Implementing Smalltalk-80 on the ICL PERQ

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science.

October 1984

Mario I. Wolczko
Department of Computer Science

Abstract

The Smalltalk-80 system is an interactive programming environment that consists of an object-oriented high-level language, a rich kernel of predefined data types and a graphical user interface. Implementation of the Smalltalk-80 system on a particular computer requires emulation of the *Smalltalk Virtual Machine*, a hypothetical computer with an object-oriented architecture and instruction set. This dissertation describes a high-level language implementation of the Smalltalk Virtual Machine on the ICL PERQ, and investigates the feasibility of a microcoded implementation.

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Since graduating in 1983, Mario Wolczko has been a postgraduate student in the Department of Computer Science at the University of Manchester.

To my parents

Acknowledgements

Many people have been instrumental in the creation of this dissertation. I would like to thank my supervisor, Steve Holden, for his guidance and support, and for suggesting the project in the first instance. I would also like to thank all the students and staff associated with the M.Sc. course in System Design; the many varied experiences of the past year have been enjoyable and rewarding. I would also like to acknowledge the work of those that have contributed indirectly to my efforts: my thanks go to the creators of the Smalltalk-80 system, without which this dissertation would be naught; to the creators of the UNIX system, for many pleasant hours spent in work and in play; and to Professor Donald Knuth, for giving the world the TEX system, which has been a joy to use. My thanks also go to Tony Arnold, for the help he provided in the creation and use of the TEX system within the Department of Computer Science. Finally, I would like to thank my family for their constant support and encouragement.

This work was supported by an award from the Science and Engineering Research Council.

1. Introduction

The Smalltalk-80 system

The Smalltalk-80 system represents the culmination of over a decade of research into object-oriented systems done at the Xerox Palo Alto Research Center (*PARC*). The latest release of Smalltalk, the third in a series, is for general distribution, whereas the earlier releases (in 1972 Shoch Smalltalk-72 [SHOC76] and 1976 Ingalls Smalltalk-76 design and implementation [INGA76]) were only for internal use within Xerox. To meet the demands placed on an implementation that was to be portable across a wide range of machine architectures, the designers of Smalltalk decided to define a hypothetical architecture on which the final system would run. To port Smalltalk-80 to a new machine, an implementor would then merely have to emulate the *Virtual Machine* Krasner Virtual Machine Byte [KRAS81], as the abstract architecture came to be known.

As a test of this implementation technique, Xerox invited four outside companies to attempt to port the Smalltalk system onto their own machines. Apple Computer, Digital Equipment Corp., Hewlett-Packard and Tektronix were successful in running Smalltalk-80 on Motorola 68000-based machines (Meyers Casseres MC68000-based Smalltalk [MEYE83], McCullough Smalltalk-80 Tektronix [MCCU83]), DEC *VAX*-11s and DECSYSTEM-20s (FALC83a, BALL83), under a variety of operating systems. These implementations, together with those done at Xerox on the *Dolphin* and *Dorado* computers, cleared the way for the general release of the Smalltalk-80 system. A definition of the Virtual Machine (written in Smalltalk) was also published Goldberg Robson Smalltalk Language Implementation [GOLD83], and serves as the main reference for future implementors. Also, the implementors documented their experiences, and their descriptions were collected together in a book Krasner Smalltalk Bits of History KRAS83 [KRAS83]. However, none of the companies involved released a Smalltalk product, and as Xerox only supports Smalltalk on its own machines, an implementation on a commonly used computer is still not widely available.

The ICL PERQ

The ICL PERQ personal workstation is widespread among the UK scientific community, and superficially has all the characteristics required of a Smalltalk host (Hopgood PERQ [HOPG82], PERQ: Introduction [ICL82a]). It possesses a powerful bit-sliced central processor, a high-resolution bit-mapped graphics display, and a pointing device in the form of a tablet and four-button puck. These features, together with a megabyte of central memory and a real-time clock, indicate that a good implementation of Smalltalk on the PERQ might be feasible. Furthermore, the ability to change the microcode at the heart of the processor suggests that an implementation with excellent performance is possible. To date, most Smalltalk implementations in a high-level language or in assembly code have suffered from poor performance, in comparison to implementations of other language systems on the same machines. The only implementations with satisfactory performance levels are those done in microcode, at Xerox, on the *Dolphin* and *Dorado* computers. Since the PERQ shares a common ancestor with these machines (in the Xerox *Alto* THAC79), it is only natural to consider it as a potentially powerful base for a Smalltalk implementation.

