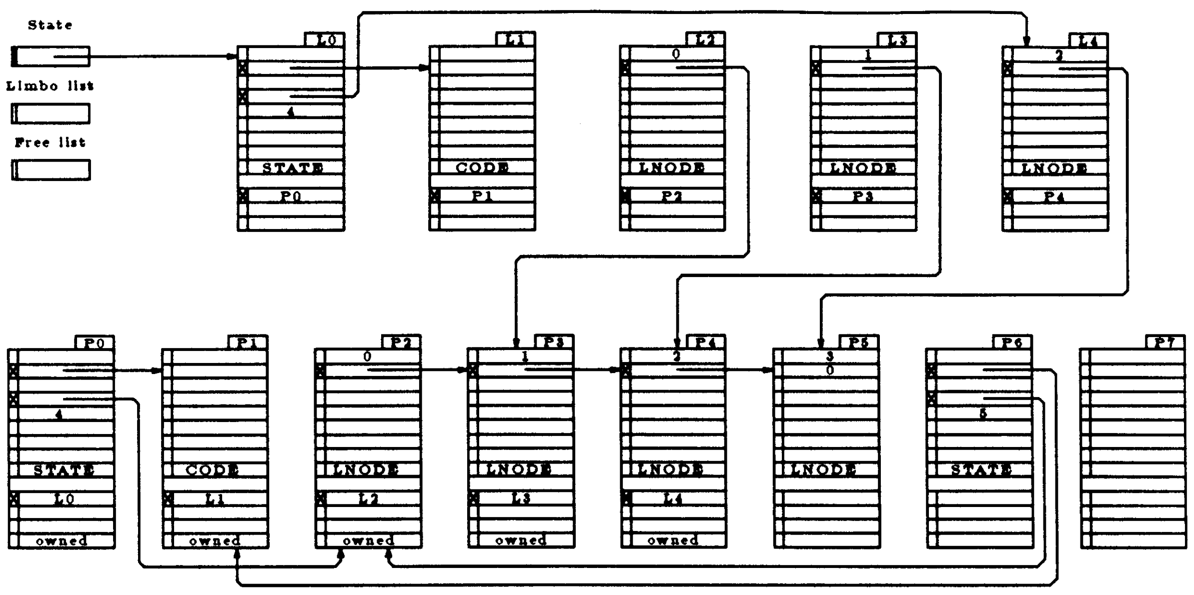


FIGURE 4-11: MEMORY JUST BEFORE GARBAGE COLLECTION

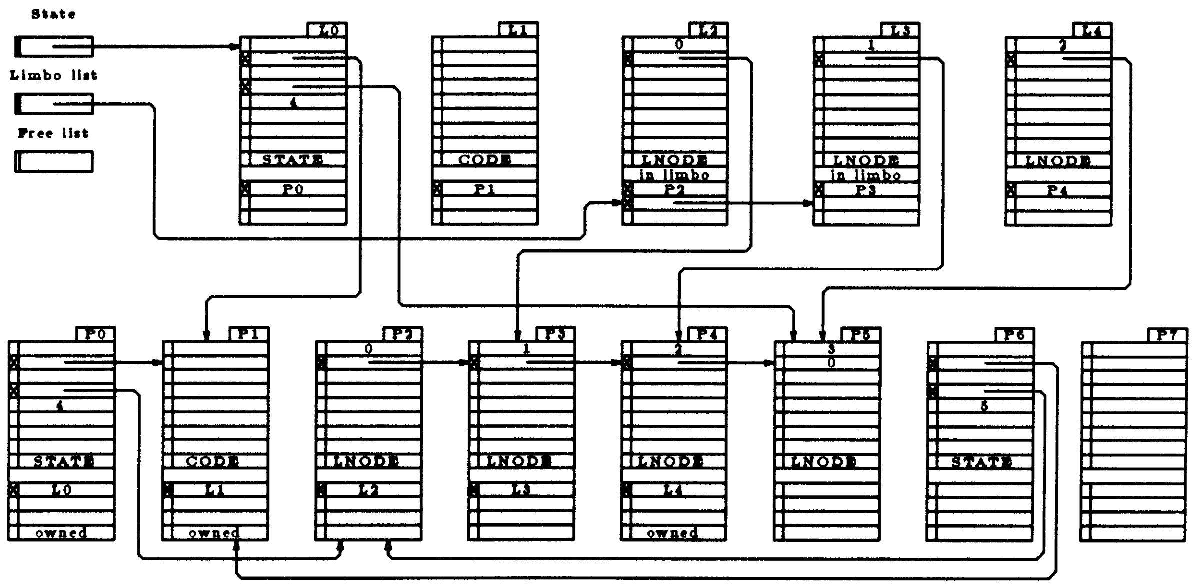


Figure 4-12 : MEMORY AFTER FIRST GARBAGE COLLECTION

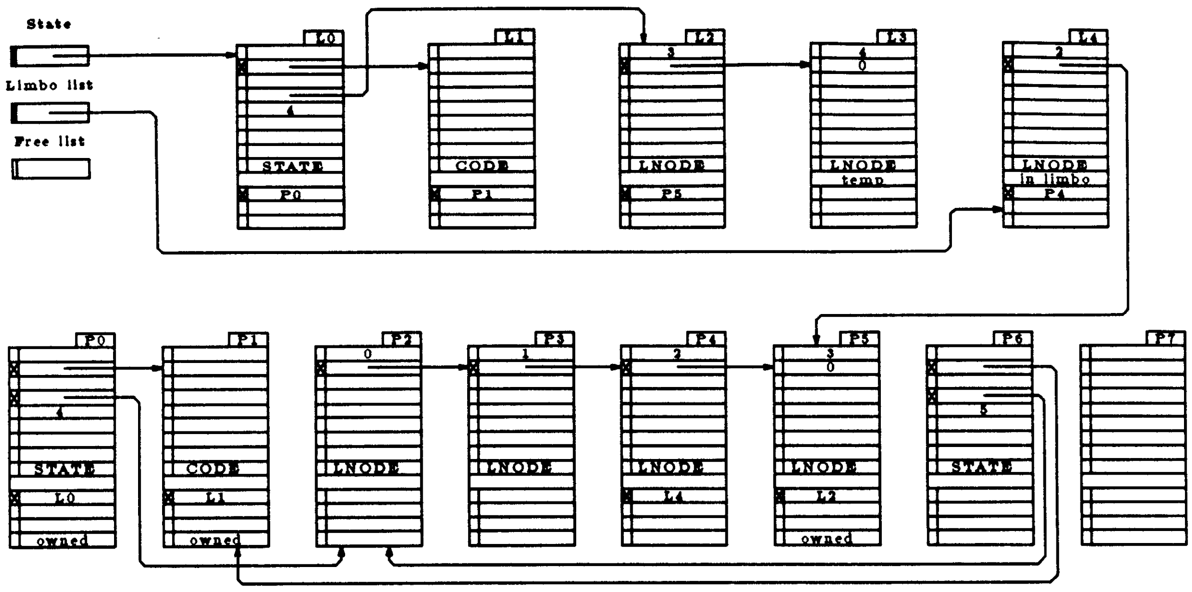


FIGURE 4-13: MEMORY AFTER COMPLETION OF FIRST RUN

The next instruction fetch causes an importation fault on the code chunk because the sweep replaced L1 with P1 in L0 (the EU's STATE pointer is immune to this kind of replacement). This time, the importation is trivial, since the code chunk is already local (this is an example of an unintelligent pname update followed by a trivial importation). The next instruction tests the cdr field of P4 (trivially imported as L4), and finds a pointer. The EU then reaches the move instruction, which needs to import P5. This is a full importation, like the ones before, except that the local chunk (L2) is allocated off of the limbo list instead of the free list. (The general strategy used by $\mathcal{M}$ is to preferentially allocate from the free list to give chunks on the limbo list more time during which they can be cheaply imported. In this case, the free list is empty, so allocation is done from the limbo list.)

After testing the cdr of L2 and finding a scalar, the EU branches to the code at label-1. It then must create a new lnode, which it does by asking $\mathcal{M}$ for a new local chunk. L3 is allocated from the limbo list for this purpose. Once again there are no more free chunks, and garbage collection runs. This time, only chunk L4 is not reachable via local IDs; all the other local chunks are marked. The first sweep does not detect that L2, a local permanent chunk, points to L3, a local temporary (a backwards pointer) because L2 has been marked. Auxiliary marking phases are only necessary when unmarked local permanent chunks contain backwards pointers. The second sweep ages L3, replaces local permanent IDs with equivalent local permanent IDs, and puts L4 onto the limbo list.

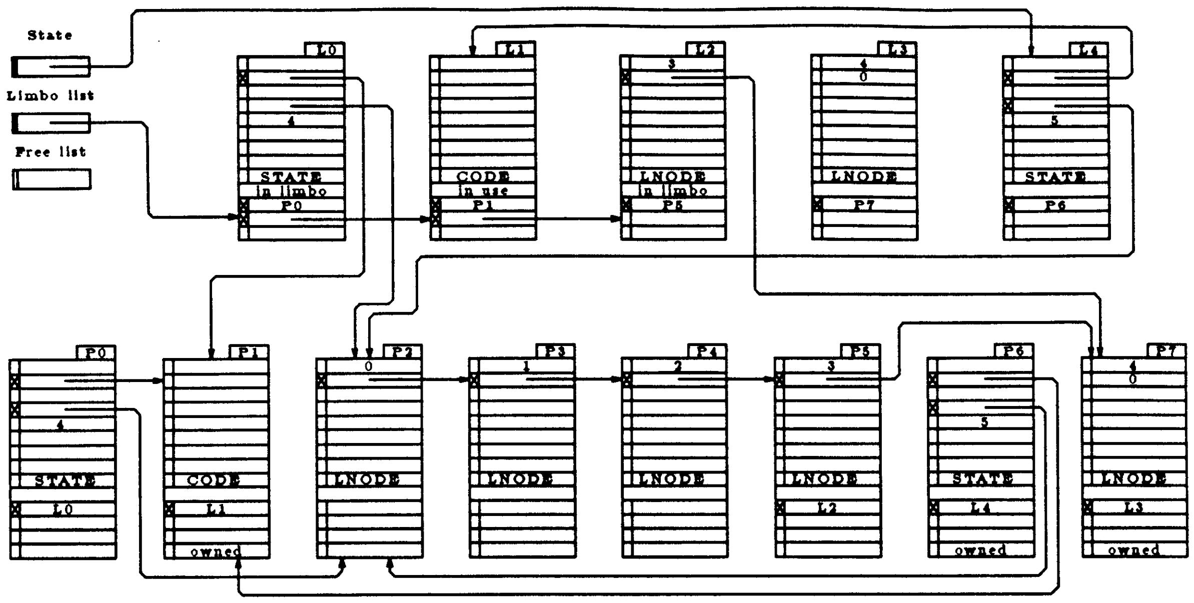


FIGURE 4-14: MEMORY AFTER CHEAP IMPORTATION

The next few instructions cause a series of trivial importations, but the program is able to complete without further chunk allocations or garbage collections. The state of memory at the completion of the first call to **push-end** is shown in figure 4-13.

