BrouHaHa - A Portable Smalltalk Interpreter

Eliot Miranda

Department of Computer Science And Statistics
Queen Mary College
University of London
Mile End Road
LONDON E1 4NS
ENGLAND

Phone: +44 1 980 4811 Ext 3920
UUCP: ...seismo!mcvax!ukc!qmc-cs!eliot
ARPA: eliot%qmc.cs@ucl-cs.arpa
JANET: eliot@uk.ac.qmc.cs

Abstract

BrouHaHa is a portable implementation of the Smalltalk-80 virtual machine interpreter. It is a more efficient redesign of the standard Smalltalk specification, and is tailored to suit conventional 32 bit microprocessors. This paper presents the major design changes and optimisation techniques used in the BrouHaHa interpreter. The interpreter runs at 30% of the speed of the Dorado on a Sun 3/160 workstation. The implementation is portable because it is written in C.

Introduction

BrouHaHa is the author's implementation of a redesigned Smalltalk-80 † virtual machine. The goal of the project was to provide a useably fast Smalltalk-80 machine that ran on available hardware. This goal has translated into an interpreter suited to 32 bit workstations running Unix ††, with a bitmapped display, a keyboard and mouse. BrouHaHa is far more efficient than the standard specification on conventional 32 bit microprocessors. The bulk of the BrouHaHa virtual machine is portable across 32 bit machines that have a C compiler.

BrouHaHa uses a novel formulation of 'compiled bitblt' that provides almost portable bitmapped graphics.

Overheads Incurred By The Standard Implementation

Smalltalk is specified by a description of a virtual machine. This virtual machine is presented as a set of Smalltalk classes and methods in the "blue book" [Goldberg83]. The specification is not intended to be efficient, but is comprehensible, and almost completely correct.

Considerable overheads are associated with the advanced features of Smalltalk-80. Implementations of the standard specification have been carefully analysed, and the overheads quantified [Falcone83, Myers83, Ungar83]. The major overheads are:

1. Object Allocation & Garbage Collection

The blue book specification allocates, and reclaims, objects very frequently; a new object is needed to hold the state of each method activation. The specification uses a reference counting garbage collection scheme. This is done to ensure good interactive response, at the expense of overall efficiency. The costs are documented in [Ungar84].

† Smalltalk-80 is a registered trademark of the Xerox corporation.
†† Unix is a trademark of AT&T

2. **Object Table Indirection**

Smalltalk uses a special data structure to represent objects. The object identifier (a 16 bit word) is used to index a table of Object Table Entries (OTEs), the Object Table (OT). Each OTE contains the object's reference count, some flags and the address of its body. The body contains a size, a class entry and the object's fields. This structure puts an overhead on all field references because they are indirected through the OT, but it does have advantages, as follows.

Storage compaction is cheap since an object's body can be moved and then its address entry updated. There is no need to scan the object space fixing up old references to point to the new position as one would have to do in a single level indirection scheme. Similarly, the OT can be used to mutate an object by swapping its OTE with that of another object.

3. **Context Allocation and Argument Copying**

The runtime state of a Smalltalk method is stored in a Context object. It contains a fixed amount of stack space, a stack and instruction pointer, message receiver, arguments, method and return linkage. Smalltalk method activations are linked lists of context objects and therefore do not form a contiguous stack.

A new context is needed to evaluate each new Smalltalk method. This context is allocated from the heap and then initialized, by setting up return linkage (sender), stack pointer and instruction pointer, and filling its stack space with nil , Smalltalk's null object. The receiver and arguments are then transferred from the sending context to the new active context. On return to the sender context, the active context is swept for object references and then returned to the free storage heap.

4. **Accessing the Next Bytecode and Top of Stack**

The blue book specification defines the instruction pointer and stack pointer to be integer indices. Accessing the top of stack and fetching the next instruction are defined to be indexing operations on the current context and the current method.

5. **Method Lookup**

Each time a message is sent to an object the method to be evaluated is looked up in the object's class hierarchy. This is the essence of Smalltalk's polymorphism and incremental compilation. The blue book specification includes a method lookup cache that remembers the results of previous lookups.

6. **Bitmapped Graphics**

Smalltalk is an interactive, graphical environment. All display output is done through one primitive method, BitBlt [Goldberg83]. BitBlt is crucial to the apparent interactive performance of the system and must be implemented efficiently.