Outline of the dissertation

In this dissertation I hope to show that a naïve implementation of Smalltalk on the PERQ (in a high-level language) is straightforward, and that a microcoded implementation with exceptional performance is feasible.

Chapter 2 presents the issues involved in implementing Smalltalk. The first section is for the reader who is unfamiliar with the Smalltalk-80 system, and is a brief introduction to the object-oriented approach as embodied in the Smalltalk language. The body of the chapter contains a detailed description of the Smalltalk Virtual Machine.

Chapter 3 describes a high-level language implementation of the Smalltalk Virtual Machine in C on the PERQ. Implementation problems are described, the performance of the resulting system is assessed, and enhancements which would improve the performance are outlined.

A possible microcoded implementation is examined in chapter 4, and the problems of implementing Smalltalk on the PERQ in this way are compared with those on the *Dorado*. The performance of the proposed implementation is estimated.

In the final chapter, the conclusions of the earlier chapters are drawn together. Appendix A details some of the more interesting features of the implementation described in chapter 3, and discusses the utility of the Smalltalk Virtual Machine.

A language that doesn't affect the way you think about programming, is not worth knowing.

A. J. Perlis, Epigrams on Programming (1982)

2. The Smalltalk-80 language and Virtual Machine

Introduction to the Smalltalk-80 language

Before one can appreciate the design of the Virtual Machine, it is necessary to have a feel for the language it is designed to support. (It cannot be over-emphasised that the Virtual Machine was designed to support the language, and not *vice versa*. It is all too easy to lose sight of this simple truth when considering the finer points of the Virtual Machine's design.)

The Smalltalk language has been described fully elsewhere, and this section should serve as revision for those who have a basic knowledge of the language, rather than as a tutorial. Later we will see some of the more subtle implications of aspects of the language on the design of the Virtual Machine. Novices are advised to read any of the excellent introductions to the language (Smalltalk system Learning Research Group Byte [LRG81], GOLD83) before proceeding, whereas the *cognoscenti* can safely skip the rest of this section.

The most fundamental concept to the object-oriented approach Robson Object-oriented systems Byte [ROBS81] is that *information processing is initiated by objects sending messages*. An object is a package that contains data and has access to *methods* for the manipulation of the data. A message sent to an object is a request for that object to execute one of its methods and to return a result. For example, the expression 3+4 is construed as follows: the message whose name is + is sent to the object whose name is 3, with an argument which is a reference to the object 4; the object whose name is 3 then invokes the method it associates with the *selector* + and returns a result, which happens to be a reference to the object 7. Note that which method is to be executed is determined by the receiver of the message, and not the sender, or the argument(s). As a counter-example, the result of the expression 'a'+'b' might well be 'ab'.

The question arises, How does an object determine which method is invoked in response to particular selector? At this point, the concept of *class* comes into the picture. Every object is a member of exactly one class, and is said to be an *instance* of that class. All objects in a class respond to the same set of messages, and each time a particular message is sent to an instance of that class, the same method is executed, irrespective of the classes of any arguments. A class describes the similarities between objects—the messages that they understand, and their internal construction—while the instances describe the differences. Thus 3 and 4 are both instances of the class Number, and can respond to the messages +, −, etc., but the results of sending +5 to both objects are different.

