A Third Generation Smalltalk-80™ Implementation

Patrick J. Caudill

Allen Wirfs-Brock

Computer Research Laboratory
Tektronix Laboratories
P.O. Box 500 MS 50-662
Beaverton, Oregon 97077

Abstract

A new, high performance Smalltalk-80™ implementation is described which builds directly upon two previous implementation efforts. This implementation supports a large object space while retaining compatibility with previous Smalltalk-80™ images. The implementation utilizes a interpreter which incorporates a generation based garbage collector and which does not have an object table. This paper describes the design decisions which lead to this implementation and reports preliminary performance results.

Introduction

Tektronix Large Object Space Smalltalk (4406 Smalltalk) is a high performance implementation of the Smalltalk-80™ System for members of the Tektronix 4400 series of workstations using the Motorola 68020 processor. The major goal of this implementation was the ability to support a very large number of active objects.

A previous Smalltalk implementation at Tektronix had demonstrated the feasibility of using a microprocessor to host high-performance Smalltalk implementations. That implementation, following the Smalltalk-80™ virtual machine specification [GoR83], was restricted to an object space of approximately 32,000 objects. As applications developers gained experience with this system the limited object space was found to be the major restriction of the system. Because of this limitation, application developers began to think of Smalltalk as a system which was excellent for application prototyping but which was not suitable for large scale applications development. 4406 Smalltalk is intended to be usable for the development and delivery of large applications systems.

A Third Generation Design

4406 Smalltalk represents the third major Smalltalk interpreter implementation within Tektronix Laboratories. As such, it benefited from experience gained from our earlier implementations as well as the experience of other Smalltalk developers.

The first Tektronix Smalltalk-80™ implementation [Mcc83] was a direct implementation of the Goldberg and Robson Virtual Machine specification. It strictly adhered to the data representations and algorithms from the specification and was implemented in Pascal for the Motorola 68000. The dismal performance of this implementation[Mcc83a] lead us to the conclusion that Smalltalk-80™ interpreters needed to be more carefully designed to match the capabilities of the host computer.

Smalltalk-80 is a trademark of Xerox Corporation.

Our second major implementation [Wir83, Wir85] was for the Tektronix Magnolia, an experimental 68000 based workstation. For this implementation, data structures and representations (e.g. object representations in memory and object table layouts) were carefully chosen to match to the capabilities of the Motorola 68000. The interpreter itself was carefully hand-coded in assembly language in order to minimize the number of instructions to implement each bytecode. Most importantly, new storage management and garbage collection techniques were developed. This implementation was an order of magnitude faster than the first interpreter and ultimately evolved into Tektronix 4404 Smalltalk, the first high performance Smalltalk implementation for a low cost workstation.

In addition to our own implementations, we were most strongly influenced by the implementation efforts of David Ungar at UC Berkeley [Ung85]. Ungar's BS II, although exhibiting only modest performance, demonstrated the feasiblity of implementing Smalltalk without an object table in conjunction with a generation based garbage collection scheme.

4406 Smalltalk Design Goals

Four major goals drove the design of the 4406 Smalltalk interpreter. These were:

1) Provide support for a large number of objects.
2) Increase performance relative to the existing 4404 interpreter.
3) Remove virtual machine design irregularities caused by the limited object space.
4) Preserve the basic semantics and functionality of the Smalltalk-80™ virtual machine.

The 32,000 object limit of the previous interpreters had become our major impediment to the implementation of serious applications using Smalltalk. Any small multiple increase of the object space size was considered unacceptable since it was assumed that applications would soon exceed any such new limit. The size of the object space should be limited only by the available processor address space. The interpreter should efficiently support widely varying object space sizes. While early uses of the system would be with relatively small applications designed for a restricted object space, the expectation was that new applications would ultimately use hundreds of thousands to millions of objects.

The larger object space needed to be achieved without reducing the speed of Smalltalk program execution. The major goal of the previous interpreter design had been the achievement of execution speeds adequate for serious development work. This goal was achieved but even higher performance levels were needed to support the larger applications being considered. An important factor contributing to the high performance of the previous interpreter was its mixed strategy approach to garbage collection. The collection strategy incorporated special allocation zones and representations, deferred reference counting, and mark/sweep collection. Unfortunately, the interactions between the various techniques were extremely difficult to understand and debug. A secondary goal of the new design was a simpler garbage collection strategy which did not sacrifice performance.