The SMU then passes the ID of the STATE chunk for the second call, P6. The microstate access causes this chunk to be imported, and this uses up the last free chunk, L4. Garbage collection runs again. This time, L2 goes unmarked; the first sweep therefore detects a backwards pointer between L2 and L3. To handle this, L3 is promoted; during this promotion, P7 is allocated to serve as the permanent copy. Then an auxiliary mark is started from L3, but L3 contains no pointers, so only L3 gets marked. The second sweep puts the unmarked local permanent chunks onto the limbo list (L3 is not limboized because it is marked). Note that the old STATE chunk L0 was dirty; limboization causes its contents to be copied back out to P0. The fetch of the first instruction then proceeds: the microstate is read out of L4, and then the EU attempts to dereference through the code chunk ID, P1. The importation fault that happens here is handled through cheap importation, since there is a valid copy (L1) of P1 in the local memory, on the limbo list. The state of memory at this point is shown in figure 4-14. Cheaply imported chunks are not removed from the limbo list due to the cost of a random deletion from a singly linked list; instead, the **allocated** bit of L1 is flipped to indicate that it is actually in use.

The first instruction of this second program faults in the first `lnode` of the list. The only free chunks at this point are those on the limbo list. Local chunk `L0` is allocated by the full importation of the first `lnode`. When $\mathcal{M}$ is asked to import the second `lnode`, it must skip over chunk `L1` since it was been cheaply imported, and so $\mathcal{M}$ allocates `L2` instead; `L1` is freed from the limbo list at this point. Thus, cheaply imported chunks are freed from the limbo list as it is traversed during allocation (see section 5.3.4). Because the limbo list is threaded through the link slot and may contain a mixture of free and allocated chunks, a special limbo list cleanup step is necessary if $\mathcal{M}$ tries to garbage collect when there are chunks on the limbo list (remember that the mark phase uses the link slot to hide the mark stack). In this example, the garbage collector does not run until there are no free chunks, so the limbo list cleanup is not needed. This second call finishes in a fashion similar to the first, and does not illustrate any more of the interesting aspects of $\mathcal{M}$.

This example has attempted to give a feeling for the operation of $\mathcal{M}$. Because the local memory was so small, $\mathcal{M}$ was forced to perform many of importations and limboizations. In a larger local memory, the entire list would be imported by the first call to `push-end`, and the second call would not have to import any of the list chunks.

4.6 Special Features

$\mathcal{M}$ includes support for both multiple threads of control and sharing of a permanent memory by multiple local memories. The multiprocessing aspects of $\mathcal{L}$ will be the subject of much future research, so details presented here are somewhat preliminary.

4.6.1 Local Memory Coherency

There are two features of $\mathcal{M}$ which are intended to help support multiple local memories share a permanent memory. The first of these is the **owner** field of a permanent chunk, which records which local memory has imported that chunk. A local memory attempting to import a permanent chunk must first acquire ownership of that chunk. If it is unowned, this step is simple. If another local memory owns the chunk in question, some kind of negotiation must go on between the two to resolve the situation.

$\mathcal{M}$ uses the **export** bit of a local chunk to indicate that some other local memory has requested ownership. Under a simple demand-exporting convention, such a chunk would be exported at first opportunity, where the various local memories interested in the chunk could fight on an equal basis for ownership. Note that only local permanent chunks are candidates for demand exportation, because other memory systems cannot know anything about the local temporary chunks.

4.6.2 Support for Multiprocessing

One of the goals of $\mathcal{L}$ is to provide low-level support for the exploitation of fine-grain parallelism. Our plan is to use the Lock and Unlock primitives to synchronize execution of interacting threads of control. We are not yet settled upon an implementation of Lock or Unlock, but the proposed implementations require some assistance from $\mathcal{M}$. This section will not go into much depth on the actual details of these implementations, but will simply attempt to present the issues relevant to $\mathcal{M}$.

Locking by Copying: One proposal for Lock involves copying the contents of the chunk in question to a newly allocated chunk (whose ID will be uniquely held by the lock-er, at least initially). Unlocking is done by copying the contents back to the original chunk, and putting a forwarding pointer in the **forward** slot of the new chunk. In this implementation, the garbage collection scheme of $\mathcal{M}$ can be modified slightly to detect and remove unnecessary forwarding pointers, so that the chunks allocated in locking can be reclaimed as garbage after an Unlock.

Locking by Key Counting: This alternative implementation of Lock uses a generation count in both the chunk and the local ID to implement locks in the local memory. When a chunk is locked, its key generation count is incremented and copied into a *key generation* subfield of a local chunk ID. Because the key generation count will have to fit into some small number of bits, it needs to be refreshed. The garbage collection algorithm of $\mathcal{M}$ can be modified to discover and deactivate obsolete keys, so that the key counts are refreshed after each garbage collection.

4.7 Summary

Before providing any more detail about the $\mathcal{M}$ system, it is perhaps useful to briefly review the objectives of $\mathcal{M}$ and the strategies devised to fulfill those objectives. The $\mathcal{M}$ memory system was designed with the following guidelines and goals in mind:

Goals:

1. The average chunk access time should be as small as possible.
2. There should be as many chunks as possible.

Guidelines:

1. Chunks are the carriers of locality.
2. Most chunks will only be used temporarily.
3. Chunks are all the same size.
4. The chunk network will contain many cycles of pointers.

Goal 1 leads to direct mapping of local chunks onto memory, since a level of indirection (ID to address translation) will slow down chunk accesses. Goals 1, and 2, and guideline 1 lead to the dual namespace virtual memory system. Goal 2 and guideline 2 lead to ephemeral garbage collection. Guidelines 3 and 4 and the direct mapping of local chunks indicate the need for a non-compacting, space efficient garbage collector that can reclaim cyclic objects — the mark/sweep method.

$\boxed{5}$ An Implementation of $\mathcal{L}$

This chapter opens with the details of a microcode emulation of $\mathcal{L}$ on a Texas Instruments Explorer I LISP machine. This emulation models many of the features of the EU, SMU, and $\mathcal{M}$ systems. The second part of this chapter presents the results of some experiments done with the emulator: the cost of temporary storage, the usefulness of the limbo list, and the effectiveness of ephemeral garbage collection.

5.1 The $\mathcal{L}$ Processor Emulation

The $\mathcal{L}$ processor is, as of this writing, still very much in the design stage. The proceeding chapters have intentionally glossed over most of the details of the EU and SMU, in part because the subject of this thesis is $\mathcal{M}$, not $\mathcal{L}$, but primarily because these other portions of the $\mathcal{L}$ architecture are much less well defined. The $\mathcal{L}$ processor emulation described in this chapter is built on a fairly complete implementation of $\mathcal{M}$ and a preliminary implementation of the EU and SMU.

The term *emulation* is traditionally used to describe a low-level simulation of one processor by another [Baer80]. Emulators are typically implemented at the microprogram level, so that the emulating machine can directly execute binary sources of programs written for the emulated machine. One common use of emulation is to implement compatibility between various members of a processor family. Anther use of emulation, more germane to this thesis, is to provide an instrumentable implementation of a machine that does not yet exist. Because emulation is done at low level, the emulator is usually reasonably fast and can support software development. The resulting programs, when run on the emulator, can provide important performance information which can be fed back into architectural decisions. Used in this way, emulation is a powerful tool for the computer architect.

There are a number of machines that could be called general purpose emulation engines.

The Nanodata QM-1, Cal Data 100, and Burroughs 1700 all lack a native instruction set. Inevitably, emulations written on these general purpose machines suffer from a poor fit between the macro instruction language and the available hardware, leading to slower than real time performance,[1] but this does not usually compromise the benefits of emulation.

The Explorer I processor [TI84], although principally used as a LISP execution engine, has features that make it a reasonably good general purpose emulation engine. The most important of these are a large (16K word) writable control store and a large number (over 1K) of scratchpad registers. The Explorer processor has 32 bit data paths throughout. Besides the writable control store, internal memories include a 64 word M memory, 1K word A memory, 1K word stack cache, 64 word microprogram control stack, and assorted special-purpose registers. Another useful feature of the Explorer I processor is the general purpose byte-field extraction and deposition operations. These instructions are able to manipulate fields from 1 to 32 bits wide. As we will see in section 5.2.1, the emulator needs to perform many bit and byte operations. The Explorer I processor is based upon the CADR processor [Knight79], and is built around the NuBus [Ward80], both developed at MIT.