The classes in the system are arranged in a tree-structured hierarchy, with class Object at the root of the tree. All the other classes are subclasses, or sub-subclasses, etc., of class Object. When a class is created, it *inherits* all the message responses from its superclass. New responses may be added to the class

later, to differentiate the behaviour of the class from its superclass. Each class maintains its own *message dictionary* that indexes methods by their selectors. When an object receives a message, it first searches the message dictionary of its class to see if there is a method associated with that message. If there is, that method is executed, in conjunction with any arguments; if there isn't, the search continues in the message dictionary of the object's superclass. If a method is not found thereat, the message dictionary of the object's superclass' superclass is searched. This hierarchical ascent continues until either a method is found, at which point it will be executed, or until no method is found in the root class, class Object. If the latter happens, the sender of the original message is in turn sent the message doesNotUnderstand.

This illustrates that it is not known in advance which method will be executed in response to a message: the method dictionary can change during the execution of a program. This property of Smalltalk, known as *late* or *dynamic binding*, is in contrast to conventional *statically bound* languages, such as Pascal, where it is known at compile-time which body of code will be executed in response to a procedure call.

So far we have seen that Smalltalk provides a message-sending capability, but what of other control structures? In essence, the basic language provides *no* control structures except message-sending and sequencing (the sending of more than one message, one at a time).* Fortunately, kernel classes are present in the system that define special messages for conditional branching and iteration.

To enable branching and iteration to be implemented in Smalltalk, the concept of a *block* is required. A block is like a method in that it contains code that can be evaluated by sending a message (possibly with arguments), but differs in that it is activated by directly receiving a message. In contrast, a method is activated by an intermediate object in response to a message. This can be illustrated as follows: Normal method activation takes place in two stages,

sender message receiver method lookup method

whereas block activation is performed directly with the special message value:

sender value message block.

To maintain uniformity in the system, blocks are treated identically to other objects. Once we have the fundamental concepts of messages and blocks, we can devise sophisticated control structures Deutsch Building Control Structures [DEUT81].

* Those with experience of the Virtual Machine may question the truth of this statement. It is certainly the case that in a real system iteration is not implemented in Smalltalk, but is assumed to be present in the Virtual Machine. However there is nothing to stop it being implemented purely in Smalltalk using recursive messaging. The main drawback (apart from inefficiency) would be that infinite loops would cause infinite stack expansion.

Now that we have control structures, what about data structures? An object may be composed of zero, one, or more *instance variables*, which are references to other objects. When an object is created, an *instance specification* in the object's class is consulted to see how many instance variables it should contain. In addition to the pre-determined number of instance variables that are enumerated in the object's class, some objects which belong to special classes can also have any number of *indexed instance variables*. Thus we can build arbitrarily complex data structures by composing objects to form larger objects Althoff Building data structures [ALTH81].

We have seen that in Smalltalk there are facilities for composing messages into control structures, and facilities for the composition of objects into larger objects. But where is the *real* work done, and where is the data *actually* stored? After all, we have not mentioned any basic arithmetic primitive functions, indicating that even + is sent as a message; similarly, we have stated that composite objects contain only references to other objects. This is where the interface between the Smalltalk language and the Virtual Machine lies KRAS81.

The Virtual Machine implements a number of *primitive methods* or *primitives*, which are invoked in the Smalltalk language by the use of a special keyword in the body of a method. These primitives perform operations such as integer and floating point arithmetic, manage the creation of, and access to objects, and serve as the interface to the outside world. In addition, there exist classes whose instances are composed of (lists of) integer values rather than references to other objects. The fields of these special objects can be manipulated by use of primitives dedicated to their access and modification.

Having said that an object is created by sending a message to its class, we can now say that this invariably leads to the invocation of a primitive. Additionally, we can state that there is no deallocation discipline as found in block-structured languages, where local variables cease to exist upon exit from their enclosing block. The temporary variables of a Smalltalk method, which disappear when the method returns, are only *references* to objects. This implies that objects are maintained in a global storage heap. But how are objects disposed of? A major asset of the Smalltalk system is that the programmer does not have to concern himself with the reclamation of storage that has been allocated to objects that are no longer required. The system automatically reclaims the storage a short time after the object falls into disuse. This eliminates the problem of "dangling references", which makes languages that require explicit deallocation inherently unsafe.

In the next section we shall see how the features provided in Smalltalk—namely message-sending and method lookup, the class hierarchy and inheritance, blocks, instance variables, primitives and automatic deallocation—have a bearing on the design of the Virtual Machine.