The standard Smalltalk-80™ virtual machine design incorporates several irregularities which were apparently introduced to reduce the total number of objects required by the standard system. For example, instances of class CompiledMethod contain both object references (the literals) and binary data (the bytecodes) even though the object memory architecture expects only objects which uniformly contain only object references or only binary data. The unusual representation was presumably choosen to eliminate the need for separate literal and bytecode objects for each method. The irregular CompiledMethod representation adds complexity to the garbage collector (increasing garbage collection overhead) and requires special primitive methods within the interpreter to support the representation. In addition, it is impossible to make subclasses of CompiledMethod. A new, large object space interpreter design should be able to eliminate such irregularities and hence provide increased performance and functionality.

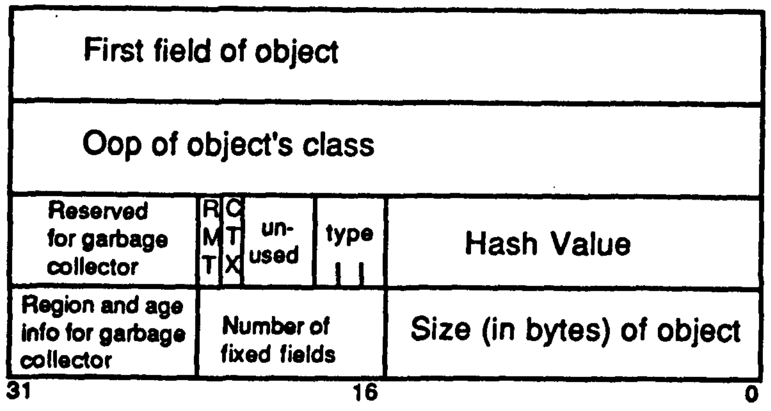
Any Smalltalk-80™ implementation must support a complex, pre-existing system environment (i.e. the Smalltalk-80™ programming environment). Such support is complicated by system code with dependencies upon virtual machine implementation details. For example, the existence of an object table is implicitly assumed by any code which uses the become: primitive with impunity or which depends upon some ordering of object names. A number of the programming aids assume that enumerating all instances of some class or all objects in the virtual image is a computationally tractable operation. While this assumption may be true for a virtual image limited to 32,000 objects it is probably not true for significantly larger object spaces. A large object space interpreter needs to continue to provide support for such operations in order to support the existing system code. Such assumptions need to be expurgated from the Smalltalk code as the system evolves towards much larger virtual images using the new interpreter.

Design Overview

4406 Smalltalk uses a pure bytecode interpreter, implemented in assembly language for the Motorola 68020 processor. We chose an interpreter over the dynamic translation techniques of Deutsch and Shiffman [DeS74] because we felt that the performance advantages exhibited by dynamic translation were insufficient to justify the increased complexity of the implementation.

Object references (Oops) are 32-bit values which incorporate a 1-bit tag field to distinguish SmallIntegers from object pointers. Object pointers directly encode the current address of the target object. An object table is not used. The size of indexable objects is limited only by the available address space.

A generation scavenging derived scheme is used for garbage collection. This differs from our previous implementations which used a deferred reference counting collector [DeB76]. As in our previous implementation, a volatile context stack is used to limit the creation of context objects.


First field of object
Oop of object's class
Reserved for garbage collector
RMT
CTX
un-used
type
Hash Value
Region and age info for garbage collector
Number of fixed fields
Size (in bytes) of object
31
16
0

CTX – 1 if this is a context object

RMT – 1 if this object has a remote indexable part

Type - 0 normal (non-indexable)
1 byte indexable
2 word (16-bit) indexable
3 long (32-bit) indexable
4 pointer indexable

Figure 1 Object Header Format

CompiledMethods are represented as a composite of separate binary (bytecode) and pointer (literal) objects. This makes CompiledMethods transparent to the garbage collector and permits their subclassing.

Object Memory Architecture

Memory is composed of 32-bit words. Each word containing an object pointer has its most significant bit set and the rest of the word contains the byte address of the object. On the 4406, virtual memory hardware can be set to ignore bits in the upper portion of the address word. A small integer has its most significant bit cleared. The rest of the word contains a value in the range 1,073,741,823 to -1,073,741,824 in 2's complement notation. The range of small integers covers the time critical cases and no large integer primitives were implemented.