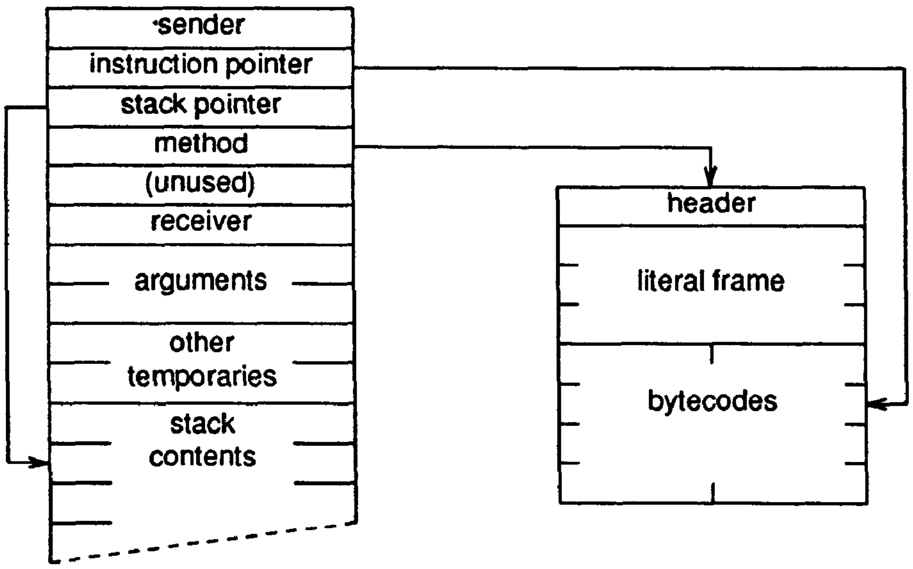

Figure 1. The Evaluation of a Smalltalk Method

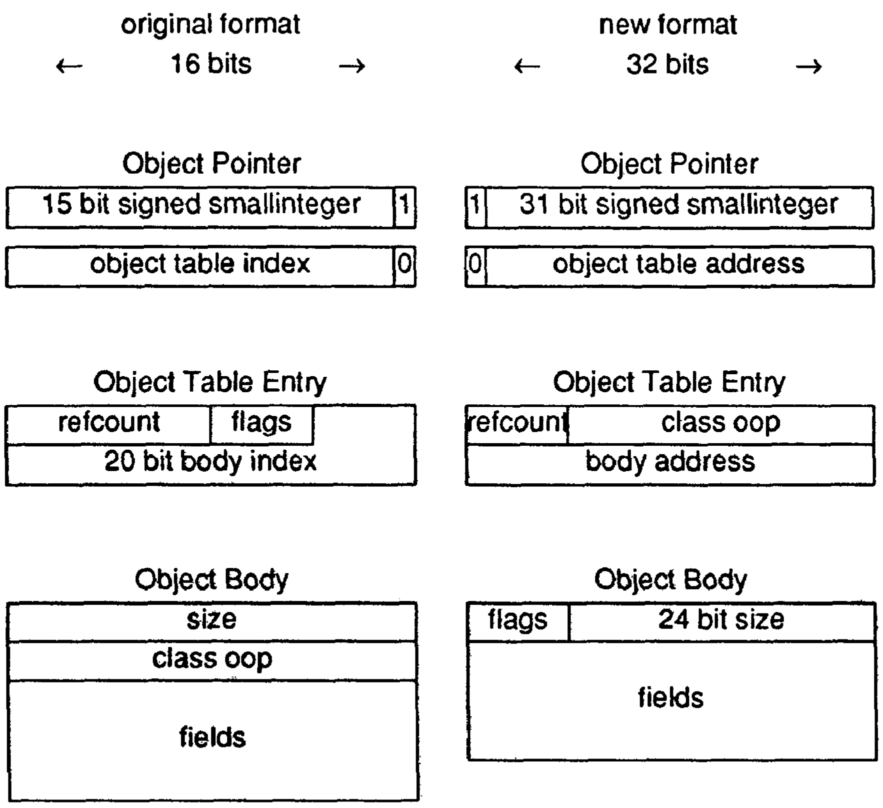

Figure 2. Object Formats in The Blue Book Specification and in BrouHaHa

A Significantly Faster Implementation

The rest of this paper presents the major design changes and optimisation techniques used in the BrouHaHa interpreter.

Straightforward Changes

References to the stack and to the next bytecode are so frequent that they should be fast. So the instruction pointer and stack pointer of the active context are stored as pointers into the bodies of the method and context, rather than as integer indices.

The representation of an object pointer (oop), object table entry and object body are changed to suit a 32 bit architecture.

Oops are either 31 bit signed integers or 32 bit addresses of object table entries, and are distinguished by the sign bit, negative oops encoding the 'smallintegers'. An object table entry is 64 bits. It contains an 8 bit reference count, 3 bytes for the class oop and a 32 bit pointer to the body. An object body has a 32 bit header followed by its fields. The header has a byte for various flags and a 24 bit size field.

A number of benefits accrue as a result of these changes. At the Smalltalk level, it is possible to have many more objects, and with a larger size field it is possible to have objects larger than 64K words. At the virtual machine level, object table indirection and reference counting are faster because they are pointer indirection operations, instead of indexing ones. Primitives that needed to use large integers no longer have to because of the increase in the range of smallintegers.

Several other simple optimisations have been tried, such as a special pool of free contexts to reduce allocation and initialisation overhead, but these were found to be tedious to implement and had little effect. The approach detailed in the next section yields a greater improvement.

The Context Cache

The standard specification incurs large overheads in its use of context objects to hold runtime state. The idea of the Context Cache is to represent runtime state in a more conventional way, while still preserving the programmer's access to this state via context objects. This idea has been developed elsewhere [Suzuki83, Baden83] but BrouHaHa is different in detail and handles blocks in a new way.

In conventional architectures, procedure call, argument passing and procedure exit are simple, efficient operations. Typically, arguments are pushed onto the stack by the caller, followed by the program counter. The callee then pushes the old frame pointer, increments the stack pointer to allocate any temporary variables and proceeds. On return the callee decrements the stack pointer, restores the frame pointer and returns. The caller then adjusts the stack pointer and continues.