5.1.1 Emulation History

The $\mathcal{L}$ processor was originally the subject of a 100 instruction per second COMMON LISP simulation [Blair86]. Next came an emulator, written in Explorer microcode, which implemented the EU and some of the features of the SMU.[2] At about the same time, $\mathcal{M}$ began evolving as a set of COMMON LISP functions callable from the emulator. This setup was good for $\mathcal{M}$'s development but ran too slowly for much serious language development. The implementation of $\mathcal{M}$ was then moved to microcode, and the EU and SMU implementations were redone. This first full emulator, like its predecessor, shared microcode space with the LISP system of a normal Explorer.

The most recent development is a port of the emulator to a specially modified Explorer system (built by John Pezaris) that can contain multiple processors. In its current incarnation, this system runs with two processors (one running a normal Explorer LISP system and serving as host, and the other running only the $\mathcal{L}$ emulation) and emulates a single $\mathcal{L}$

[1]For example, Marsland and Demco [Marsland78] present a study in which the QM-1 emulated a PDP-11/10 with a 50% performance degradation.
[2]Work on this initial version was done by LaMott Oren.

processor.[3] The emulator code in use on the multiprocessor system and the single processor system are similar, but the multiprocessor system is simpler in many ways. The remainder of this section describes the details of this multiprocessor emulation, which we call $\mathcal{L}_0$.

5.1.2 System Level Details

The $\mathcal{L}_0$ system is built around a 16 slot NuBus chassis. The standard machine configuration contains two Explorer I processors, two 8 Mbyte memory boards, two I/O boards, one network board, and one disk controller. The LISP system uses one memory and one I/O board, and controls the disk and network. The $\mathcal{L}$ processor uses a memory board and an I/O board. Interprocessor communication in this system is done exclusively via the NuBus. Each processor also has a local bus connecting it to its memory board.

The $\mathcal{L}$ and LISP processors communicate via a 3-level message-passing protocol.[4]. The lowest level of this protocol is used simply to load the correct microcode onto the $\mathcal{L}$ processor; the next level is used for simple diagnostics. The highest level is a shared-memory implementation of a message-passing system.

The $\mathcal{L}$ processor's memory board is divided up into 4 regions: two message queues, local memory space, and permanent memory space. The LISP processor's message queue is located on the $\mathcal{L}$ processor's memory board for simplicity's sake (the LISP processor's memory board is under the control of its virtual memory system). The LISP processor configures the exact layout of the $\mathcal{L}$ memory board when booting the $\mathcal{L}$ processor.

The emulation is interfaced to the user through a LISP control program. This program is essentially a **read compile emulate extract print** loop. An expression is typed by the user, and read in by the program. This expression is passed to a cross-compiler, written in LISP, which produces $\mathcal{L}$ machine code packed into code chunks in the permanent memory of the $\mathcal{L}$ processor. The control program then remotely invokes the emulator on the $\mathcal{L}$ processor, passing it the ID of the STATE to begin execution from. The emulator runs until it reaches the end of the code (or is stopped by a non-proceedable exception), at which time it passes a status code back to the control program. The control program then extracts the value produced by the emulation from the STATE chunk, and prints it out (or else prints an error message). The high-level programming language is similar to SCHEME. Details of

[3]Chapter 6 will describe our plans for using this machine with multiple $\mathcal{L}$ processor emulations.
[4]Much of the development work on this message system was done with Milan Singh.

this system are described in the $\mathcal{L}$ Reference Manual [L87].

The emulator, implemented by approximately 2000 lines of Explorer microcode, is broken up into several modules:

- *bootstrapping*: startup of the $\mathcal{L}$ processor
- *message passing*: communication with the LISP processor
- *diagnostics*: error reporting and debugging support
- *control*: initiation and completion of the emulation, instruction fetch and decode, state management
- *instruction execution*: details of instructions
- *operand access*: dereferencing metanames, importation fault handlers
- *garbage collection*: details of gc
- *exception handling*: what to do if things go wrong
- *accounting*: collection of performance data

Of these modules, only operand access, garbage collection, and exception handling contain important pieces of $\mathcal{M}$.

5.2 Chunks in $\mathcal{L}_0$

Chunks in the $\mathcal{L}_0$ system are mapped onto the flat address space memory provided by the NuBus. This section presents details of chunk representation in $\mathcal{L}_0$ and implementations of the basic memory operations Elt and SetElt.

5.2.1 Chunk Mappings and ID Formats

Local chunks are mapped directly into memory; that is, a local chunk ID is the NuBus address of the first slot of the chunk (neglecting tag bits). Values in $\mathcal{L}_0$ are 33 bits — 32 data field bits and the reference bit. Since the NuBus supports 32 bit words, we split the reference bits of each slot off and keep them in the word holding the attribute bits. Each local chunk occupies a block of 16 words. Words 0 – 8 hold the data fields for slots 0 –

8. Word 9 holds the reference bits for the other slots (including the hidden slots) and the attribute bits. Word 10 is the permanent ID slot, word 11 is the link slot, and word 12 is the forward slot. The last three words are unused. Because the reference bits are packed together into a single word, testing and modification of these bits requires heavy use of bit test, extraction and deposition operations. Most of the required bit operations can be done with a single Explorer microinstruction.

Permanent chunks are laid out similarly, but are not necessarily directly mapped into memory. Thus to go from a permanent chunk ID to an address, the emulator calls a special translation routine. Currently, this routine simply returns the unmodified ID (i.e. permanent chunks are directly mapped); future implementations of $\mathcal{L}$ will use this translation routine to provide virtual permanent chunk management (see section 6.3).

Because the NuBus addresses are byte addresses, and chunks are 64 bytes long, the low 6 bits of a given chunk ID are a constant (zero, in this case). Thus, these bits can be used as tag bits. Currently, there is only one tag bit, which differentiates between a permanent and local ID. There are 26 bits left over for the pointer portion of the ID, so the $\mathcal{L}_0$ system can have up to $2^{26} = 64$ M different chunk IDs.

The emulator needs to perform three principal operations on IDs: calculation of an address, given an ID and a slot number; comparison of IDs; and testing the tag bit of an ID. The ID format describe above allows these three operations to be performed efficiently. A slot address is composed by multiplying the slot number by four (to adjust for byte addressing on the NuBus) and depositing the result in the low 6 bits of the ID. This can be done in a single microinstruction. Testing IDs for equivalence requires one instruction if the IDs are of the same type (both local or both permanent). Testing the attributes of an ID takes one instruction. The $\mathcal{L}_0$ pointer format thus leads to efficient implementations of common pointer operations.

The host LISP processor accesses a chunk by directly reading or writing the memory location in question. Because the $\mathcal{L}$ memory is not in the LISP physical memory map, these operations look like I/O operations to the LISP processor, and so are "untyped". This is an improvement over the single processor system, which expended considerable effort in cooperating with the tagging conventions of the LISP machine.

```
(defun elt (chunk slot)
    (cond ((local? chunk)
           (simple-elt chunk slot))
          ((permanent? chunk)
           (simple-elt (import chunk) slot))
          (t (error "bad id"))))

(defun setelt (chunk slot value)
    (cond ((local? chunk)
           (simple-setelt chunk slot value))
          ((permanent? chunk)
           (simple-setelt (import chunk) slot value))
          (t (error "bad id"))))

(defun local? (bits)
    (and (= 1 (reference-bit bits))
         (= 1 (local-tag-bit bits))))

(defun permanent? (bits)
    (and (= 1 (reference-bit bits))
         (= 0 (local-tag-bit bits))
```

FIGURE 5-1: Elt AND SetElt IN $\mathcal{M}$

5.2.2 Elt and SetElt

Operand accesses in $\mathcal{L}$ machine instructions are specified in terms of MetanameFetch and MetanameStore. As described in chapter 2, the compound memory operations MetanameFetch and MetanameStore are built out of the simpler Elt and SetElt operations. Because the EU is only allowed to dereference through local chunks, Elt and SetElt must check the type of the chunk ID they have been passed, and invoke importation if necessary. Figure 5-1 gives an idea of how this can be done by examining the tag bits in each ID.

In this formulation, `simple-elt` and `simple-setelt` are primitive routines that deal directly with the low-level implementation of local chunks. To hold 33 bit values, these routines use two registers for each value. The first register holds the data bits, and the second holds the reference bit in its lsb. Figure 5-2 shows `simple-elt` and `simple-setelt`; the angle brackets in the argument lists denote the breakup of the 33 bit values.

The code in figure 5-2 is still somewhat simplified, because there are other exceptions that can arise. Chunk attributes must be checked during both `simple-elt` and `simple-setelt` operations to detect locked, forwarded, and read-only chunks, so both of these routines must read the reference bits of the chunk.

```
(defun make-address (data slot)
    (dpb (* 4 slot) (byte 6 0) data))

(defun simple-elt (<i-data i-ref> <s-data s-ref>)
    (if (= 0 (i-ref)) (error "bad id"))
    (if (or (<= 0 s-data 8)
            (= 1 s-ref))
        (error "bad slot number"))
    (read (make-address i-data s-data)))

(defun simple-setelt (<i-data i-ref> <s-data s-ref> <v-data v-ref>)
    (if (= 0 (i-ref)) (error "bad id"))
    (if (or (<= 0 s-data 8)
            (= 1 s-ref))
        (error "bad slot number"))
    (write (make-address i-data s-data) v-data)
    (set-ref-bit i-data s-data v-ref)
    (set-dirty-bit i-data s-data))
```

FIGURE 5-2: `simple-elt` AND `simple-setelt`

Another complication that affects this code is pointer updating. Recall from chapter 4 that importation provides an equivalent local ID for some permanent ID. After an importation, pointer updating is supposed to replace the old permanent ID with the new local ID so that future accesses will use the local ID. But Elt is passed a "disembodied ID" as an argument; Elt (and hence `import`) have no idea where the ID has come from.

$\mathcal{M}$ contains a system which is able to properly update these disembodied IDs in most cases.[5] A small (one ID) cache remembers the last permanent ID fetched in an Elt operation and the location (chunk and slot) that the ID was fetched from. After finding a local ID, but before returning, `import` compares the ID it was passed against this remembered ID. If they match, then an update can occur; `import` writes the local ID into the remembered location. The update cache is then invalidated. The update cache is also invalidated by SetElt if the write done by SetElt is to the remembered location.