Since there is no object table, the header of objects has been expanded to a three word header [Figure 1] which includes information that had been previously stored in the object table. This header contains a 16-bit size field which gives the size of the object in bytes. The actual storage size of the object is this size rounded up to a multiple of four bytes. This allows the size field to indicate directly the size of byte arrays rather than using separate modifiers. There are two bytes reserved to hold a copy of the information from the class's instance specification field. One of these bytes contains the number of fixed fields. The maximum number of fixed fields has been reduced to 256, which is not considered a significant limitation since there are no byte codes to address even that many. The object type code has been expanded to include the categories byte, word, quad and pointer, but there is currently no primitive support for quad-byte data objects. There is a bit to show if this object is a context. With this information it is unnecessary to access an object's class object, which might cause a page fault, to access the indexable fields of that object.

In the standard Smalltalk-80™ system the primitive message asOop returns an integer value which represents the object's position within the object table. This value is then commonly used for hashing. In 4406 Smalltalk a sixteen bit field within the object header is filled with a hash code, which is the value returned by the primitive message asOop. This is necessary since the garbage collector may move objects changing their object pointer value. Since this is not enough bits to provide a unique value, the message asObject is not implemented. There are also sixteen bits reserved for use by the garbage collector. These maintain, among other things, the approximate age of the object in the system.

Indexable objects may be broken into two parts, a *base* part and a *remote* part which contains the indexable fields [Figure 2]. All object pointers address the base part which contains a pointer to the remote section if one exists.[Kra84] With only indexable fields stored in these remote sections of the object, only array accessing primitives, such as at: and at:put:, must be aware of them. Any size object with an indexable field may have a remote part but we generally only create an object with a remote part if it is larger than a minimum size. The remote indexable part makes growing an indexable object much easier as the remote part may be reallocated and only the one pointer must be changed, rather than all pointers to the object. The remote part has its own header which has a 32-bit size field allowing objects as big as memory. This overcame the 64K byte limitation imposed by the 16-bit size field in an object. The remote header also contains a back pointer to the base object memory and a field for use by the garbage collector. The remote parts are allocated from a separate heap memory managed with a free list.

In our first Smalltalk interpreter we noticed that the reference counting system used an excessive amount of processor time. This was especially true of short lived objects referenced from a context stack. We