In addition to these operations, Smalltalk allocates a new context, copies the arguments from the calling context to the new context and nils out the new context's stack. On return the result is copied onto the caller's stack and the new context is reclaimed.

In BrouHaHa, a special area of memory, called the Context Cache, is used to hold a fixed number of the most recent method activations. The rest of the activation stack is held in Context objects, stored on the heap, as usual. Two interpreters are used. One is optimized for execution of activations in the cache, the cache interpreter. The other is compact and slow, and executes activations on the heap.

The context cache is formed from two blocks of memory. One block contains the contiguous stack for the 'cached contexts', the cache stack. The other block is a circular list of 'cache frames' containing pointers to each activation's environment. These are the method oop, and pointers to the top of stack, next bytecode, base of stack, method body, and receiver body. The first, or lowest, frame in the context cache contains the oop of the top Context in the heap-resident activation stack.

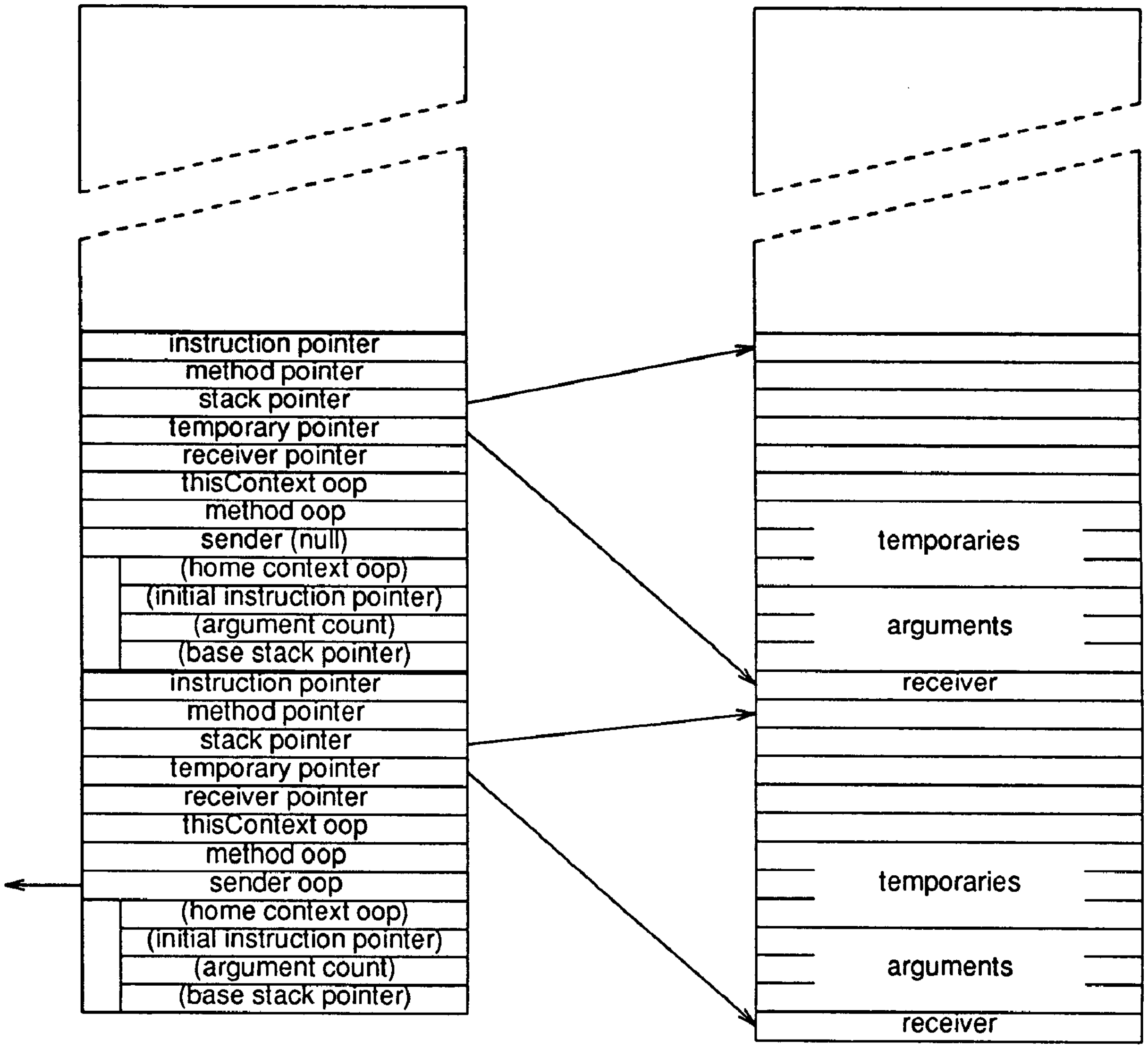

Figure 3. The BrouHaHa Context Cache

When a message is sent from the currently active frame, a new frame is needed to evaluate it. The next frame in the circular list of frames is examined. If it is in use, it must be the first, or bottom, frame in the cache and the cache must be full. In this case, the bottom frame is freed for use by flushing it onto the heap; an ordinary heap context is allocated and initialised from the information in the bottom frame and its associated part of the cache stack. The next frame's sender is set to point to the heap context, inserting it at the head of the list of heap activations.