A one ID update cache is not large enough to correctly handle all name updates. In $\mathcal{L}_0$, a large fraction of Elts occur as part of a metaname operation, where an ID fetched in step i is dereferenced in step $i+1$. In this case, a one ID cache is sufficient to catch most of the updates.

[5]Not updating an ID does not compromise the storage abstraction, but will lead to unnecessary trivial importations and poor estimations of the working set.

```
(defun mark (root)
   (when (local? root)
       (set-mark-bit-in root)
       (let ((mark-stack (list root))
            ((current-chunk))
         (loop
           (setq current-chunk (pop mark-stack))
           (do-slots (s current-chunk)
               (when (and (pointer? s)
                          (local? s)
                          (unmarked? s))
                    (set-mark-bit-in s)
                    (push s mark-stack))
           (if (null mark-stack) (return)))))))
```

FIGURE 5-3: MARK PHASE

The code for **simple-elt** and **simple-setelt** in figure 5-2 also performs range checks on the slot number. If, as shown in the figure, the range of IDs is restricted to the data and type slots of the chunk, then the hidden slots are truly hidden; they are completely inaccessible to $\mathcal{L}$ programs. This is fine for local memory, which is managed at a low level, but we might wish to write high level $\mathcal{L}$ programs to manage outer levels of the storage hierarchy. One way of handling this would be to create a supervisor mode in which the hidden slots became visible, and carefully code the supervisor routines so that the storage abstraction is never violated.

5.3 Garbage Collection

The garbage collection emulator module is the heart of the $\mathcal{M}$ system. It is broken up into several submodules — mark, first sweep, second sweep, importation, and allocation.

5.3.1 Marking

Marking proceeds basically as outlined in figure 3-1, except that only local pointers are followed. Because marking does not cause object swapping, there is no inherent reason to choose either breadth or depth first marking; $\mathcal{M}$ marks depth-first. The mark stack is distributed through the link slots of chunks. This leads to a precondition on marking: no markable chunk can be using its link slot before marking begins. Figure 5-3 gives a LISP implementation of $\mathcal{M}$'s marking procedure.

```
(defun sweep-1 ()
   (do-all-chunks (c)
       (when (and (unmarked? c)
                  (local-permanent? c)
                  (dirty? c))
          (do-slots (s c)
             (when (and (pointer? s)
                        (local-temporary? s))
                (promote s))
                (if (unmarked? s)
                    (mark s)))))))
```

FIGURE 5-4: FIRST SWEEP PHASE

Because the reference bits for the entire chunk are clustered into a single word, the `pointer?` test in figure 5-3 does not require reading the slot contents. On the other hand, the `local?` test does require fetching the slot contents (a pointer) so that its tag bit can be examined, and the `unmarked?` test requires fetching and dereferencing the pointer.

5.3.2 First Sweep

After marking completes, the garbage collector begins the first sweep pass. In this pass each local chunk is examined to see if it contains a backwards pointer that needs to be used in an auxiliary mark. Because local temporary chunks must be pointed to by local pointers, backwards pointers that reside in marked local permanent chunks have already been marked through in the mark phase. Thus the first sweep phase need only consider unmarked local permanent chunks. Code for this phase is presented in figure 5-4.

The three tests in the outer `when` clause can all be done with information from the attribute slot of the chunk. The test `local-temporary?` requires dereferencing a chunk ID and reading the attribute bits. The inner `when` clause detects backwards pointers that need to be marked from to preserve local temporary chunks. The routine `promote` changes the status of a chunk from local temporary to local permanent. As a side effect of `promote`, the original local ID is replaced with a permanent ID.

At the end of the first sweep we can make two assertions. First, all unmarked local temporary chunks are truly garbage, and can be reclaimed. Second, all unmarked local permanent chunks contain no local temporary IDs, and so can be limboized.

```
(defun sweep-2 ()
   (do-all-chunks (c)
       (cond ((and (unmarked? c)
                   (local-temporary? c))
              (push-on-free-list c))
             ((unmarked? c)
              (if (dirty? c)
                  (write-out c))
              (push-on-limbo-list c))
             ((local-temporary? c)
              (if (> (incf (age c)) threshold)
                  (promote c)))
             (t (replace-ids-in c)))))
```

FIGURE 5-5: SECOND SWEEP PHASE

5.3.3 Second Sweep

The second sweep (see figure 5-5) re-examines each local chunk. Unmarked local temporaries have all their reference bits cleared (to destroy any possible dangling pointers) and are threaded onto the free list. Unmarked local permanent chunks are limboized — if the chunk is dirty, then the associated permanent chunk is updated with an appropriate (permanent-ID only) version of the new contents; then the local permanent chunk is threaded onto the limbo list. Marked local permanent chunks are subjected to ID replacement: all their local permanent IDs are replaced with equivalent permanent IDs (recall the discussion in section 4.4.2). Marked local temporary chunks are aged, and promoted if over the age threshold.

It is possible to merge the two sweeps into a single sweep. The only danger in not performing the auxiliary marks triggered by the first sweep is that after the mark phase there are some unmarked local temporary chunks that are not garbage. A unified sweep would need to provisionally reclaim unmarked local temporary chunks. Limboization could then be responsible for identifying those local temporary chunks that would have been marked by an auxiliary mark in the two-sweep scheme. At the end of a unified sweep a free list cleanup could then remove the non-garbage chunks from the free list. A unified sweep is probably more efficient than two sweeps because the number of backwards pointers is usually quite small. However, the current system uses two sweeps because the implementation is simpler.

5.3.4 Chunk Allocation and Importation

The final submodule of the garbage collection side of the $\mathcal{M}$ system deals with chunk allocation and importation. Importation has already been discussed in detail in section 4.3.1, but there are a few subtle points to cheap importation. Allocation from the free list is fairly straightforward, but allocation from the limbo list is more complex, because of cheap importation. This section thus concentrates on cheap importation.

When a chunk is limboized, the local and permanent chunks still remember each other (see figure 4-5). Limboization of a chunk in a local memory releases the ownership of the corresponding permanent chunk. The owner field of a permanent chunk is actually split into two fields — one for the current owner, and one for the former owner. When a permanent chunk is imported into a local memory, that memory is made the current owner of the chunk. Limboization copies the identity of the current owner to the former owner field, and erases the current owner field. If the local copy is allocated for another purpose before the chunk in question is cheaply imported, then the former owner field is erased, and the associative links are destroyed. If the chunk is to be cheaply imported instead, then $\mathcal{M}$ checks the to make sure that either the owner id's match, or the former owner id matches and the association is valid. Operations on the owner field of permanent chunks must be done in an atomic manner because there can possibly be many local memories sharing a permanent memory.

Because cheap importation can steal chunks off of the limbo list, allocation of free chunks from the limbo list is more complicated. It is expensive to guarantee that the limbo list only contains free chunks, because of the cost of deleting a random element from a singly linked list. Cheap importation therefore simply sets the allocated bit in each chunk it reclaims. When allocating chunks from the limbo list, $\mathcal{M}$ skips over chunks that have already been allocated by cheap importation. Cheaply imported chunks are the only non-garbage local chunks which keep important information in their link slot. The mark phase of garbage collection uses the link slot, so before garbage collection can begin, all cheaply imported chunks must be removed from the limbo list. This is called a limbo list cleanup. The current system does not garbage collect until local memory is full, so it does not have to worry about cleaning up the limbo list.

data movement	move and movei
arithmetic and logical	add, and, etc.
tests	to affect condition codes
conditionals	create boolean values from cc's
bit manipulation	ldb, dpb, rotates, shifts
chunk allocation	alloc
inter-state control	call, return, and activate
intra-state control	jumps and conditional branches
exceptions	traps of various kinds
data structure support	for lists, structures, and arrays
i/o	NuBus operations, chunk-blt

Table 5.1: $\mathcal{L}_0$ EU INSTRUCTION CLASSES

5.4 The EU and SMU

As mentioned at the start of this chapter, the $\mathcal{L}_0$ emulator also implements the EU and SMU. This section presents some of the details of those systems. It is important to note that the EU presented here is somewhat outdated, and the SMU used by $\mathcal{L}_0$ lacks many important features.

5.4.1 The EU

The EU in the $\mathcal{L}_0$ system can execute about 55 different instructions, grouped into 11 classes (see table 5.1). Most of these instructions are in a one or two-operand format. Ldb, dpb, and some of the data-structure instructions are three-operand instructions. Simple instructions are encoded into 16 bits: 6 bits for an opcode and 5 bits each for source and destination operands. The 5 bit operand field can directly encode all length one metanames and most length two metanames. Longer metanames are encoded in an extra 16 bits immediately following the instruction.

$\mathcal{L}_0$'s EU has a built-in exception handler. No instruction is allowed to perform observable side effects until it is guaranteed to complete without taking an exception. If an instruction does cause an exception, the EU simply resets itself, aborting the instruction. The exception is then handled and (assuming the exception is proceedable) execution resumes with the instruction that caused the exception. This can result in a considerable waste of internal state: a three operand instruction with each operand specified by a length four metaname. This instruction can cause up to eleven importation faults (one on the STATE chunk, one on the CODE chunk, and nine on the operand accesses). If this instruction is begun when there are fewer than eleven free chunks, it may abort after importing ten chunks (note that allocation of a chunk is not a side effect).

Raw emulation speed on simple instructions (with short metanames) is about 0.2 Mips, or 5 μs per instruction. Instruction fetch and decode takes about 1 μs, and operand accessing takes the rest of the time. The Explorer microcode cycle time is 286 ns with a two-level pipeline; so 5 μs corresponds to roughly 35 microinstructions, but many of the cycles are taken up by wait states for memory operations. The compiler produces code with very long metanames (which cause many memory cycles per instruction), so compiled code runs at only about 50 Kips.

