14

LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems

Ted Kaehler
Glenn Krasner
Software Concepts Group
Xerox Palo Alto Research Center
Palo Alto, California

roduction

The Smalltalk-80 virtual machine is specified as a memory-resident system containing up to 2^{15} objects. When full, it typically occupies about 2M bytes of memory. Unfortunately, many machines do not have this capacity in main memory, and many applications require, or will require, more than this capacity. To solve this space problem, one typically uses a virtual memory system in which the resident, "real" memory is used as a cache for the larger mass storage, "virtual" memory. LOOM, Large Object-Oriented Memory, is a virtual memory system designed and implemented for the Smalltalk-80 system. The most important feature of the LOOM design is that it provides virtual addresses that are much wider than either the word size or the memory address size of the computer on which it runs.

LOOM is a single-user virtual memory system that swaps objects and operates without assistance from the programmer. Virtual memory systems may be characterized by the amount of attention that the programmer must pay to the transfers between virtual and real memories, by the extent to which the memory is shared among users, and by the granularity of transfer between memory levels. Overlay mechanisms are an example of systems that require much programmer attention, while all common paging systems require none[1]. Databases may be

viewed as the extreme in allowing sharing; the virtual memory for Interlisp-D[2] is one example of a single-user virtual memory. Most overlay systems transfer program segments, while paging systems transfer disk pages, and a few systems such as the OOZE virtual memory for Smalltalk-76[3] transfer objects.

The LOOM Design

We view virtual memory design as a process of trying to determine what happens most often, making it go fast, and hoping that it will continue to be what happens most often. Our experience with previous Smalltalk systems gave us three major assumptions on which we based the LOOM design: programmers and users have a large appetite for memory, object-swapping is an efficient and effective scheme, and the Smalltalk-80 design for handling resident objects is worth keeping. From these assumptions and the desire to provide a large number of objects on a machine with a narrow word width, we created the major design decisions.

- LOOM assumes that the object is the unit of locality of reference. It swaps individual objects between primary and secondary memory, and allows into main memory only those objects actually needed by the interpreter. Unlike paging systems, LOOM packs objects in main memory at maximum density.
- LOOM is designed for machines with 16-bit words. Fields of objects in main memory are 16 bits wide.
- The address space of the secondary memory is large. LOOM allows as many as 2^{31} objects.
- The interpreter accesses objects in main memory exactly as it does in a resident Smalltalk-80 interpreter. When the necessary objects are already in main memory, the interpreter runs as fast as it did in the resident system.

In order to allow the large number of possible objects, and yet treat the resident objects in the same way they are treated in a non-LOOM Smalltalk-80 implementation, we decided to create two different name spaces. The same object is identified by names from different spaces when it resides in different parts of the system, as shown in Fig. 14.1. The identifier of an object is called an *Oop*, which stands for "object pointer." An object in secondary storage has a 32-bit Oop (a long Oop), and each of its fields containing a pointer to another object holds that

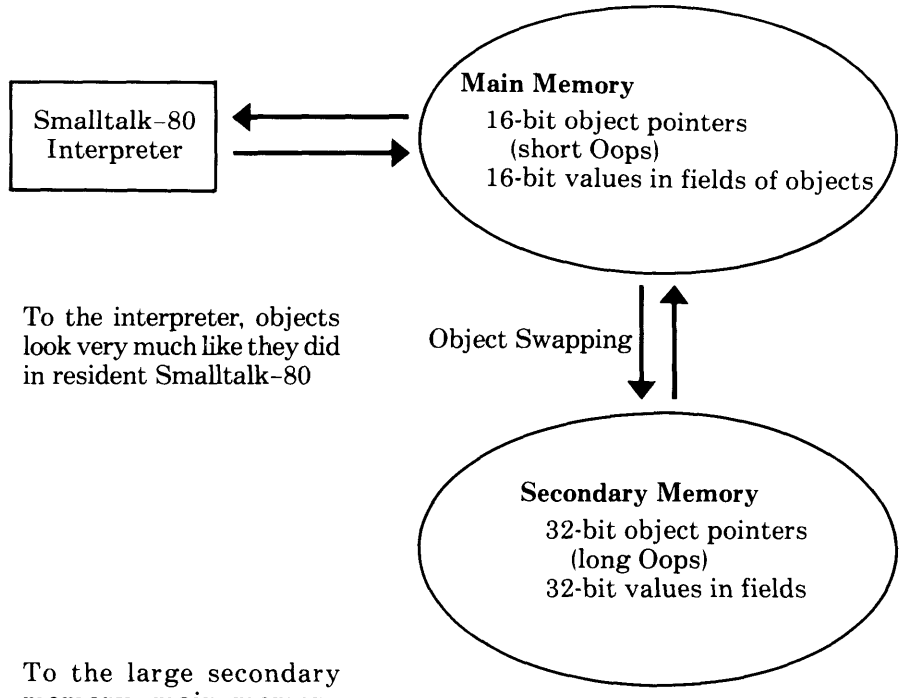


Figure 14.1

pointer as a 32-bit Oop. An object cached in main memory has a 16-bit Oop (a short Oop) and 16-bit fields. As in the resident Smalltalk-80 implementation, main memory has a resident object table (*ROT* or sometimes called an OT), which contains the actual main memory address of each resident object. An object's short Oop is an index into the ROT, so that the object's address can be determined from its Oop with a single addition and memory reference. When an object is brought into main memory from disk, it is assigned a short Oop, and those of its fields that refer to other objects in main memory are assigned the appropriate short Oop. Fields pointing to objects that are not resident are handled specially, the details of which make up the crux of LOOM.

Thus, when all objects in the working set are in main memory, LOOM behaves just like a resident Smalltalk-80 implementation—all objects have short Oops that index the ROT, providing their actual core address. When an object in core must access one of its fields that refers to an object that is not in core, something special must happen. LOOM brings that object into core, assigns it a short Oop, and resumes normal Smalltalk execution. The main memory resident space of 2^{15} objects acts as a cache for up to 2^{31} objects on the disk.

The LOOM Details

The important issues in the LOOM design implementation are:

- The representation of resident objects,
- The representation of objects in secondary memory,
- The translation between representations, and
- The identification of times when the translations must occur.