The empty frame can now be used to evaluate the new method. The receiver is on the stack beneath the arguments. The new frame sets its temporary pointer to point at the message receiver, so that it accesses the receiver and arguments through the temporary pointer. The stack pointer of the previous frame is set to point beneath the receiver, and the instruction pointer is saved in the previous frame. Then any unused temporary variables in the new context are set to nil.

Smalltalk heap contexts have a fixed amount of stack space (room for either 12 or 32 oops). In the context cache, the stack space is large enough for each frame to use the maximum space of 32 oops. As more method activations are created in the context cache so the top of stack will grow upwards. Occasionally there will not be enough room at the top of the cache stack to fit in all of the stack of the new frame. When this occurs, the receiver and arguments are copied to the bottom of the stack. Since the cache stack memory is big enough for all the cached contexts' stack, there is no need to check for overlap of the new frame's stack with that of the bottom frame. This is because that stack space would have been used by a frame that will have been flushed onto the heap.

On return from a method activation, the sender field of the returning frame is checked. If the sender is non-nil, it contains the oop of the top context in the heap-resident activation stack. The result is copied to the top heap context and control passes to the heap interpreter. The heap interpreter is similar to the standard specification and it does reference count the stack. However, as soon as the heap interpreter sends a message it creates a new frame in the cache, transfers the receiver and arguments onto the cache stack, and transfers control back to the cache interpreter. Since the heap interpreter runs very infrequently it accounts for a very small part of the overall execution time. It is therefore optimized for space instead of speed.

The use of the context cache results in much reduced context allocation, reclamation, initialisation, and argument copying. As an example, consider the fibonacci function:

```
Integer fibonacci
    self > 2
          ifTrue: [↑(self - 2) fibonacci +
                      (self - 1) fibonacci + 1]
          ifFalse: [↑1]
```

This version returns the number of function calls needed to evaluate it. In a conventional implementation, Smalltalk will allocate *N fibonacci* contexts to evaluate *N fibonacci*. Using the context cache, BrouHaHa need only allocate as many contexts as there are cache overflows. For a cache of depth *C* there are less than *(N - C) fibonacci* overflows. For a cache of depth 8, evaluating *16 fibonacci* results in a reduction in context allocation by a factor of about 60.

Since a cache frame is not defined by Smalltalk, it can be bigger than a conventional Context; more information can be stored. Each frame stores the stack pointer and instruction pointer, rather than the indices into the context stack and method. Each frame also holds the addresses of the method body and receiver body to avoid object table indirection when accessing literals or receiver variables.

Deferred Reference Counting

Using conventional heap contexts, stack operations cause many redundant reference counting operations. Many pushes and pops may be performed, each needing at least one reference counting operation, even though the net change in reference counts over these operations may be zero.

The context cache can be used to reduce this redundant reference counting overhead. The technique is presented by Scott Baden [Baden83]. The idea is not to reference count the contents of the cache stack. It is rather to keep track of objects that might be referenced from the cache stack in a table called the "zero count table" (ZCT). Every time a new object is created by the cache interpreter it is placed in the ZCT. Also, whenever a heap reference counting operation (e.g. assignment to an instance variable) causes an object's reference count to fall to zero, it cannot reclaimed since it might be referenced from the cache stack. Instead it is placed in the ZCT. A bit in the object body header is used to mark objects in the ZCT, so as to avoid multiple entries.

Periodically the ZCT will fill up, at which point the cache stack and ZCT are scanned. Any object not referenced from the cache stack that has a zero reference count can be reclaimed. Using this technique cache stack pushes and pops can be done without reference counting, and can be made simple, data-movement operations. The technique has the disadvantage of wasting storage space between ZCT reclamations, but the overhead of reference counting is greatly reduced.

Preserving Smalltalk Semantics

Smalltalk gives the programmer access to the active context. The pseudo-variable thisContext refers to it and is translated into a push-active-context bytecode. For example, here is the method used to insert a breakpoint while debugging.

```
Object halt

    NotifierView
        openContext: thisContext
        label: 'Halt encountered.'
        contents: thisContext shortStack
```

A pointer to the active context is also needed on process switch; it is stored in the suspended context variable of a process object.

The context cache must maintain this access transparently. The simple way to ensure this would be to flush the cache on every reference to the active context. This is not done, and some care is taken to avoid cache flushes. Multiple context caches are used to avoid cache flushes on process switch. Cache allocation, on a least recently used basis, only occurs when a message is sent by the heap interpreter.

The cache interpreter handles the push-active-context bytecode and process switch in a special way. A pseudo-object is created to represent the context and used instead. This object has a nil class oop, and so understands no messages. A pointer to the context's cache and cache frame is stored in the pseudo-object. The message lookup routine is modified so that when an object is found not to understand a message its class is checked. If it is a pseudo-context, then it is converted into a standard heap context †, and the message resent.

When a pseudo-context is converted into a heap context, all the frames up to and including the pseudo-context's frame are flushed onto the heap. If the pseudo-context is that of the active frame, then control is transferred to the heap interpreter and the message resent from there.

The object table allows the pseudo-object to mutate into a heap context by allocating a new heap context, initialising it from the pseudo-object's frame and swapping the object table entries of the pseudo-context and the heap context †.