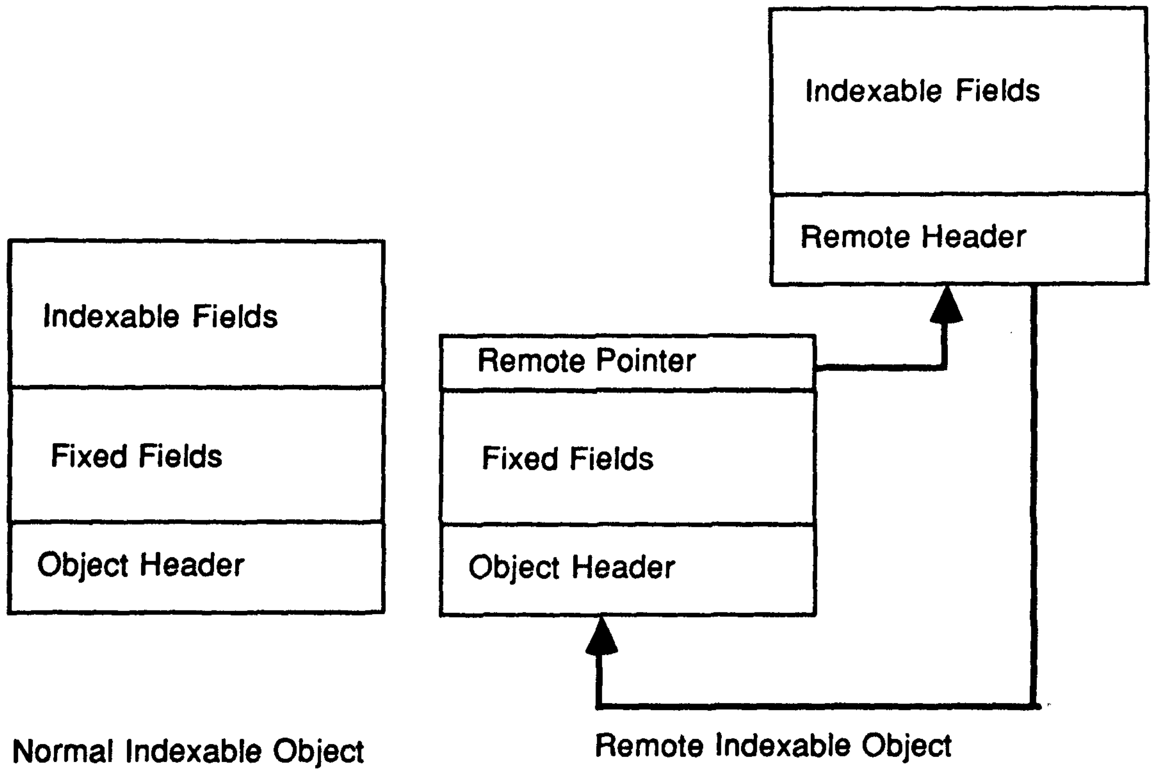


Figure 2 Indexable Object Organization

overcome this problem in our second implementation by only reference counting these objects when memory ran out, at which time objects with zero counts were freed. Reference counting also has problems with circular garbage which require an additional mark-sweep subsystem to periodicly clean memory. In a virtual memory system the free list caused a very fragmented memory with little locality of reference, and had a tendency to thrash. With no object table to hold the reference counts yet another layer of complexity would have been added. Instead a stop-and-copy mechanism was used following the leads of Lieberman and Hewitt[LiH83], Ungar[Ung84] and Ballard[Bal82]. Our scheme breaks memory down into several regions each of which is a pair of Baker half-spaces. All new objects are appended to the end of the active memory in one region. When the active half-space for a region is full, all live objects are copied to the other half-space. After a fixed number of copies within a region, an object is reallocated to the next older region. This has the advantages of quick allocation, short lived objects are never seen by the collector, and a high locality of reference for new objects.

When a region fills, the garbage collector copies all objects known by the interpreter from it. Next, objects referred to by active contexts are copied, then objects referred to by objects in other regions. Finally objects referred to by just-copied objects are copied. In each case copying is usually to the other half-space but could be to the next older region. A table of older pointers into this region is kept to determine active objects within the region. We do this to keep from having to scan the older regions in their entirity. When this table is scanned, the referenced object is copied and the pointer is updated at the same time.

The system currently has seven regions. This appears to be enough to prevent premature tenuring of objects, that is placing intermediate lived objects into permanent storage. But it is few enough that objects are not copied too many times before they move into a region which is less actively collected. This area of the interpreter may be fine tuned as we become more familiar with the performance of the system.

Contexts are both very transient and contain references to ephemeral objects, therefore they account for a large amount of the activity of new objects. A lesson we learned in the second generation system was that by handling contexts differently we could achieve great gains in performance. In both that system and this one, contexts are allocated from a push down stack. Only if a pointer to a context is created is the context converted to a real object. In both systems references from context stacks are handled specially. In this system when a context object is activated a note is made in a special table. Stack references are not checked to see if they refer to a younger object. When a garbage collection is about to be made, all such activated contexts are scanned and the appropriate references are put in the tables. This saves the work of checking and marking references which are to be immediately destroyed.

The garbage collector provides another reason for having remote parts for large objects. When a region is salvaged the remote part does not have to be moved. Since larger objects exhibit a tendency to have a longer life this can amount to a large saving. Also, a very large object could fill or overfill a region, causing many other objects to be prematurely aged into an older region. If only the base part uses space from the region then this effect will be minimized.

Primitives

The primitive become: had to have major changes to operate. In our previous implementations, become: swapped the addresses of the data for two objects in the object table, exchanging the meaning of their object pointers. This was a relatively cheap operation. Now, with no object table, the operation is much more difficult. We rely on a series of stratagems to get reasonable performance out of the primitive. If the two objects are the same size their memory representations may be interchanged. A large percentage of the use of become: is to grow the indexable portion of an object. We can do this by swapping the base parts of the objects and forcing the indexable parts to be remote. If this creates a small remote part, the garbage collector will merge the remote part back into the base the next time the object is copied. In the worst case, such as adding an instance variable to an object, all memory must be scanned and pointers to the two objects exchanged. This can be speeded up by using the reference information acquired by the garbage collector to change pointers in older regions. In this last case the hash codes must also be exchanged as it is a function of the object pointer and not the object.