The Smalltalk-80 Virtual Machine

The Smalltalk-80 Virtual Machine consists of three major parts:

- The bytecode interpreter
- The object memory manager
- The primitive methods

Control within the Virtual Machine is managed by the bytecode interpreter, which fetches instructions and arranges accesses to memory and execution of primitives. The instruction set of the Virtual Machine consists of a set of *bytecodes*, which are interpreter commands, encoded in one, two or three bytes. Bytecodes are provided for:

- Stack manipulation
- Conditional and unconditional branching
- Message sends (including activation of primitives)
- Return from messages

When a method in the Smalltalk language is compiled, the compiler creates an object of class CompiledMethod, which contains a list of the bytecodes that represent the method. Should the method subsequently be activated by a message, the interpreter creates a *context* in which it can place the temporary variables required by the method, and begins execution of the method's bytecodes. A context is similar to an *activation record* or *stack frame* in a conventional block-structured language.

The object memory manager has three primary functions:

- It allows the interpreter to read the fields of an object,
- It allows the interpreter to change the fields of an object, and
- It is responsible for managing the allocation and deallocation of space for objects. Allocation takes place when a request is made by the interpreter for a new object, but the object memory manager must detect when an object is no longer referenced by any other objects, and deallocate that object automatically.

The primitive methods (or primitives) are a set of functions built into the Virtual Machine, which can be invoked from Smalltalk methods to perform operations that cannot be specified in Smalltalk, or that are too inefficient in Smalltalk. Some primitives are present purely to improve the performance of the system, but these are optional, and the implementor can choose not to implement them.

The object memory manager

We have already said that objects are composed of references to other objects, and that they are stored on a heap. It is possible to have references to objects in the form of memory addresses, but this is not done in practice. Instead, there is an *object table* that maintains every object's memory address, and references to objects are indexes into this table. Such an index is usually known as an *object pointer*. The advantage of this extra level of indirection is that the object memory manager only has to change the address in the object table if it moves an object around in memory. This is likely to be a frequent operation because objects come in a variety of sizes, and allocation and deallocation of objects of different sizes will sooner or later lead to fragmentation (more on this later).

An allocation request from the interpreter has to be satisfied in two parts: (1) a free object pointer has to be found, and (2) a free *chunk** of memory for the object has to be found. To do this, the object memory manager must maintain a free pointer list and a free chunk list. Since the majority of objects come in a small range of sizes, the recommended practice is to maintain a free chunk list for each of the commonly used sizes of object. This situation is illustrated in Fig. 1.

Note that every chunk, whether occupied by an object or not, has as its first word a *length* field, that gives its length, in words. Furthermore, chunks that contain valid objects have an object pointer to the object's class as the second word of the chunk. The instance variables of an object begin in the third word. This means that the smallest sensible size of chunk is two words, to contains objects that have no instance variables.

---

* The formal definition of the Virtual Machine uses this term to denote a block of memory.