† An alternative approach is to allocate enough space for the context and only use part of it for the pseudo-context [Caudill86].

Any cache frame that creates a pseudo-context, either by the push-active-context bytecode or by process switch, keeps a note of the pseudo-context. Multiple references to the active context will access the same pseudo-context. Special care is taken when returning from cached frames that have pseudo-contexts. A corresponding heap context is created, otherwise subsequent references to the pseudo-object would access a now invalid (or different) context, resulting in chaos.

The process switch primitives are modified to allow pseudo-contexts to be stored as suspended contexts in Process objects. On process switch, control is transferred to either interpreter as appropriate. Any access to pseudo-contexts stored in processes will involve sending a message to the pseudo-context. This will result in the cached context being flushed to the heap, invisible to the programmer.

Block Contexts

Blocks are Smalltalk's version of lambda functions, and are used to implement most Smalltalk control structures. Blocks are created by sending the blockCopy: message to the active context. In response, a BlockContext object is created, which represents the block. The block stores the active context as its home context, an argument count (supplied as the argument to the blockCopy: message), and the point in the home context's method at which to begin execution. When a block is sent a value message, the context that sent the value message is remembered as the block's caller, and the block is evaluated.

Initially, the number of arguments supplied with the value message are checked against the argument count of the block. If they agree, the block is evaluated in an environment taken from the home context. Execution starts at the point given by the initial instruction pointer.

There are three problems with blocks as specified in the blue book, two are general to Smalltalk and one is specific to the BrouHaHa context cache. The general problems are

1. Standard Smalltalk-80 blocks are non-reentrant. This is because a block contains its own return linkage and saved state (caller, instruction pointer and stack pointer). Multiple activations of the same block overwrite each others' stack pointer, instruction pointer and return linkage. So if a block calls itself it can only return to itself.

2. Block arguments are allocated as temporary variables stored in the home context. Multiple activations of the same block overwrite each other's block arguments.

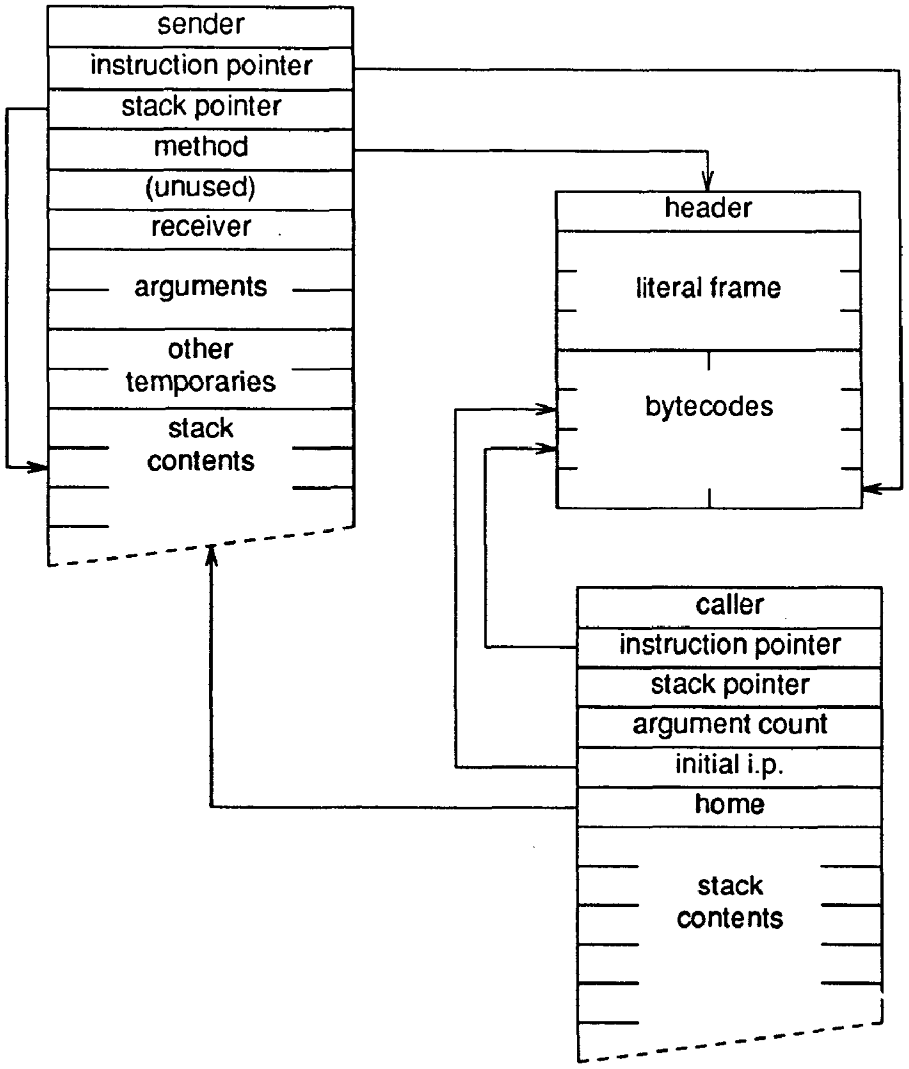

Figure 4. The Evaluation of a Smalltalk-80 Block

The Tektronix 4404 implementation [Caudill86] fixes the first case at the implementation level. In the Tektronix system tail-recursive functions using blocks work, even though the language has not been changed. As an illustration of the two general problems, here are two formulations of the factorial function.