The Representation of Resident Objects

Resident objects are represented in a manner similar to their representation in a resident Smalltalk-80 system. Each object has as its name in main memory, a short (16-bit) Oop. The Oop indexes the ROT in order to provide the starting address of the object's body, as shown in Fig. 14.2. The ROT entry also has reference-count bits, and a few other bits, described later. The body of each object contains a word for the length of the body, a pointer to the object's class, and the object's fields. Each field is either a pointer to another object or a collection of "bits", in the same manner as resident Smalltalk-80 fields. We will only deal with pointer fields here. Each field (as well as the class pointer) that refers

Format of Objects in Main Memory

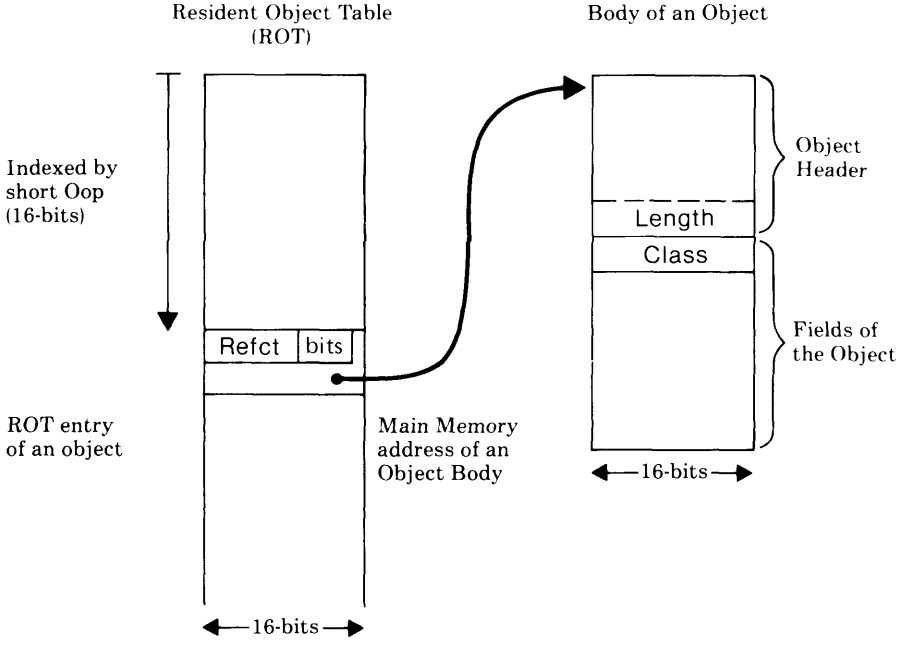


Figure 14.2

to another resident object contains the short Oop of that object. Fields that refer to non-resident objects (objects on secondary storage) contain a short Oop of one of two types, a *leaf* or a *lambda*.

In addition to these fields, resident objects in a LOOM system have three extra words. Two of these words contain the long (32-bit) Oop of that object. The third word, known as the delta word, contains a delta reference count and some other bits. The short Oop of an object is not only an index into the ROT for that object's address, but is also the result of a hash function applied to that object's long Oop. See Fig. 14.3, p. 256. The algorithm for translating an object's short Oop to its long Oop is:

1. Index the ROT with the short Oop to get the body address
2. Load the long Oop from the first two words of the body

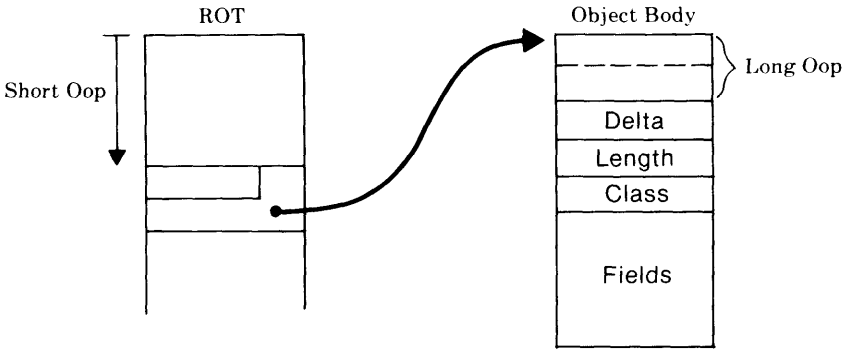
The algorithm for translating an object's long Oop to its short Oop is:

1. Convert the long Oop into a short Oop by applying the hash function
2. Index the ROT with this short Oop to get a body address
3. Look at the first two words of the body
4. If they match the long Oop, then the short Oop is correct
5. If not, create a new short Oop from the current one with a reprobe function (e.g., add 1), and go to step 2

The Representation of Objects in Secondary Memory

Secondary memory is addressed as a linear space of 32-bit words. Objects start with a header word that contains 16 bits of length and some status bits. Each pointer field in the object is 32 bits wide. Non-pointer fields (such as the bytes in Strings) are packed, with 4 bytes in each 32-bit word. Resident Smalltalk-80 SmallIntegers are rather short to be occupying a full word on the disk. However, since they represent legitimate object pointers, their 15 significant bits are stored along with a flag value in a 32-bit pointer field on the disk. The long Oops in pointer fields are 31-bit disk pointers, addressing as many objects as will fit into 2^{31} disk words (32-bit words). Fields of objects on secondary storage always refer to objects in secondary storage and do not change when the object to which they point is currently cached in main memory. As shown in Fig. 14.4, no information about primary memory is ever stored in secondary memory. Information such as an object's short Oop, its location in primary memory, or whether it is currently cached in primary memory are never recorded in secondary memory.