The primitives relating to the number of objects and amount of core left were difficult to define. Since there is no object table there is no real limit on the number of objects. coreLeft is also difficult because of the way in which memory is allocated and because there may be dead objects still occupying memory. We finally implemented a new primitive which returns the memory used and the total number of objects in the system. This is done by scanning all memory counting objects and is a slow operation. oopsLeft is then answered as an estimate from the average size of the objects and the amount of free memory. This estimate is inaccurate and also costly. The oopsLimit and coreLimit were left unimplemented. The system will die when it runs out of 32M of virtual memory but by that time the trashing behavior should be really obnoxious.

A different problem was encountered with someInstance and nextInstance. In previous versions these were used to scan the object table to find all instances of a class. Hence they implicitly assumed some permanent ordering of objects. No such ordering exists since the garbage collector may move and reorder objects as part of its operation. A scan of memory would not have given the same result if the garbage collector became active between calls. In the standard image nextInstance was only used in the method for allInstances. So we implemented this method as a primitive which returns an array of the pointers to all objects within a given class. The other two messages can be implemented using this primitive. Since objects are not immediately collected when they die, allInstances may find some of these zombie objects, unused but uncollected, and bring them back to life.

There are no primitives implemented for large integers. The new range of small integers more than covers the cases for which large integer primitives were previously used. Since small integers are larger than the address range of the machine, they may be used for all indexes in array and string handling.

Our hardware includes a sophisticated floating point processor. We extended the primitives for floating point to access some of the functions this processor provides. The new primitives perform trig and log functions at a high rate of speed. The hardware also understands small integers and the floating point primitives will take small integer arguments, greatly increasing the speed of expressions like (aFloat + 1).