Tail-recursive:

```
| factblock |
factblock ← [:n|
    n > 0 ifTrue: [n * (factblock value: n - 1)]
          ifFalse: [1]].
factblock value: 3
```

Head-recursive:

```
| factblock |
factblock ← [:n|
    n > 0 ifTrue: [(factblock value: n - 1) * n]
          ifFalse: [1]].
factblock value: 3
```

The block argument n is allocated as a temporary variable in the home frame. So in fact, with blocks as specified, the above code is equivalent to the following:

Tail-recursive:

```
| factblock n |
factblock ← [:blockarg|
    n ← blockarg.
    n > 0 ifTrue: [n * (factblock value: n - 1)]
          ifFalse: [1]].
factblock value: 3
```

Head-recursive:

```
| factblock n |
factblock ← [:blockarg|
   n ← blockarg.
   n > 0 ifTrue: [(factblock value: n - 1) * n]
         ifFalse: [1]].
factblock value: 3
```

In standard Smalltalk-80 and BrouHaHa with blocks both pieces of code cause an error since blocks are non-reentrant. In the Tektronix 4404 implementation and in BrouHaHa with redefined blocks the tail recursive version has value six as one would wish. This is because the argument n is pushed onto the stack before the recursive call. However, the head recursive version returns zero since n is overwritten with zero in the last activation before any multiplication is done.

There are two versions of BrouHaHa. One uses the standard definition of blocks, and does not permit reentrance. The second version uses a changed definition of blocks, which is bootstrapped with the blocks version.

The context cache has performance problems with blocks. Blocks are created and evaluated frequently, but the standard blockCopy primitive creates them on the heap. If blocks are implemented as in the standard specification, then use of blocks will necessitate cache flushes. There are at least four alternatives.

1. Flush the home context when the block is created, and flush the caller when the block receives a value message.

2. Use a pseudo-context to represent the home context and flush the home context and caller on value.

3. Convert the block into a cached form on value.

4. Change the definition of blocks to avoid cache flushes.

The standard BrouHaHa interpreter takes the first approach. BlockCopy: is not handled primitively; instead, the blockCopy: message is actually sent. The pseudo-context that receives the message will be flushed onto the heap by the message lookup routine. The last alternative can be efficient and, with the new definition, blocks can be made re-entrant. In the redefined interpreter and its associated image, the change in the definition of blocks is as follows. A new class is created, called Closure, whose instances are used to represent unevaluated blocks. When the blockCopy primitive is evaluated it returns a Closure rather than a BlockContext. The Closure contains its home context (which may be a pseudo-context), initial instruction pointer and argument count, as do BlockContexts, but no more. On receiving a value message, a new block is created from the closure. This means that, from the point at which a block is sent a value message, execution is identical to that of the standard system †. The value primitives for Closures create a new frame in the cache, a block frame. The block frame takes its receiver and temporarys from the home context which may be on the cache or the heap. As for ordinary cached contexts, no argument copying or stack initialisation is needed on evaluating the block. So using closures, the cache is not flushed on either blockCopy: or value

Closure is simple to implement. It has 25 methods, all essentially copies of methods in BlockContext, and needs three new primitives, new versions of blockCopy:, value and valueWithArguments:

The SystemTracer is a Smalltalk tool that allows one to write out the entire contents of the system in the form of a snapshot file that may be run by the virtual machine. The tracer allows one to change image formats, object formats, and in this case, to determine the object identifiers of specific objects. The blocks version of BrouHaHa is used to run the SystemTracer, which writes out an image in which class Closure has a known object identifier, or 'guaranteed pointer' (this is the case for all objects that need to be known by the interpreter). The closures version of the interpreter is then run on the new image. The SystemTracer is modified for the new 32 bit format. Apart from the inevitable byte-ordering problems this is easy to do.

Of course having re-entrant blocks is nice. However, fixing the sharing of arguments is more difficult, requiring modifications to the compiler and additional bytecodes to access nested variables. This work has not been undertaken in BrouHaHa.

A Portable Implementation

BrouHaHa is written almost entirely in C. Everything except screen output and keyboard and mouse input is portable between 32 bit machines with a C compiler. BrouHaHa currently runs on the Whitechapel MG1 and MG200 workstations and on Sun 3 machines. The MG1 is based on the ns32016 microprocessor, the MG200 uses the ns32332. Both are little-endian, and on these machines BrouHaHa runs at 7% and 16% of the Dorado

† Note that, inside a block, thisContext refers to the active, block context, not to the closure, so that code that accesses blocks through thisContext is not changed.

respectively. The Sun 3 uses the mc68020 microprocessor, which is big-endian. On the Sun 3/160 BrouHaHa runs at 30% of a Dorado.

BrouHaHa can be broken down into three sections of code, machine independent code for the interpreter, machine independent code for debugging the interpreter, and machine dependent code for keyboard and mouse input and screen output.

interpreter	9500 lines
machine dependent i/o	1750 lines of C
	750 lines of assembler
virtual machine debugger	2500 lines

Achieving portability of the interpreter and debugger is not difficult. The C macro preprocessor is used to choose the byte-ordering of an Object Table Entry and an object body header. Other than this the interpreter is unchanged between machines. Achieving efficient code generation is more difficult.