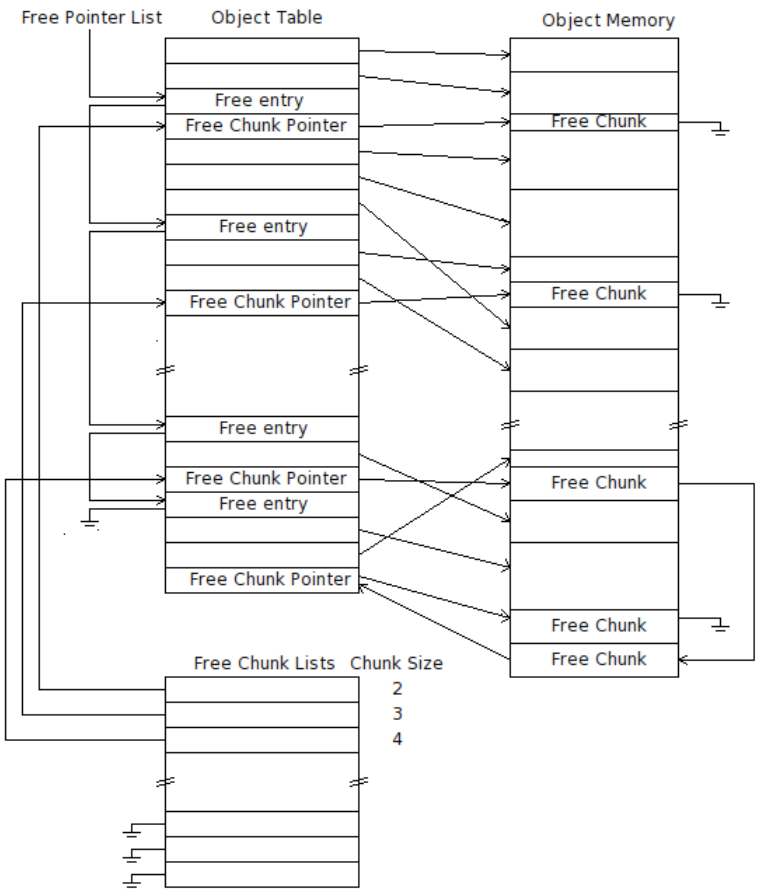

Figure 1. The structure of the object memory

There are three types of object that do not fit into this scheme. First, small integers are encoded into object pointers, by using a *tag bit*. This makes common arithmetic operations more efficient, and saves on storage for integers, at the expense of halving the number of possible non-integer objects. The second and third types are those objects that do not have object pointers in their fields. One type, the word objects, has word-sized integers stored directly in its fields, unencoded. (Note that a small integer pointer is an object

pointer, albeit without an associated explicitly stored object.) The other type has bytes packed into its fields, again unencoded. The object memory manager must provide an interface that allows the interpreter to access and modify the fields of pointer, word and byte objects, and also to determine the length and class of an object, given its object pointer.

The most difficult part of object memory management is the automatic reclamation of storage. In most implementations, a *reference count* is associated with every object. Whenever a new reference to that object is made, its reference count is incremented. Similarly, whenever a reference to an object is destroyed, its reference count is decremented. When the reference count of an object falls to zero, the storage allocated to that object can be reclaimed. This involves decrementing the reference counts of all objects it in turn refers to, which may involve further deallocation. Before a reference count overflows the storage allocated to it, it *sticks* at a maximum value, and never changes thereafter. An object with a stuck reference count cannot be deallocated by the reference counting mechanism. To do this a *mark-sweep garbage collector* is required.

Even though storage is usually automatically reclaimed by the reference counting mechanism, there are two reasons why an allocation request may not be satisfied, although enough free space exists for the size of the desired chunk. The first is due to fragmentation: allocation and deallocation of objects of varying sizes can lead to a situation where many small free chunks exist, though none of the chunks is large enough to satisfy the request. This can be cured by compaction, which involves moving objects around in memory to obtain a single, large contiguous area of free memory. Because the object memory manager uses an object table, this is a straightforward task.

The other situation in which a request can fail occurs when a number of unused objects form an isolated cycle. None of the reference counts can reach zero, and none of the objects in the cycle can be accessed. A mark-sweep garbage collector will also cure this problem. The first stage of a mark-sweep collector marks all objects that are still accessible by the interpreter. The second stage examines every chunk in the memory and sweeps the inaccessible chunks onto a free list. Reference counts are recomputed during the marking phase and compared with the stored reference counts; this allows inaccessible objects whose reference counts have stuck to be reclaimed.

The bytecode interpreter

The bytecode interpreter manages execution of bytecodes, performs process switching, and interfaces with the input devices (keyboard, pointing device, buttons, millisecond timer and real-time clock) and output devices (screen and cursor). There are four categories of bytecode:

The Virtual Machine deals with three kinds of variables (remember that a variable is an object pointer):

- Variables that are local to the receiver of the last message sent

- Variables that are local to the method currently being executed
- Global variables, which are accessed indirectly via a system dictionary

In addition, each method has a local pool of object pointers, known as the *literal frame*, which contains pointers to objects referred to in the Smalltalk source. Stack bytecodes are provided to push onto, pop off, and store from the stack, variables (local or global), pointers from the literal frame, and other frequently used pointers (to self, true, false, -1, 0, 1 and 2).