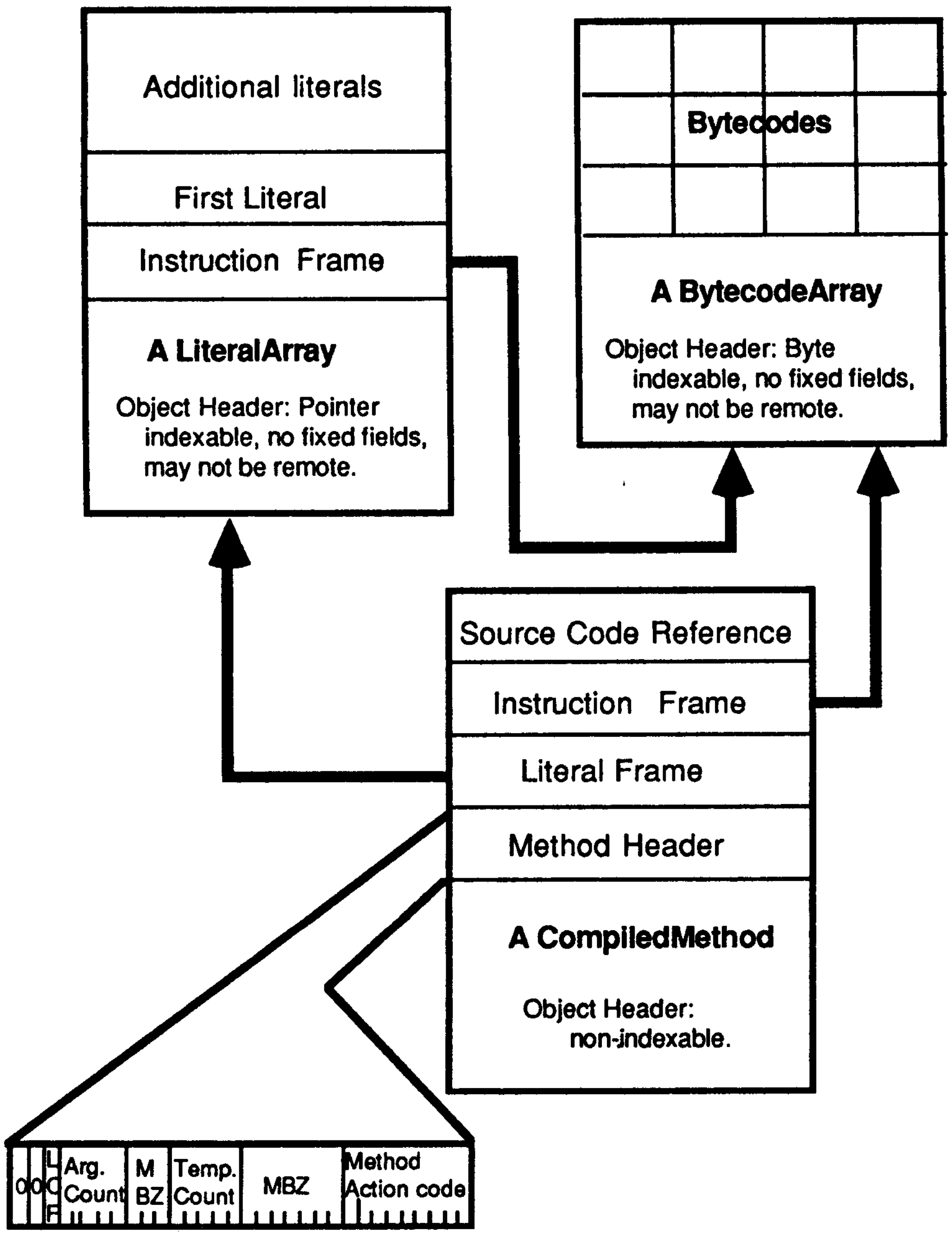
Compiled Method Representation

The representation of instances of class CompiledMethod was changed, making it consistent with the system's standard object architecture. The original form was used to reduce the number of objects needed to store a method, but with no real limit on the number of objects, the new form has several advantages. The new class CompiledMethod, for instance, may be subclassed. It also simplifies the garbage collector because there are now no objects which contain both binary and object pointer data. A compiled method is now represented by a composite structure consisting of three objects, a CompiledMethod, a BytecodeArray, and a LiteralArray. The root object contains four fixed fields: a control header, a source code reference, a reference to the BytecodeArray, and a reference to the LiteralArray[Figure 3].

The header fields provide the control information used by the interpreter to execute the compiled method. Within the standard compiled method representation used by previous interpreters, control information was divided between a header stored as the first literal and an optional header extension stored as the last literal. Information needed by the interpreter to determine how to execute a method was highly encoded and was divided among several fields of the header and header extension. This control information is now completely encoded within a single field of the new compiled method header. This significantly reduces the complexity of the interpreter code used to initiate execution of a method. In addition, there are now sufficient unused bits and encodings to contemplate extensions to the functionality of compiled methods.,

The source code field is used to locate the source code from which the method was compiled. It currently stores a small integer identifying a location within the standard Smalltalk source or changes file. Under the old representation, this value was represented by a 24-bit integer formed from three extra bytecodes added to the end of each compiled method. Because of its ad hoc nature, the old source code reference representation did not easily accommodate alternative mechanisms for storing source code. With the new representation, subclasses of CompiledMethod may use the source code field in any way which seems appropriate. For example, a subclass might choose to store a string containing the source code in the source code field.

The LiteralArray holds those object references which were stored in the pointer part of the old representation. The first literal



LCF (Large Context Flag)

0:	This method can use a small context
1:	This method requires a large context.

MBZ	(Must Be Zero).
Arg. Count	Number of arguments this method requires.
Temp. Count	Number of method tempories (exclusive of arguments).

Method Action Codes:

0:	Create a context and execute method
1-255:	Execute primitive with this number
256-287:	Return instance variable (n-256) of receiver
288:	Return the reciever
289:511:	Unused --- Invalid.

Figure 3 Compiled Method Structure

value, which in the old representation contained the method header, now stores a reference to the bytecode array. This reduces the amounting of working state information needed by the interpreter. The BytecodeArray contains only the actual bytecodes to be executed. It is no longer overloaded with other information such as literals or source code references.

The literal array could have been represented as a pointer indexable part of the root object. This would have reduced a compiled method to a composite of two objects instead of three. This choice was not made since we assumed that subclasses of CompiledMethod would want to add additional instance variables. Support for such subclasses using the two object representation would have complicated (and reduced the efficiency) of the bytecodes which access literals. For this same reason, subclasses of LiteralArray may not add any instance variables.

The Smalltalk protocol supported by class CompiledMethod is essentially the same as in previous versions. This is possible since the actual data structure is hidden within the class definition. In some cases the implementation of the protocol differs greatly from previous versions. For example, special primitives are no longer needed or provided for creating instances of CompiledMethod or for accessing literals. In a few instances, the protocol of CompiledMethod did not adequately hide implementation details. Knowledge of the source code reference technique was scattered throughout the virtual image with explicit use of size and at: messages to access the last three bytecodes containing the reference. New protocol was created for these situations and this protocol is now supported by both 4406 Smalltalk and the older systems.