5.4.2 The SMU

The SMU's job is to keep track of runnable states. In $\mathcal{L}_0$, runnable states are pointed to by *task* chunks, and all task chunks are kept in a doubly-linked ring. A register in the Explorer processor, `M-TASK`, points to the task chunk corresponding to the currently executing state. `M-TASK` is also used as the root ID in the mark phase of garbage collection. The emulator runs a fixed number of instructions from the states pointed to by a task chunk before moving on to another runnable state. Certain instructions, `activate`, `deactivate`, and `suspend`, remove or add task chunks. `Calls` and `returns` do not affect the task ring but instead swap state IDs inside of a task.

5.5 Some Results

The benchmark data presented here is not intended to demonstrate realistic speed characteristics of either $\mathcal{M}$ or $\mathcal{L}$. There are several sources of inaccuracy. First, the execution times in the emulator do not accurately reflect the performance that would be possible in a custom $\mathcal{L}$ system. Second, the compiler used in this benchmark produces very low-quality code. In particular, the activation record size generated by this compiler is several times larger than necessary. Because these activation records are built from chunks, the dynamic fraction of allocation instructions is approximately 14%. This forces $\mathcal{M}$ to do a lot more work than it should, and hence makes $\mathcal{M}$ look slow. Third, the EU speed of this system is unrealistically slow, meaning that the mean percentage of time spent executing instructions is too high (this makes $\mathcal{M}$ look better than it is). Finally, `Hilbert` and the other benchmarks are very small programs that do not really test the virtual memory aspects of $\mathcal{M}$ with much rigor. Larger and more realistic benchmarks are in the works, but depend on some ongoing compiler development.

The benchmark information is not without value, however. The benchmark programs do provide good examples of the ephemeral garbage collection aspects of $\mathcal{M}$, and they give achievable (but not very remarkable) points on the performance curve. The benchmark data indicates that at least some of the design decisions and assumptions behind $\mathcal{M}$ are borne out in practice. Even so, it would be misleading to extrapolate very far from such a small and biased set of data points.

5.5.1 The Benchmarks

The data presented in this section was collected by running three different benchmark programs. During these benchmarks, the emulator maintained about 20 event counters, a timer (real time) for each of the garbage collection phases, and a timer for the total run time. Each test was run from a "cold-start" configuration, with all STATE, code, and data chunks initially located in permanent memory (as in the example in chapter 4). The only parameter that varied in this study was the size of the local memory: it ran from 250 to 10000 chunks in 74 steps.

`Hilbert`

The `Hilbert` benchmark was designed to test the ability of $\mathcal{M}$ to cope with large numbers of ephemeral activation records. `Hilbert` contains four mutually recursive functions that together draw Hilbert curves (figure 5-6 shows the first through sixth order curves). The high-level $\mathcal{L}$ code for `Hilbert` is given in figure 5-7. The benchmark program drew the first through seventh order Hilbert curves in succession.

`Starburst`

The second benchmark, `Starburst`, draws several star-like patterns on the screen. `Starburst` is iterative, not recursive, and so does not have as large of a working set as `Hilbert`.

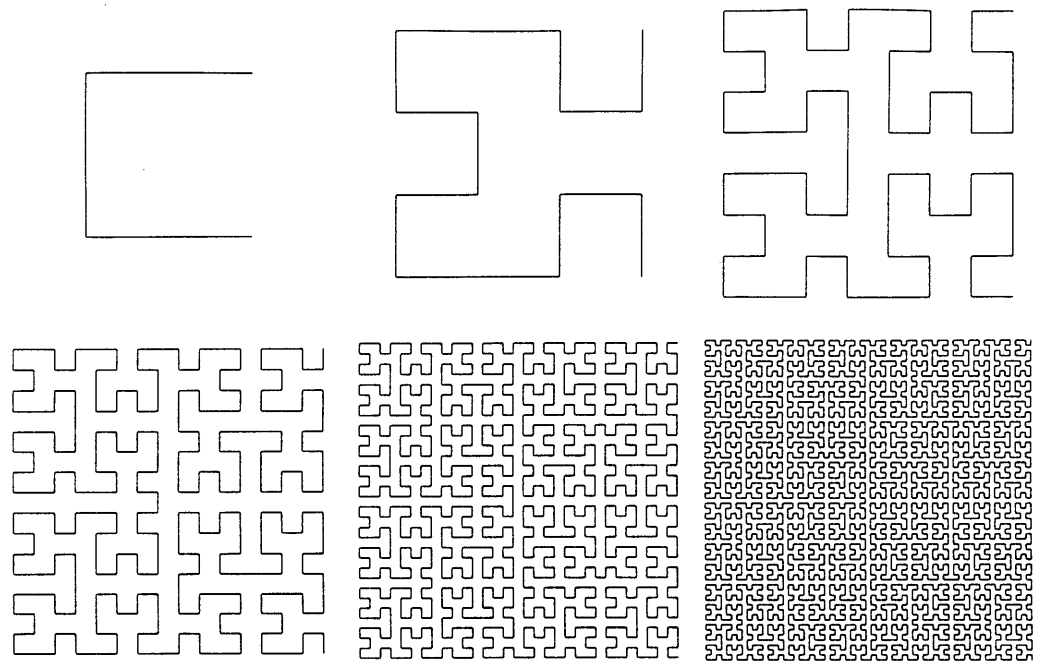


FIGURE 5-6: FIRST THROUGH SIXTH ORDER HILBERT CURVES

Tree

Hilbert and Starburst only allocate chunks to create new activation records. Since the two programs run in strict lifo order, these activation records become garbage quickly. The Tree benchmark is intended to measure the cost of slightly longer term storage. Tree inserts and deletes nodes from binary tree, so it deals with a data structure that persists longer than a typical activation record.

5.5.2 Overall Results

The benchmarks confirm some general impressions of how $\mathcal{M}$ works. For a given benchmark, the performance statistics all show roughly matching "knees" located at some value of the local memory size. (see, for example, figure 5-8). We can interpret this value as an estimate of the working set size of the benchmark. Local memories larger than this working set size all have comparable statistics. Smaller local memories cause $\mathcal{M}$ to work much harder.

The time required to garbage collect follows the same pattern (see figure 5-9). For sufficiently large local memories, garbage collection requires around 3% of the total run

```
(define (Hilbert (order %integer%))
  (block
    (define h %integer% 512)
    (define x %integer% 0)
    (define y %integer% 0)
    (define x0 %integer% (truncate h 2))
    (define y0 %integer% x0)
    (define (plot)
      (block(draw-line x0 y0 (- x x0) (- y y0))
        (setq x0 x)
        (setq y0 y)))
    (define (a (i %integer%))
       (if (> i 0) (block (d (1- i)) (setq x (- x h)) (plot)
                          (a (1- i)) (setq y (- y h)) (plot)
                          (a (1- i)) (setq x (+ x h)) (plot)
                          (b (1- i)))))
    (define (b (i %integer%))
      (if (> i 0) (block (c (1- i)) (setq y (+ y h)) (plot)
                         (b (1- i)) (setq x (+ x h)) (plot)
                         (b (1- i)) (setq y (- y h)) (plot)
                         (a (1- i)))))
    (define (c (i %integer%))
      (if (> i 0) (block (b (1- i)) (setq x (+ x h)) (plot)
                         (c (1- i)) (setq y (+ y h)) (plot)
                         (c (1- i)) (setq x (- x h)) (plot)
                         (d (1- i)))))
    (define (d (i %integer%))
      (if (> i 0) (block (a (1- i)) (setq y (- y h)) (plot)
                         (d (1- i)) (setq x (- x h)) (plot)
                         (d (1- i)) (setq y (+ y h)) (plot)
                         (c (1- i)))))
    (dotimes (i order)
      (setq h (truncate h 2))
      (setq x0 (+ x0 (truncate h 2)))
      (setq y0 (+ y0 (truncate h 2))))
    (setq x x0)
    (setq y y0)
    (a order)))
```

FIGURE 5-7: HILBERT CURVE PROGRAM

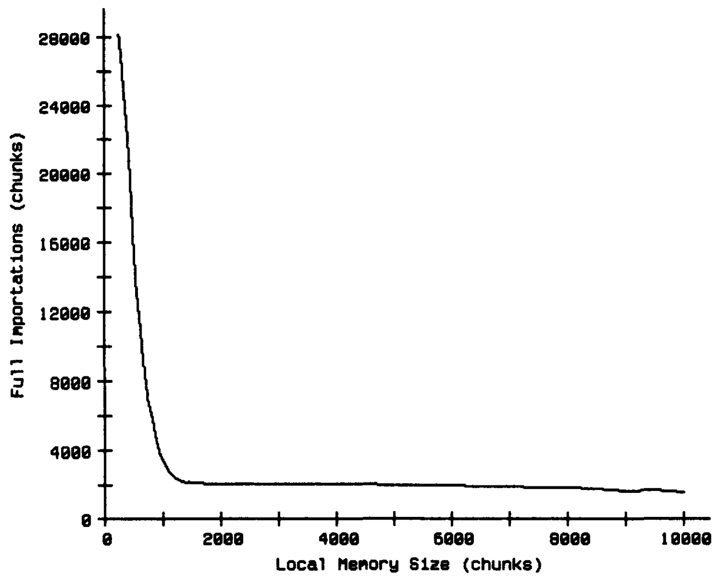


FIGURE 5-8: TYPICAL BENCHMARK CURVE

time. For very small local memories, garbage collection time can be as much as 25% of the run time, depending upon the benchmark. In this latter case it is not really fair to charge all of this time to the garbage collector, since very small local memories force $\mathcal{M}$ to do a lot of virtual memory work in the garbage collection cycles (recall that limboization and working set estimation both happen during garbage collection). In the limit of small local memories, $\mathcal{M}$ thrashes just like traditional virtual memories do, because the working set does not fit into the local memory.