C Optimisations

The interpreter makes most frequent use of the stack pointer, the instruction pointer, and the current frame pointer. Ideally these should be held in machine registers. C allows the programmer to suggest which variables should go in registers, so called register variables; but register variables can only be declared within functions; there are no global register variables in C. However, it is possible to get global register variables by using a little legerdemain.

Typical C compilers allocate registers in a predictable way. If the sp, ip and fp are declared as register variables in every function in the same order, the compiler will allocate the same three registers to hold these variables in all functions. Common declarations are provided using the C macro preprocessor; a macro called BEGIN contains the declaration of the global register variables, and all function bodies begin with BEGIN.

```
#define BEGIN\
{ register OOP   *stackPointer;\
  register BYTE *instructionPointer;\
  register FRAME *frame;

  ...

/* smallInteger add, primitive 1 */
BOOL   _pSIAdd()
BEGIN
  register SIOP a;

  if (isSmallInteger(a = (SIOP)stackTop)) {

      a = intValueOf(a);
      a += intValueOf(*--stackPointer);
      if ( isIntValue(a) ) {
          *stackPointer = intObjectOf(a);
          RETURN(true);
      }
      stackPointer++;
  }
  RETURN(false);
END
```

The compiler is run to produce assembler (all unix C compilers allow this). An editor script is then run on the assembler which alters the register save and restore masks in procedure entry and exit instructions to remove the saves and restores of the global registers. This simple technique implements global register variables. On the Sun, the interpreter is 35% faster and code bulk is reduced by 14%† as compared with a version that does not use global registers.

Some C optimisers will try to remove redundant assignments. A procedure which initialises any of the global register variables will be optimised away, since the optimiser assumes the variables are only in scope within the procedure and hence considers the assignments redundant. To get around such optimisers the results of any assignments to the globals must be used. The macro preprocessor is used to generate a call of a non-existent function before a return from or end of a function.

```
#define END REGFIX(stackPointer, \
                   instructionPointer, \
                   frame); }

#define RETURN(e) \
if (1) { \
    REGFIX(stackPointer, \
           instructionPointer, \
           frame); \
    return e; \
} else ††
```

† This figure is a percentage reduction of the code for the interpreter and input/output, not of the final executable module, which includes large libraries.
†† The if (1) { <statements> } else idiom allows one to expand a macro containing a series of statements correctly in conditional statements, i.e. as one would use a function.

Again, an editor script is run on the assembler output to remove all calls of the dummy function REGFIX.

The global register optimization technique is applicable on many machines, and is used on both the Whitechapel and the Sun implementations.

A number of miscellaneous techniques are used to improve code generation in BrouHaHa. The macro preprocessor is also used to provide in-line functions: operations like object table indirection, stack access and reference counting are defined by macros. Editor scripts are sometimes used as simple peep-hole optimisers. For example, converting three instructions addb <n>,<addr>, cmpb 0,<addr>, beq <label>, into the acbb <n>,<addr>,<label> instruction on the ns32000, or removing the bounds check on the main bytecode switch in the cache interpreter.

BitBlt

Smalltalk uses the BitBlt primitive for all screen output, and it must be fast if the system is to have a good apparent interactive response. The operation takes two subrectangles of pixels from a source and destination bitmap, and combines these with a halftone pattern according to one of 16 possible combination rules, placing the result in the destination. This algorithm is complex, and is slow if done interpretively, but has too many cases for each to be dealt with by a static function. On machines without special hardware a very successful technique has been developed [Pike84] known as 'compiled rasterop' or 'compiled bitblt'. Very briefly, each call is handled by dynamically compiling a sequence of instructions to perform the operation. Small chunks of machine code, a few instructions in length, are concatenated into a near optimal function and the resulting code is called. Because any necessary tests are done once during the compilation process, and eliminated from the code so produced, the compiled code is much faster than the corresponding interpretive version.

Cases with a width of less than or equal to 16 pixels can be done by accessing a single word in the source and destination bitmaps per scan-line. These cases are best handled by an assembler function with a loop for each combination rule. In BrouHaHa, the assembler function is derived from a version of the code written in C. The assembler generated by the compiler is optimised by hand. The C code is fairly portable, given differences in bit-ordering, and the optimisations are simple.

The on-the-fly compilation version is written in C. The Whitechapel implementation was a first attempt at compiled bitblt. It worked by defining the 32000 machine code with a number of macros and then copying the relevant machine instructions to a block of code which is then called. Writing and debugging took about three weeks and was very tedious. It is also completely dependent on the 32000. When porting BrouHaHa to the Sun a better technique was used.

The required code chunks are written in C as about 90 small functions (most are one line long). Each contains common variable declarations, as does the main routine, again using a BEGIN macro. An editor script is used to remove prologue and epilogue from the functions in the compiler-produced assembler, leaving behind only the relevant code chunks in the bodies of the functions. C allows the programmer to take the address of a function. The main routine can then concatenate the code chunks by copying the memory from the start of one small function to the next into a block of memory.