Virtual Image

The virtual image for the 4406 is a direct derivative of the image used in our earlier implementations. A new subclass of SystemTracer was created which, when run in a 4404 Smalltalk image, produces a clone of that image suitable for execution by the new interpreter. The cloner converts all objects and oops to the representation used by the new system. CompiledMethods and MethodDictionaries are converted to the corresponding clone structures and new class definitions for these classes are substituted for those used in the old system. After a clone is created and executed with the new interpreter, additional class and method definitions are usually filed into the image. Most of these changes are optimizations which eliminate limited object space assumptions. For example, a number of the uses of the become: primitive can be eliminated.

One major change to the image is the manner in which existing instances of a class are handled when an instance variable is added to or removed from the class definition. The standard Smalltalk-80™ system attempts to "mutate" all existing instances of the class to use the new representation. This is accomplished by first finding all existing instances of the old definition. For each one, a new instance is created, using the new definition, and all data is copied from the old instance to the new instance. Finally the identities of the old and new instances are interchanged using the become: primitive. Under our new interpreter, this is potentially a very time consuming operation. Finding all instances of the old definition requires a search of the entire object space. Worst yet, each become: operation (and one is required for each existing instance) may require another complex scan of the entire object space. This situation is made even worse by the fact that the search for all instances may uncover zombie objects, which are eligible for garbage collection. In addition, our experience suggests that class definitions are most commonly modified while debugging and that in such situations, all existing instances are often discarded immediately after changing the definition. In order to eliminate the overhead of object mutation we developed a technique we call *lazy mutation*.

The intent of lazy mutation is to defer the mutation of objects until they are actually used in a computation. This eliminates the search for instances and avoids finding zombie objects. In addition, only objects which are actually needed are mutated. Lazy mutation is accomplished by replacing the method dictionary of the old class definition with a dictionary which defines only the message doesNotUnderstand:. In addition, the superclass of the old class definition is set to nil and a reference to the new class definition is stored within the old definition. When a message is sent to such an instance of the old class, a response to the message will not be found. Hence the doesNotUnderstand: method will be activated. This method contains the code to mutate the instance into an instance of the new class.

Performance

The appendix lists the results of executing the standard Smalltalk-80™ benchmark suite using 4406 Smalltalk compared to the results obtained with our previous system running on the same hardware. These results indicate that we were successful in achieving our goal of increasing the performance of the interpreter. While simple operations such as loading variables show only minor speed increases (and in a few cases are actually slower), more complex operations such as object creation are much faster. The net effect as shown by application level benchmarks (such as compiling a method) is that the new design is 25 to 50 percent faster.

This performance increase is readily apparent to a casual user poking around with the mouse. When the system is in such use the youngest region is garbage collected about once a second. These collections are seldom noticeable to the user but may be seen if you have a rapidly moving image on the screen. The basic image distributed with this interpreter has 33,000 objects. This image would be much too large to run on the old system.

A large application, Views has been run and benchmarked on both the 4406 and 4404 Smalltalks. For the computations benchmarked, the new system runs better than three times faster than the 4404 system. The larger speed increases were in tests which calculated Hilbert matrices in rational arithmetic. The new system did not expand into large integers as soon due to the larger range of small integers. For these cases the speed of the new system was up to eight times faster.

To file in the Views application on the 4406 takes about half the time that the 4404 takes. The performance increase for this operation is less for two reasons. The process is inherently I/O bound and I/O speeds are about the same for both systems. However, fileIn also creates a large percentage of permanent objects. During the fileIn process these objects typically must be moved from one region to another by the garbage collector several times. This increases the garbage collector overhead. We have considered techniques for allocating objects that we know will be relatively permanent into an older region directly but have not found a way to integrate this cleanly with the image.

The memory requirements of the new system are larger. The binary objects, which include bytecode arrays, are the same size but pointer objects are twice as big. Also the space used by the Baker half-spaces doubles the amount of virtual address space required for an image. This can cause thrashing during garbage collection if the size of the image is the same as the total real memory. The real memory does not have to be the same as the total virtual size since older regions are seldom collected. We have found that satisfactory performance on the 4406 requires about 1½ to 2 times the amount of real memory as the same image on a 4404.