At this point, it is timely to explain method contexts. A method context is created when a message is sent, resulting in the execution of a non-primitive method. In an implementation of a conventional block-structured language, each procedure call causes an activation record to be created on top of a stack. This record contains all the local variables, and enough housekeeping information to enable non-local variables to be accessed and the record removed from the stack when the procedure returns. A single stack can be used for all the activation records because of the strict last-in-first-out discipline used for procedure calls and returns. However, it was decided in the design of the Smalltalk Virtual Machine that each activation record, or *method context*, would be an independent object. There are several good reasons for this: (1) Because contexts are instances of the class MethodContext they can respond to interrogatory messages, which makes it easy to write an interactive debugger in Smalltalk. (2) Due to the existence of blocks contexts, it is not the case that every method context can be deallocated when the method returns. (An example of this is given in the Appendix A.) (3) Allocation and deallocation of contexts can use the routines provided by the object memory manager.

A method context contains up to eight parts (see Fig. 2):

- A pointer to the context from which the message was sent that activated the current method (the *sender*)
- An offset into the current method, that points to the next bytecode to be executed (the *instruction pointer*)
- An offset into the context, that points to the top of the stack (the *stack pointer*)
- A pointer to the method associated with this context
- A pointer to the receiver of the message (sometimes referred to as self)
- A list of pointers to the arguments
- The temporary variables (or *temporaries*). The arguments and temporaries are collectively known as the *temporary frame*. The size of the temporary frame can be calculated at the time the method is compiled

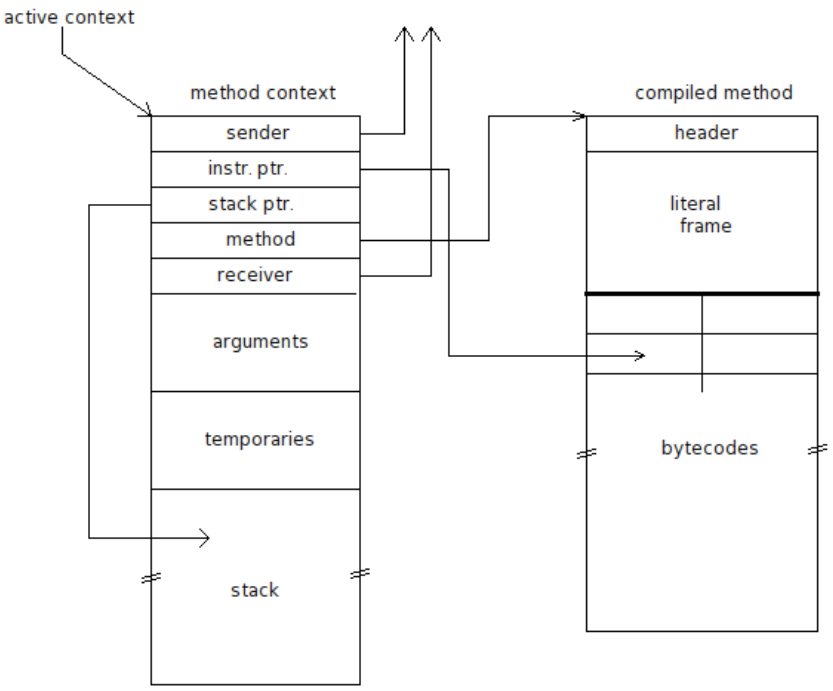

Figure 2. The structure of a method context

• The local stack. Obviously, the stack size varies during execution, but the maximum size can be calculated at compile-time (see also Appendix A).

Although we have said that iteration and branching can be implemented in the Smalltalk language by sending messages, for efficiency the designers of the Smalltalk-80 Virtual Machine decided to have the compiler translate these special forms of message into more conventional jump instructions. The unconditional jumps simply add an offset to the instruction pointer; the conditional jumps test for the presence of true or false on the top of the stack, and decide to jump or not on the outcome of the test. An error message, MustBeBoolean, is sent if neither true nor false is found on top of the stack.