Finding an Object's Long Oop from Its Short Oop



Finding an Object's Short Oop from Its Long Oop

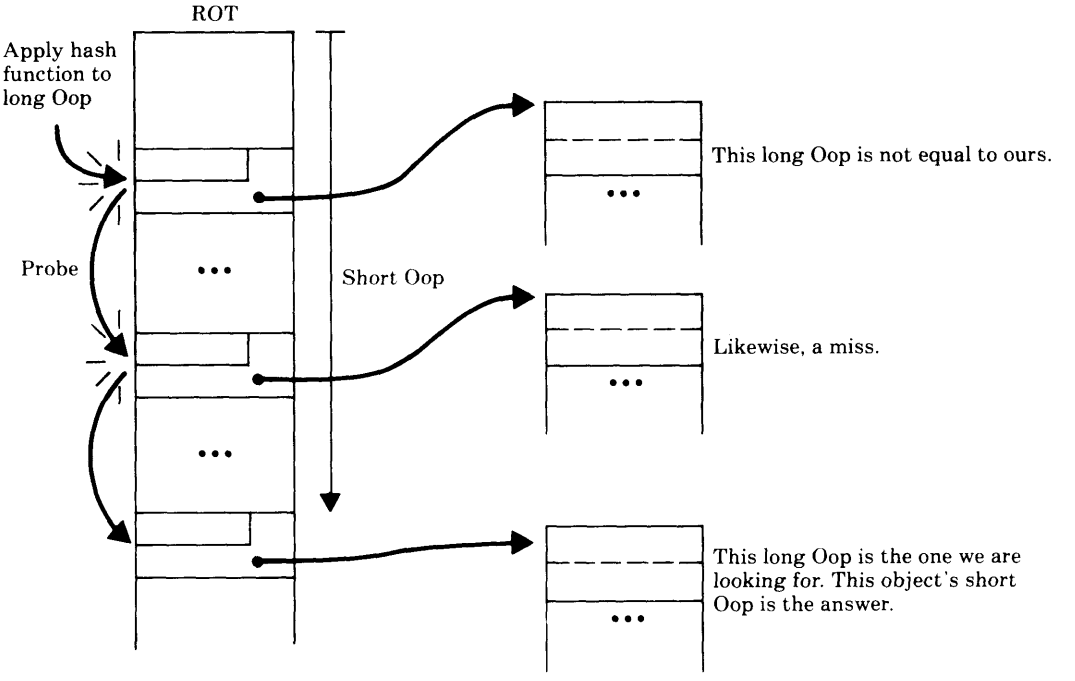


Figure 14.3

How Objects in Primary and Secondary Memory Refer to Other Objects.

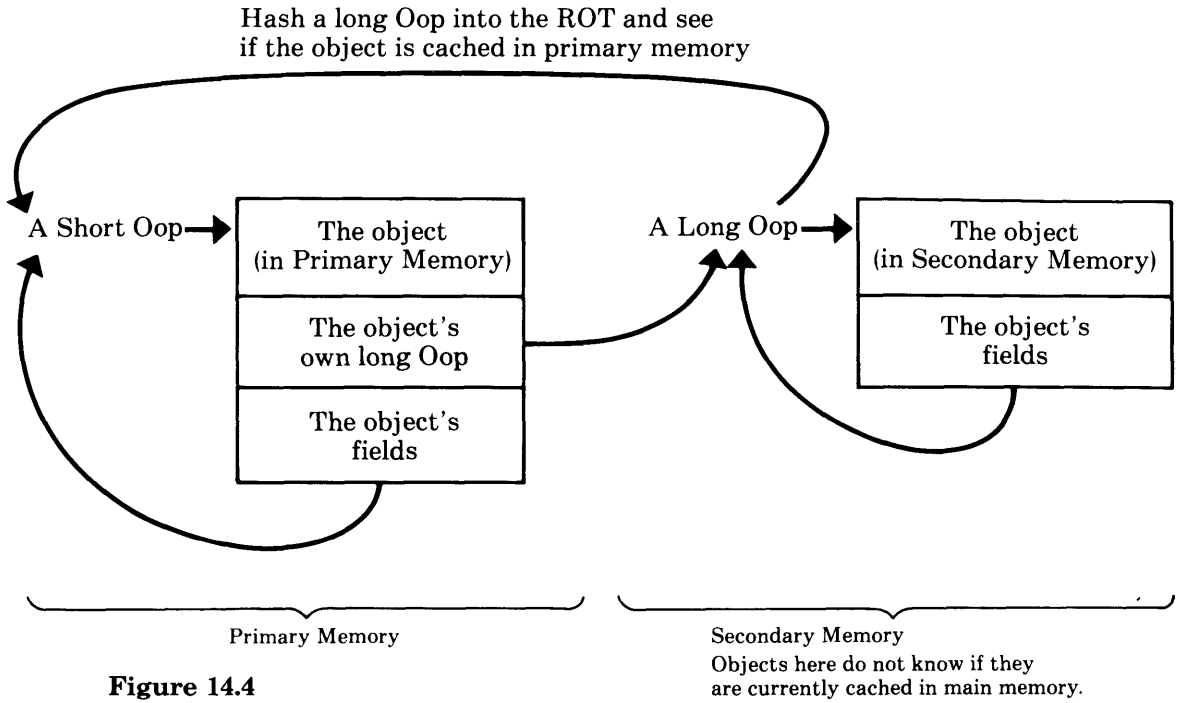


Figure 14.4

When an object on secondary storage is brought into main memory, its fields must be translated from the long form to short form. The object is assigned an appropriate short Oop (one to which its long Oop hashes), a block of memory is reserved for it, and all of its fields are translated from long Oops to short Oops. Those fields that point to objects already in main memory are given the short Oops of those objects; those that point to objects not in main memory are handled in one of two ways, with leaves or with lambdas.

☐ *Leaves* Leaves are pseudo-objects that represent an object on secondary storage. They have a short Oop hashed by that object's long Oop and a ROT entry, but their image in memory only contains a length word, disk address words, and the delta word. Their image contains no class word or fields, as shown in Fig. 14.5. Leaves therefore, only take up 4 words of memory, whereas the average object takes up 13. Leaves are created without looking at that object's image on secondary storage.

This is very important, since a major cost in virtual memories is the number of disk accesses. The short Oop of the leaf may be treated as if it were the short Oop of the object; it may be pushed and popped on the stack, stored into fields of other objects, without ever needing the actual contents of that object. Its reference count can be incremented and decremented (see p. 262).

A Leaf

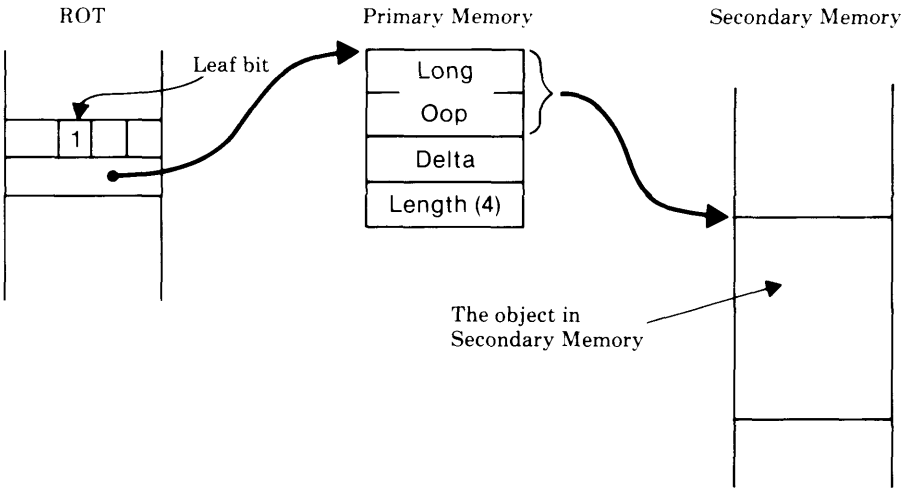


Figure 14.5

An object is always in one of three states. Either the entire object is in main memory, a leaf for the object is in main memory, or the object exists only on the disk. See Fig. 14.6. When the interpreter needs a field from an object which is represented by a leaf, the entire object with its fields must be brought into main memory from disk. Since the leaf contains the disk Oop, the body is easy to find. After the body is translated into main memory form, its core address is stored into the leaf's OT entry, and the leaf body is discarded. Short Oop references to the object remain the same, but now the full object is actually there. Since a leaf can be substituted for an object body and vice versa with no effect on pointers to the object, LOOM is always free to make more room in main memory by turning resident objects into leaves.