The above results lead to the question of just how big the local memory should be. This question is similar to the question of how big a physical memory is needed in a virtual memory system. The answer is simply that the memory size should be such that the cost savings in adding more memory is exactly offset by the cost of that memory. Of course, this depends critically upon how cost is measured. In the benchmarks presented here, there is no advantage to having a local memory of more than about 2K chunks, but as we have noted, these benchmarks are tiny programs. A "workstation" type $\mathcal{L}$ machine would need a much larger local memory, perhaps 128K chunks.

It is tempting to look at the performance of $\mathcal{M}$ running these tiny benchmarks in very small local memories as indicative of how large programs would behave with reasonable amounts of local memory. In other words, for a given benchmark, we can divide the local memory size by the estimated working set size and get a parameter ρ that measures the fit

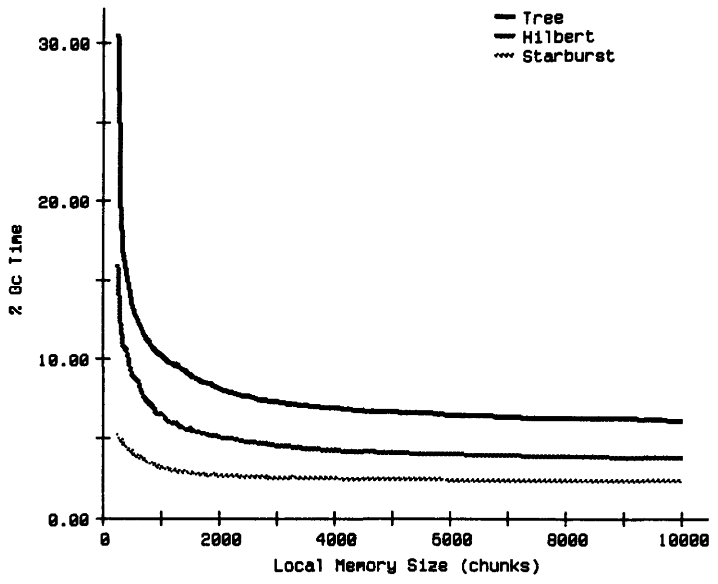


FIGURE 5-9: FRACTION OF RUN TIME SPENT IN GARBAGE COLLECTION

of the benchmark to the local memory. If $\rho \gg 1$ then we are in a "large memory" regime; in this regime overall performance is not limited by the memory system, and the performance curves should be independent of ρ. When $\rho \approx 1$ or less, we are in a "small memory" regime; here performance depends on ρ. A ρ-based generalization is probably not a very good one, because the behavior of large programs is much more complex than small ones, but it does give a rough idea of what could happen.

5.5.3 Effectiveness of Ephemeral Collection

`Hilbert` allocates roughly 15 million chunks during its run, all of which eventually become garbage. Figure 5-10 illustrates the number of permanent chunks allocated during `Hilbert`. Even with a local memory as small as 250 chunks, only about 0.5% of temporary chunks survive and get promoted to permanent status. For larger local memories this percentage falls off to 0.02%. Thus, $\mathcal{M}$ is able to collect a large fraction of the garbage created by `Hilbert` in the local memory. The other benchmarks show similar results. Small local memories ($\rho \approx 1$) force $\mathcal{M}$ to make a decision about the permanancy of a chunk too soon, and so $\mathcal{M}$ guesses incorrectly more often. For large local memories, $\mathcal{M}$ is able to wait somewhat longer and is able to guess more accurately.

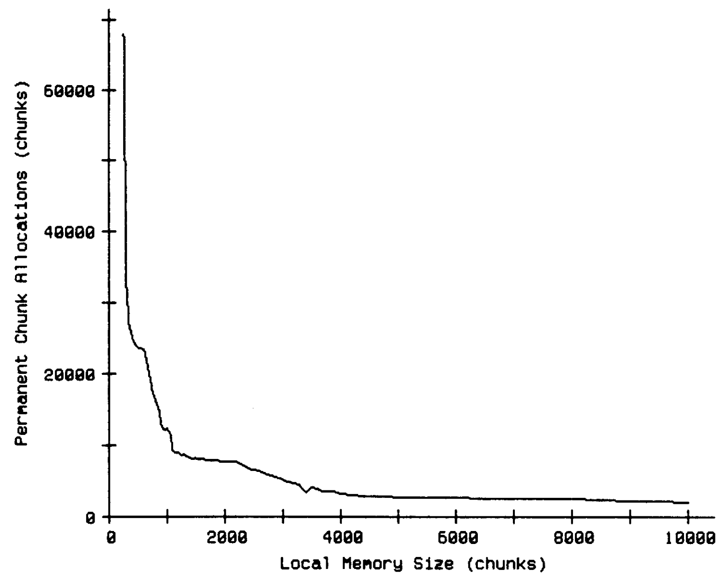


FIGURE 5-10: NUMBER OF PERMANENT CHUNKS ALLOCATED DURING Hilbert

5.5.4 Hit Ratio, Importation and the Usefulness of Limbo

We can define the ***hit ratio*** in $\mathcal{M}$ as the fraction of Elts and SetElts which do not take an importation fault. Since all metaname operations start at the STATE chunk kept in the EU, and this ID is explicitly made local during the instruction fetch, no metanames can suffer faults on their first link (but subsequent links can cause importation faults). Because $\mathcal{M}$ is not responsible for this, we amend the above definition of the hit ratio: the ***primary*** hit ratio is the fraction of Elt and SetElt operations that do not take an importation fault and are not simply passed the STATE as an ID. This version of the hit ratio will measure how well $\mathcal{M}$ can keep local memory filled with interesting chunks. As the data from Hilbert shows in figure 5-11, the primary hit ratio (solid line) varies from 96.5% to 98.5% as the local memory size grows.

Because importation faults can be handled in three different ways, depending upon circumstances, we can also define other hit ratios; for example, the ***secondary*** hit ratio is simply the primary hit ratio plus the fraction of Elts and SetElts that cause a trivial importation. The secondary hit ratio, also shown on figure 5-11 (dotted line), varies from 98.5% for a small local memory to 99.9% for a large local memory.[6]

Figure 5-12 shows the distribution of importation fault handling among trivial, cheap,

[6]The primary hit ratio levels off at 98.5% because there are a few cases of name updating that the one ID update cache does not detect. One of these occurs in the calling sequence used by the $\mathcal{L}_0$ compiler.

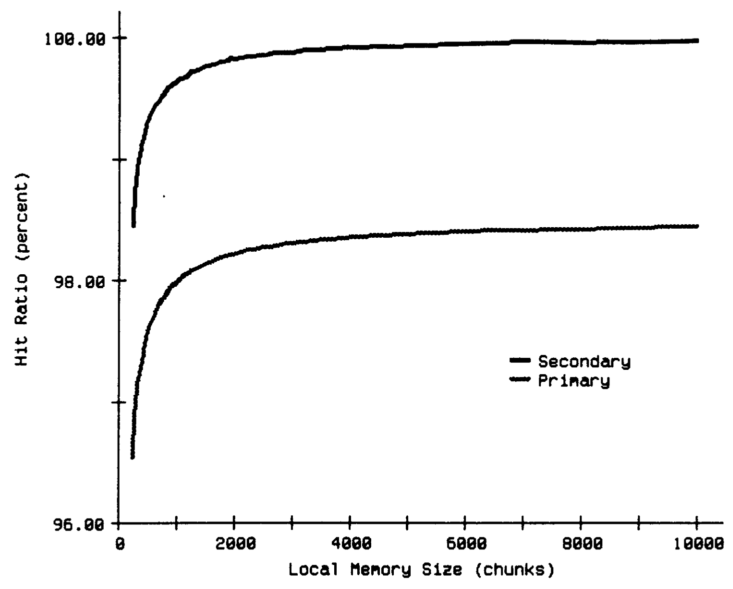


FIGURE 5-11: HIT RATIO

and full importations. Note that cheap and full importation faults are much rarer than trivial importations. Cheap and full importations never happen in more than about 10% of importation faults; typical percentages are much lower. The maximum number of cheap and full importations comes when the local memory size is small. One question raised in chapter 3 was whether the cost of maintaining the limbo list is worth the potential gains of cheap importation. The answer to this question, based upon the benchmark data, is a qualified no.

The disadvantages of maintaining a limbo list and allowing cheap importation are that it greatly complicates garbage collection, importation, and chunk allocation (these complications occur mainly in the form of additional *cases* that each module must consider, and so do not directly impact the performance of the other, more frequently used cases, like trivial importation). The main advantage of cheap importation is that it saves us from having to copy the contents of a chunk from permanent to local memory. Because cheap importations are relatively rare, and the checks for when cheap importation is permitted are about as costly as copying a chunk's worth of data from one place to another, cheap importation does not provide any benefits to the current system.

The qualifications to this conclusion are twofold. First, cheap importations rise dramatically as the local memory size is made smaller ($\rho \approx 1$).The second qualification concerns the cost of full importation. The current $\mathcal{M}$ system does not model all the aspects of permanent

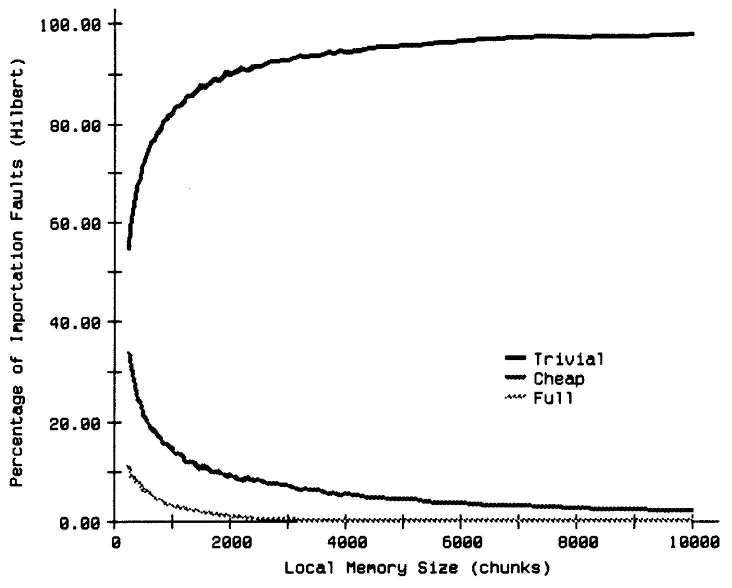


FIGURE 5-12: DISTRIBUTION OF IMPORTATION FAULTS

memory. We have preliminary plans (discussed further in chapter 6) for a virtual permanent memory; under such a model permanent chunk access would be much more expensive than it is now. Thus, cheap importation may be valuable in the future.