References

[Bal82] Ballard S. and Shirron S. "The Design and Implementation of VAX/Smalltalk-80" in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983. pp. 127-150

[DeB76] Deutsch, L.P. and Bobrow, D., "An Efficient Incremental Automatic Garbage Collector," *Commun. ACM* 13,3. Sept. 1976, pp 522-526.

[DeS84] Deutsch, L.P. and Schiffman, A. "Efficient Implementation of the Smalltalk-80 System," *11th Annual ACM Symp. on Prim. of Programming Languages*, January 1984, pp. 297-302.

[GoR83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Kra84] Krasner G., Ungar D. and Malcolm M. "About Become" *Smalltalk-80 Newsletter* Sept. 1984, pp 1-2

[LiH83] Lieberman, H. and Hewitt, C. "A Real-Time Gargage Collector Based on the Lifetimes of Objects." *Commun. ACM* 26,6. June 1983, pp 419-429.

[Mcc83] McCullough, P. "Implementing the Smalltalk-80 System: The Tektronix Experience", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983. pp. 59-77.

[Mcc83a] McCall, K. "The Smalltalk-80 Benchmarks", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983. pp. 151-173.

[Ung84] Ungar, D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, April 1984. pp. 157-167.

[Wir83] Wirfs-Brock, A. "Design Decisions for Smalltalk-80 Implementors", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison-Wesley, Reading, MA, 1983. pp. 41-56.

[Wir85] Wirfs-Brock, A. "The Design of a High Performance Smalltalk Implementation", *Nikkei Electronics*, June 3, 1985, pp.233-245, (Japanese).

Appendix Benchmark Results

Standard Smalltalk-80 benchmarks [Mcc83a] execution times in seconds. The Small Object Space (SOS) times are for the Tektronix 4404 Smalltalk V1.5g interpreter. The Large Object Space (LOS) times are for Tektronix 4400 Large Object Space Smalltalk V2.1x. Both interpreters were run on a Tektronix 4406 with 4 megabytes of main memory.

Test name	SOS-4406	LOS-4406
load an instance variable	1.369	1.207
load 1 as a temp	1.085	1.06
load 0@0	1.086	1.062
load 1, 40 times; send ==	2.053	2.107
load nonRefcounted literal	1.143	1.102
load literal indirect (overflow refct)	1.421	1.193
store into an instance variable	1.99	1.093
store into a temp	0.949	0.911
add 3 + 4	1.2	1.176
test 3 < 4	1.239	1.151
multiply 3 * 4	1.534	1.43
divide 3 by 4	0.183	0.176
add 20000 + 20000	0.737	0.125
add 80000 + 80000	0.075	0.012
activate and return	2.001	1.687
short branch on false	1.212	1.039
simple whileLoop	2.863	2.893
send #at: (to an array)	0.42	0.366
send #at:put: (to an array)	0.743	0.616
send #at: (to a string)	0.501	0.437
send #at:put: (to a string)	0.632	0.518
send #size (to a string)	0.46	0.272
create 3@4	0.446	0.19
execute ReadStream next	0.768	0.594
execute ReadWriteStream nextPut:	1.025	0.773
send ==	1.206	1.211
send #class (to a point)	0.18	0.172
execute blockCopy: 0	1.987	7.339
evaluate the block: (3+4)	0.857	0.749
create 20 uninitialized points	1.286	0.433
execute aPoint x	1.635	1.279
load thisContext	3.04	1.914
send #basicAt: (to a set)	0.696	0.423
send #basicAtPut: (to a set)	0.889	0.626
3 perform: #+ with: 4	1.052	0.841
replace characters in a string	0.032	0.025
convert 1 to floating point	0.17	0.052
add 3.1 plus 4.1	0.218	0.061
call bitBLT 10 times	0.755	0.681
scan characters (primitive text display)	0.379	0.094
read and write class organization	2.333	2.121
print a class definition	1.776	1.421
print a class hierarchy	1.743	1.223
find all calls on #printStringRadix:	8.01	5.965
find all implementors of #next	1.562	1.146
create an inspector view	1.957	1.134
compile dummy method	4.066	2.885
decompile class InputSensor	2.645	1.828
text keyboard response using lookahead buffer	1.307	0.793
text keyboard response for single keystroke	3.82	2.345
display text	2.317	1.27
format a bunch of text	1.809	1.219
text replacement and redisplay	5.87	3.35