☐ *Lambdas* Lambdas are the second way to represent fields of resident objects that refer to objects on secondary storage. Lambda is a place holder for a pointer to an object which has not been assigned a short Oop. Its purpose is to reduce the number of leaves in the system. Lambda is a pseudo-Oop, a reserved short Oop (the Oop 0) which is not the name of any resident object. Consider an object which has a lambda in one of its fields. To discover the actual value of that field, LOOM must go back to the object's image on secondary storage, look in that

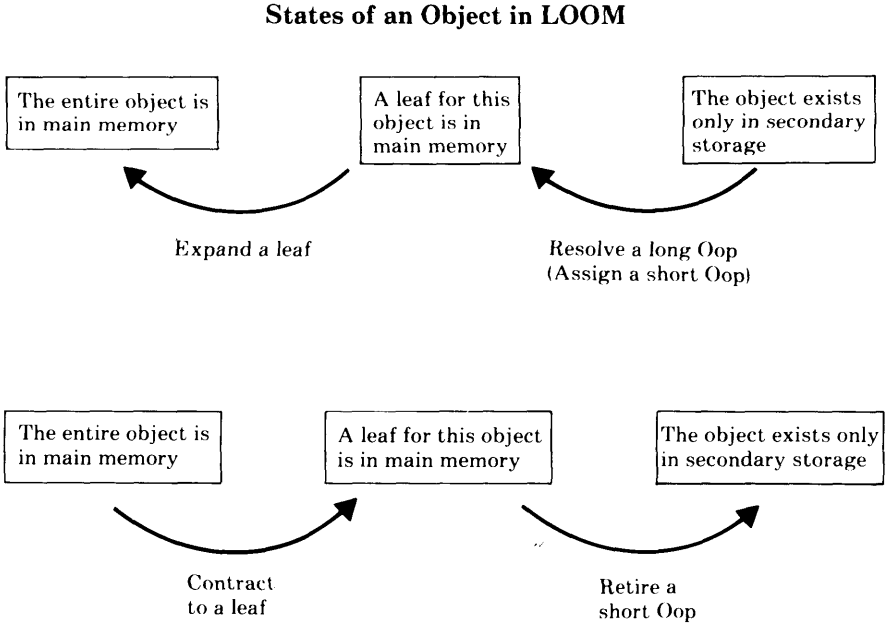


Figure 14.6

field for a long pointer, and create a leaf or resident object. This means that the cost of fetching a lambda field is an extra disk reference. However, unlike leaves, lambdas do not take up ROT entries (they all use the single pseudo-ROT entry at 0) and they do not take up any main memory storage. Since the number of ROT entries is limited to 2^{15}, and main memory is a somewhat scarce resource, this saving can be important. During an object's typical stay in main memory, some of its fields will not be referenced. If leaves are created for the values in those fields when the object is swapped in, and then destroyed again when the object is thrown out, much work is wasted. Putting lambdas into fields which will not be referenced during the object's current stay in primary memory saves both the space and the time needed to create and destroy many leaves.

Determining whether to make the fields of an object be leaves or lambdas when the object is brought into main memory is a tricky business. The choice of strategy strongly affects the performance of a LOOM system. Creating a leaf takes more time and uses up more memory and a ROT entry, but does not cause any extra disk accesses. A lambda will cause an extra disk access if the field it occupies happens to be referenced, but a lambda is faster to create. One way to make the decision between leaf and lambda is to rely on history; if a field was a lambda when this object was written to the disk one time, it is likely to remain a lambda during its next trip into main memory. Each pointer field of the disk contains a hint, the *noLambda* bit, and the object faulting code follows the advice of the hint.

The Translation Between Object Representations

Translating between the memory-resident and secondary-storage representations of an object is straightforward. For those fields that contain short Oops, the Oop refers to an object or a leaf. The corresponding long Oop can be found in the header of the object or leaf. If the field refers to an object which has not yet been assigned a long pointer, a long pointer is assigned to the object and a copy is installed in the field. For those fields that contain lambdas, the field is guaranteed not to be changed from the object's previous disk image. (The object's disk image is read before it is written). If the object being translated still has some short pointers to it (has a positive in-core reference count), then it must be converted to a leaf instead of being deleted completely from core.

When to Translate

We have already mentioned when the translation between representations must occur. When a field of an object being brought into main memory has the noLambda bit set, and that field refers to a non-resident object, then a leaf is created. A leaf is also created when a field of a resident object containing a lambda is accessed. When the interpreter needs to access a field in a leaf, the flow of control in LOOM begins (see Fig. 14.7). The leaf is expanded into a resident object; its fields are translated from long form to short form. This is called an *object fault* (because the similar situation in paging virtual memory systems, trying to access a page that is not resident, is called a *page fault*). The inverse operation, *contracting* an object into a leaf, may be done at any time. The final part of an object's journey into primary memory consists of destroying the leaf and reusing its short Oop and memory space. This can only be done when there are no longer any fields in any resident objects pointing to the leaf.

Lambdas may be resolved into leaves and leaves may be expanded into full objects before they are needed, and this is called a *prefetch*. The complementary operations of contraction and prefetch of objects can both be done in the background. The exact order and mix of objects to prefetch or contract can be adjusted at run-time to optimize the performance of secondary storage (disk head movement or network traffic).

LOOM Implementation Details

In this section, we provide some details of how LOOM may be implemented. In particular we discuss the discovery of object faults, reference-counting, and the assignment of the extra bits in the ROT entry and the delta word.

Object Faults

Object faults occur when the interpreter tries to access a field in a leaf or a field in an object whose value is lambda. By the time the interpreter scrutinizes them, all objects must be full resident objects. How can leaves and lambdas be discovered without greatly slowing the speed of the interpreter?