5.5.5 Cost of Temporary Storage

On a flat address space architecture, stacks are an optimum structure for implementing block-structured languages. In $\mathcal{L}$ this stack must be built out of chunks. Because stack storage is used only temporarily, simulation of a control stack with chunks causes large amounts of chunk allocation and deallocation. These operations are potentially much slower than stack pushes and pops, and could substantially limit the performance of $\mathcal{L}$.

There are several possible solutions. One is to have the compiler detect when an activation record is no longer needed and explicitly deallocate it, perhaps returning it to a special activation record pool. This solution is complicated by multichunk activation records; it is possible that the number of instructions needed to explicitly deallocate a multi-chunk structure could take longer than the garbage collector takes to reclaim the same structure. We thus rely on the garbage collector to efficiently reclaim activation records.

A separate experiment was run on the $\mathcal{L}_0$ system to measure the cost of temporary storage. This experiment compared the execution time of the two loops shown in figure 5-13. The code on the right allocates a chunk each time around the loop, and then destroys

```
       (movei 500000 (2))                      (movei 500000 (2))
loop-0 (movei 0 (3))                    loop-1 (alloc (3))
       (movei 0 (3))                           (movei 0 (3))
       (inc -1 (2))                            (inc -1 (2))
       (jgt loop-0)                            (jgt loop-1)
```

FIGURE 5-13: LOOPS FOR MEASURING COST OF TEMPORARY CHUNKS

the reference to it. The code on the left executes the same number of instructions with the same instruction sizes but does not allocate any chunks. The run times of the loops should be similar except for the time needed to execute the `alloc` instruction. By subtracting the run times for these two loops, we can therefore estimate the cost of using a chunk temporarily.

The experiment was run for both 200,000 and 500,000 loop cycles. The difference in run times divided out to about 13.5 μs per cycle, implying that the cost of allocating and then implicitly deallocating a chunk on the emulator is 13.5 μs. This is about the same amount of time that it would take to run the two instructions that would be needed to allocate and deallocate the chunk. Thus, on this emulator at least, it is just as time-efficient to let the garbage collector discover and reclaim an inaccessible chunk as it is to try and explicitly return the chunk to the free list via some type of deallocation instruction.

5.5.6 Summary of Benchmark Results

The benchmark results presented in the preceding sections were intended to demonstrate the following points about $\mathcal{M}$:

- The performance of $\mathcal{M}$ depends on many factors. The most important of these are the local memory size and the working set sizes of the programs. Small values of ρ, the ratio of estimated working set size to local memory size, force $\mathcal{M}$ to adopt a more virtual memory-like aspect.

- The garbage collection algorithm used by $\mathcal{M}$ does not appear to cause a performance bottleneck unless ρ is very small. The performance degradation that occurs for small ρ is a manifestation of the limitations of a virtual memory system, and not an integral problem with the garbage collector.

- Ephemeral garbage collection in $\mathcal{M}$ works well. Its effectiveness declines somewhat as ρ does, because small local memories force $\mathcal{M}$ into making premature decisions about the temporality of chunks.
- Temporary storage is inexpensive. The garbage collector requires about as much time to reclaim a temporary chunk as the EU would require to run an instruction to reclaim the chunk.

$\boxed{6}$ Future Work

$\mathcal{M}$ is rather preliminary in many ways. Because the $\mathcal{L}$ architecture is not yet well-defined, future developments may require extensive modification or redesign for $\mathcal{M}$. This section explores some of the possible directions that work on $\mathcal{M}$ and related systems could take.

6.1 Prediction of the Working Set

As mentioned in section 4.4.2, the way that $\mathcal{M}$ estimates the working set is rather primitive. The combination of replacement of local permanent IDs by equivalent permanent IDs during the second sweep and trivial importations effectively implements a simple variant of the working set page replacement algorithm [Baer80]. Just before a garbage collection, the local permanent chunks that are locally accessible are exactly those that either were dereferenced through in the processing phase preceding the garbage collection or are pointed to by temporary chunks. Just after a garbage collection, no local permanent chunk is locally accessible from another.

We can do away with pname updating and subsequent trivial importation by correctly tracking which local permanent chunks are in the working set. One way to do this is to include a *visited* bit in each chunk. This bit is cleared by the sweep and then set if the chunk is used in an Elt or SetElt operation. When we examine memory just before garbage collection, all visited chunks are also locally accessible (thanks to importation), but some locally accessible chunks have not been visited. These latter chunks are out of the working set estimate, and should be exportable. The difficulty here is that in order to export a chunk, all local IDs referring to that chunk must be replaced. This requires checking all the local IDs in a visited local permanent chunk to see if they point to non-visited chunks, and updating them if they do. All IDs in non-visited local permanent chunks would be updated as well.

If we drop the requirement that all non-visited local permanent chunks be exportable, then the above method is simplified; we now only need to update IDs in non-visited local permanent chunks. This allows some of the current non-visited chunks to be exported after two garbage collections: the first to replace IDs and make these chunks locally inaccessible, and the second to actually limboize them.

The decision about whether or not $\mathcal{M}$ needs an improved working set estimate cannot be made without realistic statistics about $\mathcal{M}$'s performance. Our emulation does not collect the necessary information to extrapolate the real-time performance of $\mathcal{M}$, and our language technology does not provide the ability to test $\mathcal{M}$ on realistic benchmark programs. Both of these barriers will hopefully be removed in the near future.

6.2 Multiple Processors and Memories

Our current model for $\mathcal{L}$ includes provisions for many processors and many memory banks. $\mathcal{M}$ does include some preliminary features intended to support multiple processors and other memories. There are many different ways that a system like this can be organized.

A single local memory can support multiple EUs. The synchronization between EUs can be done with the lock primitive. The number of EUs that can operate from a single local memory will be primarily determined by the bandwidth requirements that the memory is able to meet. A slow local memory may only be able to support a few EUs.

Multiple local memories can share a permanent memory. Synchronization among local memories can be accomplished by the ownership protocols for permanent chunks. Our ownership mechanism does not currently allow for shared access to permanent chunks.

There are many problems that need to be examined in a multiprocessing system — interprocessor communication, distribution of work among processors, garbage collection of shared memory areas, to name a few. We hope that the $\mathcal{L}$ processor emulations will allow us to conduct reasonable experiments to measure the effects of different architectural decisions.

6.3 Management of Permanent Memory

As mentioned in chapter 3, $\mathcal{M}$ manages just one (well, maybe one and a half) levels of the $\mathcal{L}$ memory hierarchy. This section presents some preliminary ideas towards the structure of the manager for the next lower (more permanent) level of memory.

A permanent memory will be shared by some number of local memories. Because of this it will have to implement some of its activities in atomic fashion (for example, test of and modifications to the owner field of permanent chunks). Permanent chunks will probably be organized somewhat differently than local chunks (in the current system, they are very similar) because the additional information used by $\mathcal{M}$ will only need to be kept for the small fraction of permanent chunks actually owned by one local memory or another. Chunks may be a good unit for memory management when the memory implementation is RAM, but mass storage devices like disks do not efficiently deal with individual transfers of this small an amount of storage. Because a permanent memory will interface to mass storage devices, there needs to be some way to cluster chunks together into cohesive units that can be transferred to and from the disks.

Garbage collection in permanent memory will hopefully be relatively infrequent. If $\mathcal{M}$ does a good job, only a small fraction of the objects created in the permanent memory will become garbage. Even so, it is probably a good idea for this next level of memory to itself include an ephemeral level, because the number of chunks in permanent memory is much larger, and any garbage collection process that had to examine all the reachable objects would be very slow. Because there are advantages in clustering related objects, and a wide virtual address space, semispace methods may be more appropriate than the methods used by $\mathcal{M}$ in the local memory, even though the fraction of live objects will be high.

6.4 Purely Relative Namespaces

Computer architects have shown a remarkable ability to underestimate the need for address space. Whenever an address space crisis looms on the horizon, architects tack on a few more bits of address and consider the problem solved, since each extra bit doubles the address space. Still, it is tempting to consider a machine architecture that has an effectively infinite address space, so that it never has to be redesigned to accommodate longer addresses.

We feel that metanames are a valid mechanism for building effectively infinite address spaces. A metaname is a kind of *operational* address for an object, and each stage of the metaname specifies local knowledge of addressability but not any global knowledge. As an example, consider an ant that is able to traverse chunk pointers. Given some arbitrary mesh of chunks and a metaname, the ant is able to traverse a path through the mesh. At each chunk, the ant need only know the first element of the metaname to decide where to go. No restriction is placed upon the size of the mesh.

Given a mesh of some size, it is clear that the number of different pointers required is equal to the number of nodes in the mesh. Since pointers must be finite-length, this does restrict the size of the mesh. This restriction can be circumvented by breaking a large mesh up into some number of *domains*, each containing a finite number of local nodes. A pointer that crosses a domain is forced to do so through a domain interface. This interface provides a translation between the ids in one domain and the ids in another.