Typically, it is the callee's responsibility to set up the new frame pointer and save registers. The compiled code does not do this, and can therefore access the registers and variables of the main routine. The main routine appends the machine code for a simple return instruction to the code, rather than a full epilogue. Thus the compiled code can access any variables in the main routine by duplicating their declarations inside each small function.

BrouHaHa compiles code for the outer and inner loops and calls it. The only machine codes that need to be known are those for the return-from-subroutine and decrement-and-branch instructions, since the compiler and assembler produce the rest of the machine code for the programmer.

Starting with a slow interpretive version of the routine in C, it took less than 20 hours to write and debug the compiled bitblt function. The resulting primitive is 10 times faster than the interpretive version. The code is considerably more portable than that for an implementation that needs to define machine code.

Colour

Using this simplified technique rasterop has been extended to use 8 bit pixel colour. The copyBits primitive now allows the source, destination and halftone forms to be either 1 bit deep bitmaps (as in standard Smalltalk-80) or 8 bit deep bitmaps. Further, the source and halftone may be integers in the range 0 to 255, representing pure colours. Six more combination rules have been added to suit colour rasterops; maximum, minimum, sum, average, source minus destination and destination minus source. These rules operate on each byte sized pixel.

If an 8 bit deep source or halftone is combined with a 1 bit deep destination then the source or halftone is compressed, 16 pixels at a time, according to one of eight colour to monochrome functions. These functions operate on a pixel and the current colour to monochrome value (cmv). The functions are foreground, background, pixel equal to cmv, pixel not equal to cmv, pixel greater than cmv, pixel less than cmv, pixel and cmv, and pixel nand cmv. If a 1 bit deep source or halftone is combined with an 8 bit deep destination then black pixels are converted into the current foreground colour and white pixels are converted into the current background colour.

Concluding Remarks

BrouHaHa was written by one person over a period of two years part time. It was started on the Whitechapel MG1 as a third year undergraduate project, and completed while the author was working as a research assistant at the University of York. In that time it changed from a conventional 16 bit implementation running at less than 1% of a Dorado to a 32 bit implementation with fast graphics, context cache and deferred reference counting, running at 16% of a Dorado on the Whitechapel MG200. BrouHaHa was ported to the Sun 3 in under 20 working days at Queen Mary College, University of London. It is to be used as a vehicle for exploring the use of multimedia in an office environment, where access to the virtual machine will be an advantage.

Future Work

BrouHaHa is still some 3.5 times slower than the Deutsch and Schiffmann Smalltalk implementation [Deutsch84], which takes the on-the-fly compilation approach seriously, and compiles bytecoded methods into native machine code. BrouHaHa currently spends 10% of its time in the code that dispatches on the current bytecode (a C switch statement), and 15% of its time setting up the new frame. It would be interesting to see if the dynamic translation approach could be adapted in BrouHaHa, in an easily portable way, perhaps by taking a threaded-code approach.

The structure of CompiledMethods ought to be changed in line with the changes in [Caudill86].

Speculations

Compilers for machine-orientated languages, like C, could be written that allow the programmer much greater control over code generation and optimization, for example, through global register variables or the specification of code sequences for frequently executed operations. Such compilers should be able to produce better, safer code, than the rather arbitrary techniques used to speed up BrouHaHa.

Under the torrent of ever faster silicon, such as the AMD 29000, reputed to be some 3.5 times faster than the 68020, it may be that a portable implementation will eventually be faster than an assembler implementation by virtue of its portability.

Acknowledgments

I would like to thank Tony Williams for first introducing me to Smalltalk in 1983. Thanks also to Mike Harrison who was my third year tutor and subsequently a tolerant employer. Finally thanks to Steve Cook and Steven Sommerville for their patient help in preparing this paper.

References

Goldberg83.
Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

Falcone83.
Joseph R. Falcone, "The Analysis of the Smalltalk-80 System at Hewlett-Packard," in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, pp. 207-237, Addison-Wesley, Reading, MA, 1983.

Myers83.
Richard Myers and David Casseres, "An MC68000-Based Smalltalk-80 System," in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, pp. 175-187, Addison-Wesley, Reading, MA, 1983.

Ungar83.
David M. Ungar and David A. Patterson, "Berkeley Smalltalk: Who Knows Where the Time Goes?," in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, pp. 189-206, Addison-Wesley, Reading, MA, 1983.

Ungar84.
David M. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pp. 157-173, April 1984.

Suzuki83.
Norihisa Suzuki, *Implementing Smalltalk-80 on Sword*, Carnegie-Mellon University, 1983.

Baden83.
Scott B. Baden, "Low-Overhead Storage Reclamation in the Smalltalk-80 Virtual Machine," in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, pp. 331-342, Addison-Wesley, Reading, MA, 1983.

Caudill86.
Patric J. Caudill and Allan Wirfs-Brock, "A Third Generation Smalltalk-80 Implementation," *Proceedings of OOPSLA'86*, vol. 21, no. 11, SIGPLAN, November 1986.

Pike84.
Rob Pike, Leo Guibas, and Dan Ingalls, "Bitmap Graphics," SIGGRAPH'84 Course Notes, AT&T Bell Laboratories, 1984.

Deutsch84.
L. Peter Deutsch and Allan M. Schiffmann, "Efficient Implementation of the Smalltalk-80 System," *11th Annual Symposium on Principles of Programming Languages*, pp. 297-302, January 1984.