The Flow of Control in LOOM

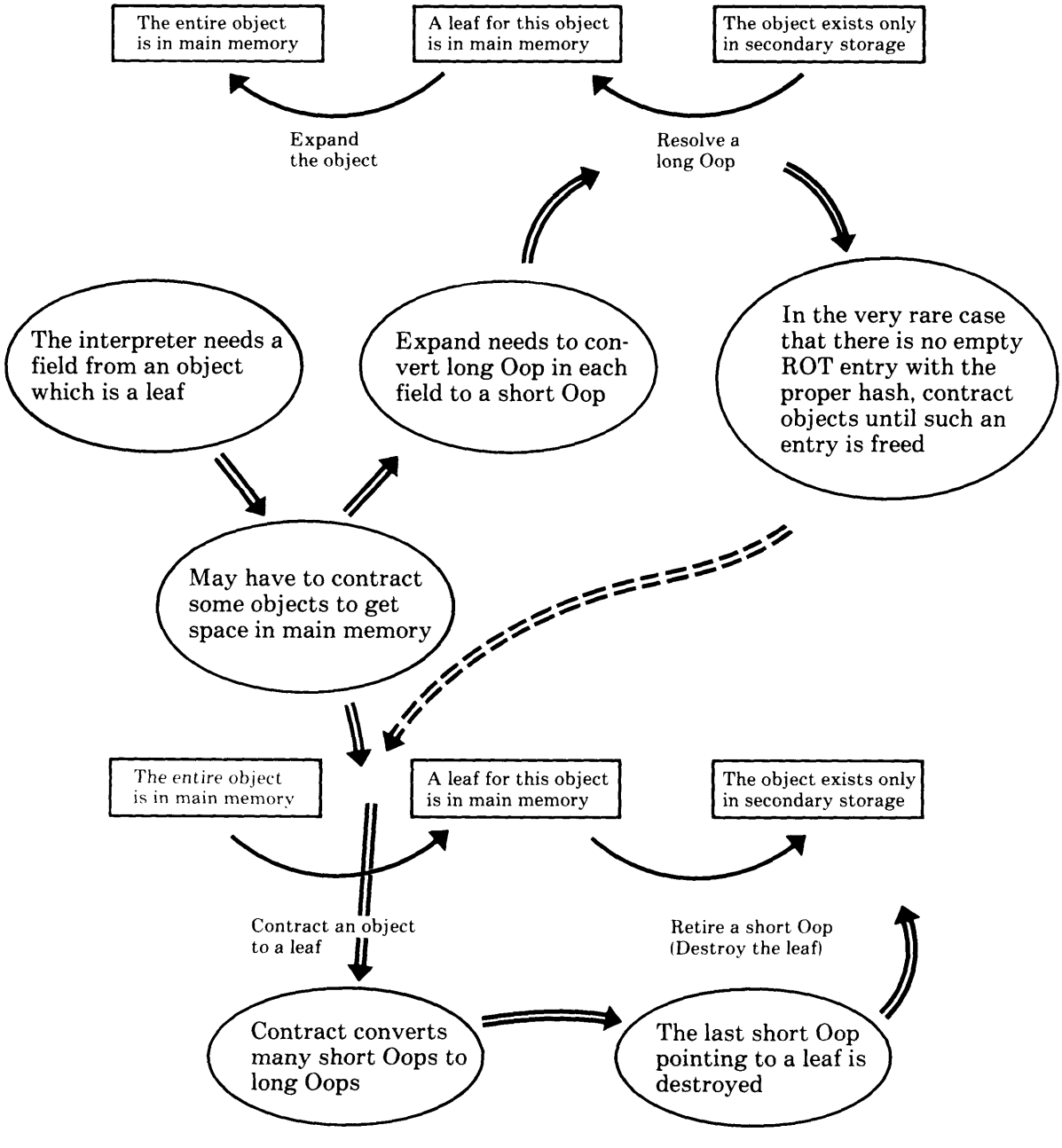


Figure 14.7

It has been our experience that implementations tend to have a single subroutine (or expanded macro) that takes an Oop and sets up some base register to point to the actual address of that object. We call this subroutine "Otmap." It corresponds roughly to the ot:bits: method of the memory manager in the formal specification of the Smalltalk-80 virtual machine, in *Smalltalk-80: The Language and its Implementation*[4]. Otmap is called if and only if you want to fetch or store a field of an object. Note that this is exactly the condition where you must test for the object being a leaf. (Otmap may sometimes be used for other purposes—for example a compaction routine may call Otmap to get the main memory address of the object in order to move it, but it wants to treat leaves and objects the same. These cases tend to be rare, so it is worth having a second subroutine for them.) We reserve one bit of the ROT entry to say whether the entry is for an object or a leaf. The Otmap subroutine tests this bit and calls the LOOM routines when the entry is a leaf. Since both words of the ROT entry are fetched anyway, this extra test usually only costs one or two extra instruction executions.

Testing for lambda however, must be done on *every* field reference. In the worst case, this would mean testing occurs every time a field is fetched from an object and every time an object is pushed onto the stack. To decrease the number of tests, we include one bit in each resident object called "holds lambda." It is set by the LOOM routines whenever that object has a field that is a lambda. The interpreter guarantees that the current context, the home context, the current method, and the receiver all have no lambdas in them. If any of them does contain a lambda, then the LOOM routines are called to make those fields into leaves. In this way, the most common fields fetched and all stack operations can work without testing for lambda. Note that these objects must be cleared of lambdas only when the active context changes. This occurs during message sends, returns, process switches, and during the execution of BlockContext value and value:.

It is useful to note that the LOOM design actually will work with leaves alone, and without lambdas. When the expand routine brings an object into main memory, it turns all the fields into leaves and never creates a lambda. This approach tends to use more short Oops and main memory than the full LOOM design, but could be an intermediate stage in the implementation; providing a working virtual memory system with only the modification to the Otmap subroutine.