The creation of domains simply pushes the problem of address space up one level. Instead of an unbounded number of mesh nodes, we now must deal with an unbounded number of domains. Because domain identifiers are invisible to the low-level storage abstraction, they do not have to be designed to fit into a fixed-length location. The interface between local and permanent memories in $\mathcal{M}$ is in some ways similar to a domain interface. In particular, permanent chunk IDs may be in reality much longer than local chunk IDs. What the local memory sees as permanent chunk IDs could be in reality just indexes into the domain interface table.

6.5 Metacaching

EU instructions with long metanames can take a long time to execute. For example, the instruction

```
    (move (2 5 7 4) (2 5 0 3))
```

requires 7 Elts and 1 SetElt. Because the first two links of the source and destination operand metanames are identical, a literal execution of this instruction will do more work than is absolutely necessary. *Metacaching* provides a way of associating metanames with chunk IDs so that the memory system can speed up the dereferencing of metanames.

Assume that we have an empty metacache and we execute the above instruction. During the source phase we fully dereference the source metaname. After each Elt, a record is made in the metacache of a metaname and a terminal ID. At the end of the source phase, the metacache holds a set of four associations:

```
(2)       -- A
(2 5)     -- B
(2 5 7)   -- C
(2 5 4 7) -- D
```

Here we have used A, B, C, and D to denote the chunk IDs encountered during dereferencing. When we enter the destination phase, we note that we have a partial match between the first two elements of the destination metaname and one of the metacache entries. Instead of fully dereferencing (2 5 0 3) we can reach the same end point by dereferencing (0 3) starting from root chunk B.

There is one subtle point to metacaching which makes it difficult. It is possible to have many valid metanames that terminate in the same ID. A MetanameStore may affect many metacache entries. The metacache must therefore keep track of the interdependencies between metanames. One way of doing this is presented in [Singh87]. In this scheme, a MetanameStore invalidates a set of metacache entries which includes the entries directly affected and some other metanames that really did not need to be invalidated. These extra invalidations are due to the encoding of the dependencies.

Another source of complication arises when the root chunk is changed. In a function call, the root chunk gets changed to a nearby chunk (that is, one reachable from the root by some metaname). In this case it may be possible to retain some of the metacache entries by suitably revising the metanames. If the new root chunk is not a nearby chunk, then all the metacache entries must be invalidated. This makes metacaches (in their present form) rather unattractive if the processor switches contexts frequently.

A metacache will work well if there is a lot of commonality among metanames, the metacache is able to keep tight control over the dependency tracking, and the metacache is not thrashed too badly by context switches. We have some sketchy plans of incorporating a metacache simulation in the emulator to test out the feasibility of metacaching.

6.6 New Emulations

The $\mathcal{L}$ architecture emulations will be undergoing many improvements in the near future. The first change will be the adaptation of the emulator to Explorer II processors. This will hopefully speed up emulation by a factor of three to five. The second change will be the development of a new EU instruction set. Our experiences with the current instruction set (and doubts about the usefulness of metacaching) have led us to consider an instruction set in which the processor is restricted to using short metanames in a majority of instructions. Because of the reduced number of memory accesses, this new instruction set will emulate more quickly. The final major change will be the exploration of a multiple-emulator multiprocessor; we will be able to run some limited number (perhaps as many as 6) $\mathcal{L}$ processor emulations concurrently.

$\boxed{7}$ Conclusions

As computing models change, underlying machine structures must change if they are to efficiently simulate those models. This can be thought of as a kind of conceptual eigenproblem — for a given computational model, there are certain machine architectures which lend themselves more naturally to efficient implementations of that model. Because of Turing equivalence, many reasonably sophisticated architectures will be able to simulate a particular model, but some subset of all such architectures will be able to do it better than others. The $\mathcal{L}$ architecture project is an attempt to create a more natural machine architecture for computational models based upon objects, first-class functions, and inherent low-level concurrency.

This thesis has presented the design and implementation of a memory system, $\mathcal{M}$, usable by the $\mathcal{L}$ architecture, that is based upon a simple object model — all objects are created from fixed-sized chunks. The constraints on $\mathcal{M}$ are the chunk basis, some known and intuitive properties of $\mathcal{L}$'s computational model (temporary storage, locality of reference), and some desirable properties for memory (large numbers of chunks, quick access to chunks, cheap temporary storage). $\mathcal{M}$ shares some of these constraints with and draws inspiration from previously implemented object-oriented memory systems.

The main features of $\mathcal{M}$ are its dual namespace (local and permanent memories) and its integration of virtual memory and garbage collection techniques to manage the local memory. The division of memory into these two spaces allows temporary objects to be garbage collected without requiring examination of the larger body of permanent objects; the integration of this garbage collection with the virtual memory system minimizes the disruption of the working set by the garbage collector.

$\mathcal{M}$'s weaknesses lie in several areas. First, the algorithms used by $\mathcal{M}$ may be too complex. For reasonable performance, $\mathcal{M}$ will need to be at least partially implemented at a very low level; there is not much room for complexity in such an implementation. $\mathcal{M}$ can be simplified somewhat (perhaps without seriously degrading performance) by the elimination of the limbo list and cheap importation. Second, the mark/sweep garbage collection techniques used by $\mathcal{M}$ may not be the most efficient. A semispace method may actually work better if there is a consistently large fraction of garbage in the local memory, because semispace methods only need to examine non-garbage objects. Third, $\mathcal{M}$ is not very sophisticated when it comes to estimating the working set. It is not clear whether this is a serious limitation or not. Finally, $\mathcal{M}$ associates a rather large bundle of overhead storage with each local chunk, and thus wastes local memory. Since many of the extra fields are not used by some chunks (e.g. local temporary chunks do not need the associated permanent chunk field), it may be possible to keep the extra information required by $\mathcal{M}$ in special, space-efficient tables.

As for strengths, $\mathcal{M}$ appears to do a good job of handling ephemeral garbage. A large percentage of garbage chunks are reclaimed in local memory, and the reclamation process is reasonably efficient. Temporary storage can be recovered by the garbage collector about as quickly as it could be explicitly freed by the programmer. $\mathcal{M}$ maintains a reasonable local ID hit ratio, indicating that the virtual memory aspects work fairly well. Access to local chunks does not require any searching, hashing, or special-purpose hardware. $\mathcal{M}$ supports special operations like locking and caching chunks.

Bibliography

[Baer80] J. Baer, *Computer Systems Architecture*, Computer Science Press, Rockville, MD, 1980, 355 - 357.

[Baker78] H. Baker, List Processing in Real Time on a Serial Computer, *Communications of the ACM 21, 4*, April 1978, 280 - 294.

[Blair86] M. R. Blair, *A Simulator for the L Architecture In Common Lisp*, B. S. Thesis, Deparment of Electrical Engineering and Computer Science, Massachusetts Insitute of Technology, May 1986.

[Cohen81] J. Cohen, Garbage Collection of Linked Data Structures, *ACM Computing Surveys 13, 3*, September 1981, 341 - 367.

[Courts87] B. Courts, Obtaining Locality of Reference in a Garbage-Collecting Memory Management System, paper submitted to the *Communications of the ACM.*

[Dijkstra78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steffens, On-the-fly Garbage Collection: An Exercise in Cooperation, *Communications of the ACM 21, 11*, 966 - 975.

[Fabry74] R. Fabry, Capability-based Addressing, *Communications of the ACM 17, 7*, July 1974, 403–412.

[Fenichel69] R. Fenichel and J. Yochelson, A Lisp Garbage Collector for Virtual-Memory Computer Systems, *Communications of the ACM 14, 8*, November 1969, 611 - 612.

[Goldberg83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[Halstead84] R. Halstead, Implementation of Multilisp: Lisp on a Multiprocessor, *ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984, 293 - 298.

[Kaehler81] T. Kaehler, Virtual Memory for an Object-Oriented Language, *Byte*, vol. 6, no. 8, August 1981, 378 - 387.

[Kaehler83] T. Kaehler and G. Krasner, LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems, in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, Reading, MA, 1983, 251 - 271.

[Knight79] T. Knight, D. Moon, J. Holloway, and G. Steele, *CADR*, A. I. Memo 528, MIT-AI Lab, Boston, MA, May 1979.

[L87] A. Ayers, R. Saenz, M. Singh, and S. Ward, L Reference Manual, unpublished memo of the MIT Laboratory for Computer Science.

[Liberman83] H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM 26, 6*, 419 – 429.

[Marsland78] T. A. Marsland and J. C. Demco, A Case Study of Computer Emulation, *INFOR*, vol. 16, no. 2, June, 1978, 112 – 131.

[Moon84] D. A. Moon, Garbage Collection in a Large Lisp System, Proc. 1984 ACM Symposium on Lisp and Functional Programming, August, 1984, 235 – 246.

[Rees86] J. Rees and W. Clinger, Eds., *Revised^3 Report on the Algorithmic Language Scheme*, A. I. Memo 848a, MIT-AI Lab, Boston, MA, September 1986.

[Singh87] M. Singh, A CMOS VLSI Implementation of the L Metacache, final report for the class 6.371 (at MIT), Spring 1987.

[Smith82] A. Smith, Cache Memories, *ACM Computing Surveys*, vol. 14, no. 3, September 1982, 473 – 530.

[Steele75] G. L. Steele, Multiprocessing Compactifying Garbage Collection, *Communications of the ACM 18, 9*, 495 – 508.

[Steele84] G. L. Steele, *Common LISP: The Language*, Digital Press, Bedford, MA, 1984.

[Terman85] C. Terman and S. Ward, L Project Proposal, private communication.

[TI84] *EXPLORER Processor Specification*, TI Proprietary Document, Copyright 1984, TI Part Number 2236414, November 6, 1984.

[Ungar84] D. Ungar, Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM Software Eng. Notes / SIGPLAN Notices, Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Ward80] S. Ward and C. Terman, An Approach to Personal Computing, *Proc. Spring COMPCON*, San Francisco, February 1980.

[White80] Jon L. White, Address/Memory Management for a Gigantic LISP Environment or, GC Considered Harmful, *Conference Record of the 1980 LISP Conference*, Stanford, CA, 1980, 119 – 127.