Reference Counting

Although some Smalltalk-80 implementations use mark/sweeping garbage collection, most implementations so far, including ours, use reference counting to identify garbage. Therefore we will describe the reference-counting scheme as it applies to LOOM. Reference counting serves two different purposes. One purpose is to detect when the total count of any object goes to zero. The other is to detect when the last short pointer to any object disappears so that the short pointer may be

reused. The resident Smalltalk-80 interpreter keeps reference counts of short pointers. This count is kept in the ROT. LOOM uses the ROT reference count to keep the number of short pointers to an object. In addition, every object on the disk contains a reference count which is the number of long pointers to the object. The total count is the sum of the number of short and long pointers to an object. Whenever a long Oop is converted to a short Oop and installed in a field in main memory, both counts for the object pointed at must change. To avoid a disk access to find and modify the long Oop count every time a field is converted, LOOM keeps a "delta" or running change in the long Oop reference count for each object in main memory. The true long pointer reference count of any object is the count found on the disk in the object's header plus the count found in the "delta" part of the object's delta word in main memory. Fig. 14.8 shows the ROT entry, object body, and disk image of an object. The object has three short Oops pointing at it. It used to have pointers from 6 long Oops, but two were destroyed recently (they were probably converted to short Oops). The total number of references to the object is seven.

There are three sources of reference-count changes. One pointer can be stored over another, a long pointer can be converted to a short pointer, and a short pointer can be converted back. Since the interpreter only deals with short Oops, every store consists of a short pointer replacing another short pointer. This high-bandwidth operation touches only the short pointer reference counts, so the existing code in the interpreter does not need modification. When a leaf expands to a normal object, pointers in its fields change from long Oops to short ones. The expand-a-leaf routine increments the short count of that object and decrements the delta of its long count. The inverse happens when the routine which shrinks objects into leaves converts short Oops to long ones.

Consider the case when the short Oop count of an object goes to zero. The reference-count routine then looks at the object's long Oop count to see if the total count of the object is zero. If it is zero, the object is truly free, and its storage can be recycled. If not, the object is still held by some long pointers. When the short Oop reference count goes to zero, and the delta reference count is zero, then the object's long Oop count on disk need not change. Thus if the ultimate long pointer count of a leaf can be guessed correctly when the leaf is created, the disk count and delta count can be adjusted so that the leaf disappears from main memory without further disk references.

Other Data LOOM Holds for Each Object

As a help to the LOOM system, two other bits are added to the ROT entry for any object—"clean" and "unTouched." Clean is cleared whenever a field of the object is changed; unTouched is cleared whenever a field of the object is read or changed. Clean tells the LOOM system that it need not rewrite the object's image on disk (unless of course, its true reference count changed). Clean is set when the object is newly created

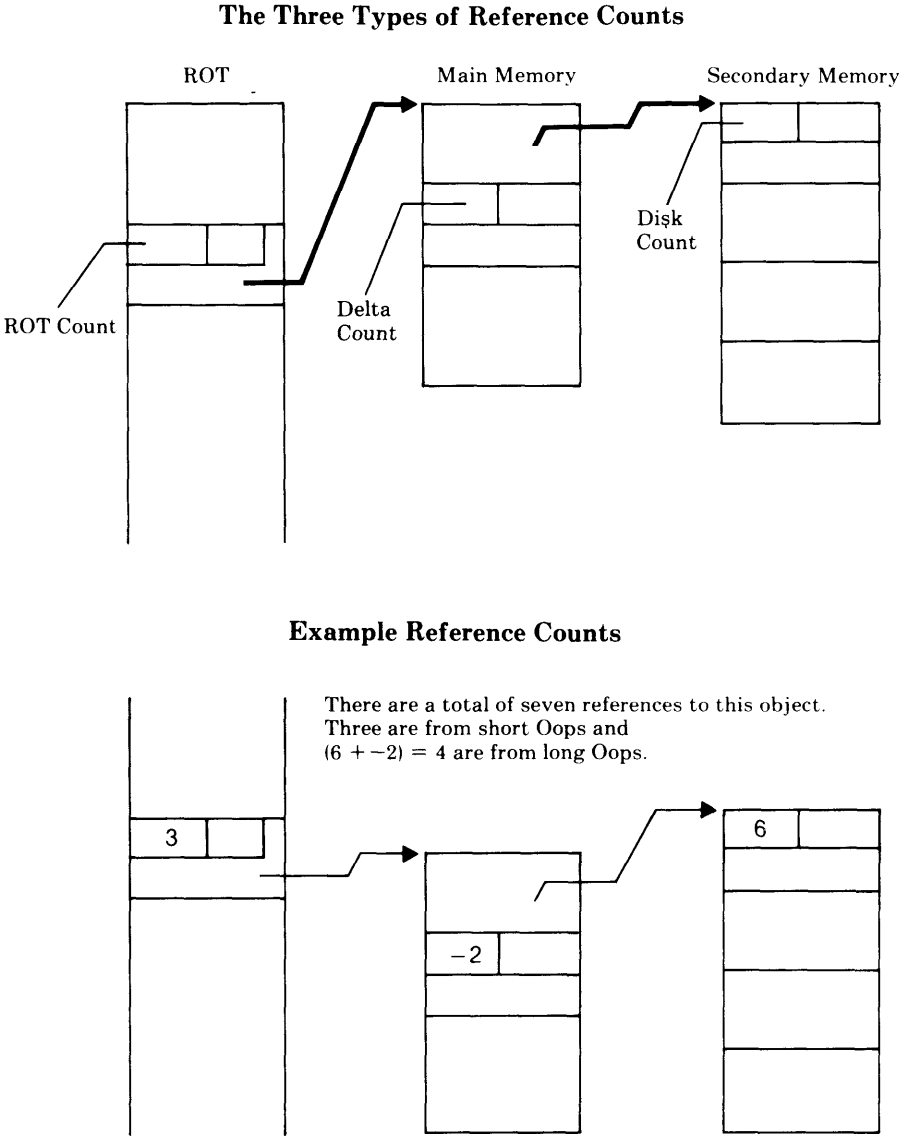


Figure 14.8

or swapped in. UnTouched is set by a routine that sweeps core whenever space is needed. Any object that the routine finds with unTouched still set has not been touched in an entire pass through memory, and is thus a candidate for being *contracted* (turned into a leaf).

The activity which is most likely to cause LOOM to thrash is the resolution of lambdas. When a lambda needs to be resolved (turned into a leaf or discovered to be an existing short Oop), LOOM must first look at the disk image of the parent object. If the pattern of computation is

such that the noLambda hint does not correctly predict which fields are needed by the interpreter, lambdas would have to be resolved often. Even so, lambda resolution is likely to happen soon after the parent was expanded, so keeping the most recently fetched disk pages in a cache relieves the need to go to the disk. When a lambda needs to be resolved, the LOOM procedure looks first in the cache of pages that is called the *disk buffer.* If it finds the object in the buffer, it can directly retrieve the long Oop for the lambda, saving one disk access.

LOOM Implemented in the Smalltalk-80 Language

The LOOM design, though based on only a couple of simple principles, has a number of reasonably complex algorithms that require a substantial amount of code. We were faced with the problem of whether to implement LOOM's object swapping algorithms in a low-level language or a high-level language. Low-level implementations typically provide better performance at the cost of some flexibility.

We opted to implement the LOOM system in our favorite high-level system, the Smalltalk-80 system. A number of factors influenced this choice. The overriding factor was that for us, the Smalltalk-80 language was the most natural way to express and understand complex algorithms. We are implementing LOOM on the Xerox Dorado computer[5] (see also Chapter 7). We believe that the Dorado has sufficient performance and memory space so that the LOOM system will not be called very often. When LOOM is called, it will run with acceptable performance. Also, once the system is up and running, we will have a complete, debugged high-level description of the algorithms. Should we decide to reimplement LOOM on the Dorado or another machine in a lower-level language, only a translation of the code would be required. In addition, we designed LOOM not only as a working virtual memory system for our Smalltalk-80 work, but also as a test-bed for virtual memory techniques. Jim Stamos' master's thesis[6] is an example of one experimental technique based on simulation. We want further studies to use a real virtual memory system.

Deciding to implement LOOM in the Smalltalk-80 language itself led to problems that might not be encountered in a low-level language implementation. In particular, the amount of "machine state" that needs to be saved when switching between running the Smalltalk-80 interpreter for "user" and for LOOM was quite large. The amount is much larger than the amount of Smalltalk-80 virtual machine state that would have to be saved to run the LOOM code written in machine language. Also, to avoid a fault on the faulting code, all of the code and other objects which comprise the implementation of LOOM must be guaranteed to stay in main memory at all times.

We handled the first problem, saving state, by reworking our interpreter. It now obeys the convention that within the execution of a bytecode, an object fault is possible only before any "destructive" operations occur. In other words, before the interpreter writes into a field of any object or changes the reference count of any object, it reads fields from all objects needed by the current bytecode. In this way, the state we needed to save was only the "permanent" state that exists between bytecodes. Temporary state within a bytecode is not saved. In our system then, if an object fault occurs, we back up the Smalltalk program counter, switch the interpreter to the LOOM system, handle the fault, and then restart the bytecode.

The second problem, insuring that no object faults occur during the execution of the LOOM algorithms themselves, went through a couple of different designs. The first method we tried was to have the LOOM objects and the user's objects in the same Smalltalk-80 space, but to mark all the objects LOOM would ever need "unpurgable", and to guarantee that free space never went below a certain level. We made an almost-complete implementation of LOOM using this method on the Xerox Alto computer[7] before moving onto the Dorado. The problem with LOOM and the user sharing the same Smalltalk is retaining the marks on objects that LOOM needs. If the user adds many methods to class SmallInteger and its method dictionary grows, how does the new array in the dictionary get marked "unpurgable"? There are many similar cases.

The LOOM implementation on the Dorado has two separate Smalltalk-80 systems in the same machine: a full-size system for user's programs, and a smaller one for LOOM. The LOOM system has some primitives that enable it to manipulate the bits inside of objects in the user system. (Note that because they use the same interpreter, the user system has these primitives also. However, they make no sense in the user system, so are never used.) Because the LOOM system uses only a small subset of the Smalltalk-80 system, it can be much smaller, and can be guaranteed to fit entirely within its portion of main memory and never cause an object fault. Fig. 14.9 provides a view of the communication between the systems.

Alternative Smalltalk Virtual Memory Designs

The LOOM virtual memory design is only one of many ways to implement a virtual memory for a Smalltalk-80 system. The advantages of the LOOM design are:

1. It runs as fast as a resident Smalltalk-80 interpreter when the working set is in core,

Two Separate Smalltalks in the Same Machine

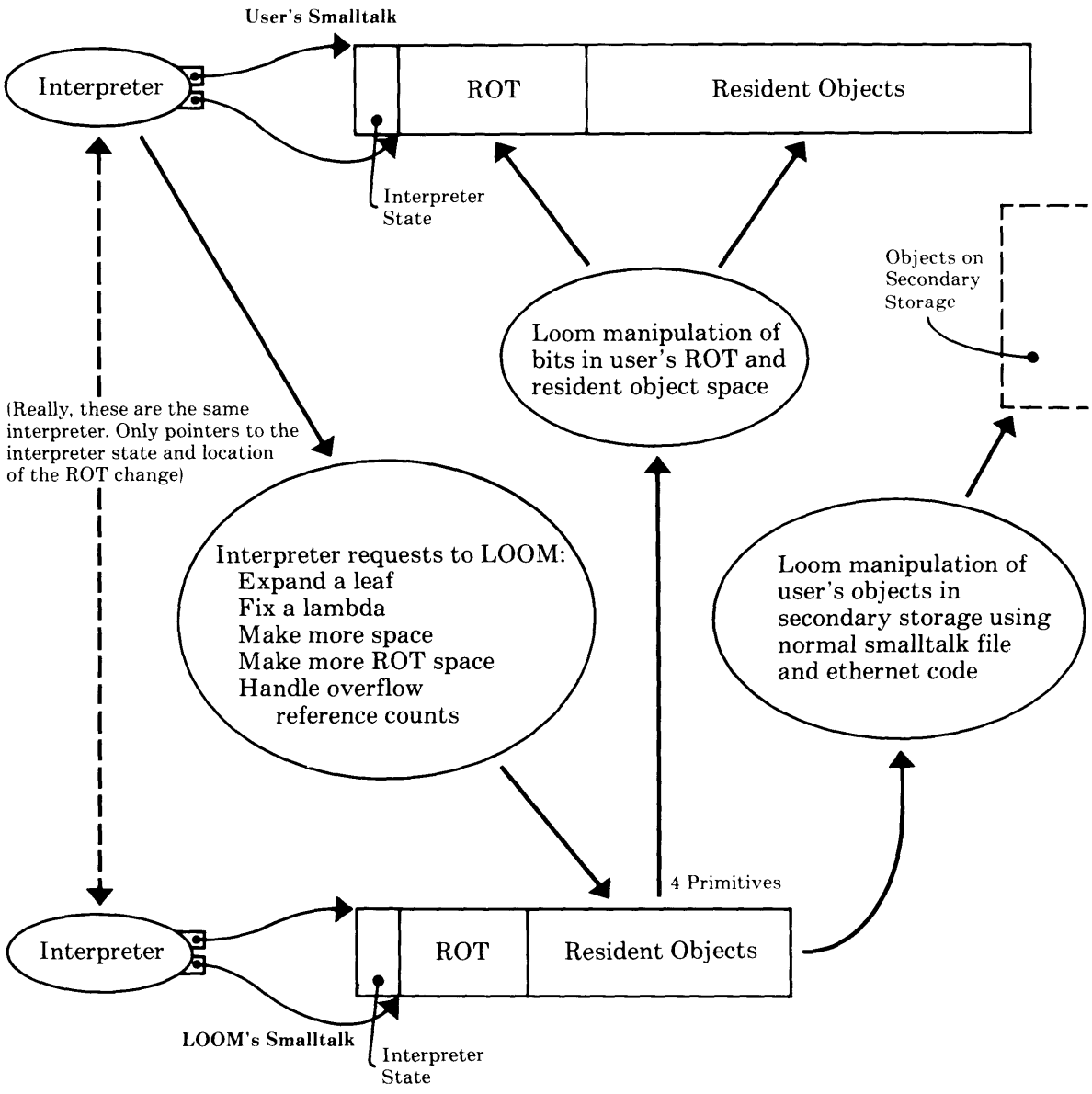


Figure 14.9

2. It uses 16-bit fields in core to conserve space,
3. It allows the interpreter to avoid handling 32-bit Oops, which makes the interpreter smaller and faster on 16-bit machines,
4. It only uses memory for objects that are actually referenced, and
5. It provides a large, 32-bit virtual address space.

Its major disadvantages are:

1. It relies on fairly complicated algorithms to translate between the address spaces,
2. It takes no advantage of current hardware technology for memory fault detection, and
3. It must move objects between disk buffers and their place in memory.

There are alternatives to many of the design decisions within LOOM and to using the LOOM design itself.

LOOM was designed specifically to experiment with various methods of "grouping" objects on disk pages. If objects which are likely to be faulted on at the same time live on the same disk page, only the first fault actually has to wait for the disk. Static grouping restructures the arrangement of objects on disk pages while the system is quiescent. It reduces the number of disk accesses for both paged virtual memories and object swapping systems. Stamos extensively studied the advantages of static grouping and compared LOOM to paged virtual memories[8]. LOOM is also designed for experiments in dynamic grouping. We have several algorithms in mind for moving objects on the disk while Smalltalk is running. These algorithms will endeavor to reduce faulting by dynamically placing related objects on the same disk page.

We also mentioned that a LOOM system can be built that only uses leaves and not lambdas. Another alternative that we did not pursue is to use a marking garbage collection scheme for resident objects and reference counting for disk references. This should be possible using the delta reference-count scheme.

LOOM is currently intended for use over a local area network. The design could be extended to bring many users, many machines, and large quantities of immutable data into the same large address space. If 32-bit long Oops are not big enough, objects in secondary memory could be quad-word aligned, giving 2^{36} bytes of address space. The LOOM algorithms are parameterized for the width of long pointers, so that a change to 48-bit wide long Oops would not be difficult to do.

The LOOM design may be used for non-Smalltalk systems. In particular, we have proposed a LOOM-like design to extend the address space of Interlisp-D. The design adds another level of virtual memory to the

existing Interlisp-D paging system by treating a page as a single object and an existing page address as a short pointer.

Learning from LOOM

Our LOOM virtual memory system is in its infancy. We are only beginning to make measurements on its performance. The design choices of the LOOM system are based on the belief that the way to design good virtual memory systems is to determine what happens most of the time, make it go fast, and hope it continues to happen most of the time. Many trade-offs were made to meet this goal. Some of the design choices we made apply to almost all Smalltalk-80 implementations and some were determined by our hardware/software environment. For example, the general idea that object swapping saves main memory over paging applies to all Smalltalk-80 systems, but the relative cost of object swapping versus paging can be heavily influenced by hardware support for one or the other. Since we know of no current hardware that supports object swapping, but we do know that a great deal of current hardware supports paging, paging has a tremendous advantage. Many of the costs of paging are hidden, such as the address computation on every memory reference, and the "built in" paging hardware on many machines. If those costs were brought into the open, and the same amount were spent on assisting object references, object oriented virtual memories might have better cost-performance than paging systems.

The LOOM design uses two levels of object addressing and translates between address spaces when necessary. Up to 2^{31} objects residing on secondary storage are represented by a cache of 2^{15} objects in main memory. These behave almost identically to resident Smalltalk-80 objects. When a reference from an object in main memory to one in secondary memory is made, an object fault occurs, the latter is brought into main memory, and processing continues. This design allows for a large virtual address space and a space- and speed-efficient resident space. Because the major algorithms in LOOM are written in Smalltalk itself, LOOM will be a major test-bed for new swapping algorithms and for new ways of reducing page faults by grouping objects in secondary storage.

Acknowledgments

The design of LOOM was a true group effort. Jim Althoff and Steve Weyer proposed an early version to improve the speed of their work on programmer directed object overlays. Peter Deutsch worked out a design for an early version of the dual name spaces (short and long Oops).

Dan Ingalls, Glenn, and Ted designed the three kinds of reference counts. Danny Bobrow said that leaves were not enough, and Larry Tesler suggested lambdas from the design of his operating system called Caravan. Ted, Dan, and Glenn worked out the final system design, and Ted and Diana Merry built a test version of the LOOM algorithms. Ted and Glenn did the Alto and Dorado implementations.

References

1. Denning, Peter J., "Virtual Memory", *Computing Surveys* vol. 2, no. 3, Sept. 1970.

2. Burton, Richard R., et al., (The Interlisp-D Group), Papers on Interlisp-D, Xerox PARC CIS-5, July 1981; (a revised version of Xerox PARC SSL-80-4).

3. Kaehler, Ted, "Virtual Memory for an Object-Oriented Language", *Byte* vol. 6, no. 8, Aug. 1981.

4. Goldberg, Adele, and Robson, David, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

5. Lampson, Butler W., and Pier, Kenneth A., "A Processor for a High-Peformance Personal Computer", Seventh International Symposium on Computer Architecture, SigArch/IEEE, La Baule, France, May 1980; (also in Xerox PARC CSL-81-1, Jan. 1981.)

6. Stamos, James W., "A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance," Xerox PARC SCG-82-2, May 1982.

7. Thacker, C. P., et al., "Alto: A Personal Computer", in *Computer Structures: Readings and Examples*, 2nd Edition, Eds. Sieworek, Bell, and Newell, McGraw-Hill, New York, 1981; (also Xerox PARC CSL-79-11, Aug. 1979.

8. See reference 6.