Whenever a message is to be sent, a send bytecode is executed. Send bytecodes are analogous to call instructions on more conventional architectures, except that when a message is sent the location of the code

to be executed is found in a method dictionary. Therefore, a send bytecode also includes an indication of the message selector, and a count of the arguments in the message. When a send bytecode is executed, it is assumed that the object pointer of the object which is being sent the message, together with object pointers of the arguments, have already been pushed onto the stack. The execution sequence is:

1. Find the class where the message lookup is to begin. Typically this will be the class of the receiver, but Smalltalk also provides a super keyword that indicates that the search is to begin in the superclass of the receiver.
2. Look in the method dictionary attached to that class for the selector of the message. If the method dictionary does not contain the selector, move to the next higher superclass and repeat the search. If there is no superclass, i.e., we have looked in class Object, send the message doesNotUnderstand to the sender of the original message. DoesNotUnderstand is defined in class Object, and should always be understood.
3. Examine the *header* of the method that has been found. The header contains information about (i) the number of arguments the method takes, (ii) the number of temporaries it uses, (iii) the size of context it requires, and (iv) whether a primitive method should be executed. If a primitive index is found, execute the associated primitive. All primitives may *succeed* or *fail*. If the primitive succeeds, continue execution with the next bytecode; if it fails, or no primitive index is present, go to the next step.
4. Create and initialise a new method context, and begin execution of the bytecodes within the new method.

In addition to the above mechanism, there is a facility for the rapid execution of up to thirty-two frequently used primitives. If one of thirty-two *special send bytecodes* is encountered, an attempt is made to execute a primitive associated with each of the special bytecodes. If the primitive should fail, a special table is consulted, and a full message lookup performed. The table contains the selector and argument count associated with each special send bytecode. This mechanism enables the Virtual Machine to execute primitives without reference to a method header.

Whenever a method completes its execution, an object must be returned to the sender. Four bytecodes provide for the return of common object pointers (true, false, nil and self) and one returns the object pointer on the top of the stack. An additional bytecode returns the object pointer from the top of the stack in a block context. Activation of block contexts is discussed in the next section.

The primitive methods

The primitives can be divided into six categories:

- Arithmetic

- Subscript and stream access
- Storage management
- Control functions
- Input/Output functions
- System functions

The arithmetic primitives operate on small integers (encoded within object pointers), large integers (byte objects), and floating point numbers, and provide the usual functions of addition, subtraction, multiplication and division, together with a range of relational operators. All of the large integer primitives are optional.

The subscript primitives provide access to the indexed fields of pointer, word and byte objects, and also enable the length of the indexable part of an object to be found. The stream primitives (which are optional) provide efficient access to objects of class Stream (which are similar to sequential files).

The storage management primitives facilitate:

- Access to the fields of a CompiledMethod. (CompiledMethods differ from other objects in that they are contain a mixture of byte fields and pointer fields.)
- Creation of objects
- Access to the fixed fields (i.e., the non-indexed fields) of an object
- Conversion between small integer pointers and ordinary object pointers
- Enumeration of all the instances of a class

The more important control primitives allow block contexts to be created and executed. Whilst one can envisage an implementation in which a block was an independent object, the Virtual Machine does not treat blocks in this way. A block consists of a number of bytecodes embedded in a CompiledMethod. To execute a block, the compiler first has to invoke the blockCopy: primitive (which creates a block context), and then execute the block context when a value message is sent to the block. We will now describe these two operations in more detail (see Fig. 3).

A blockCopy: operation creates a block context linked to the current (or *active*) context. The block context uses the same method pointer as its enclosing method. A block context differs from a method context in that (1) the method pointer is replaced by an argument count which indicates how many arguments the block takes, (2) The receiver field is replaced by a pointer to the *home* context, where all the local variables of the textually enclosing method can be found, and (3) a new field, the *initial instruction pointer*, points to the first bytecode to be executed in the block.

When a value message is sent to the block context, it becomes the active context, and execution continues at the bytecode indicated by the initial instruction pointer. In becoming the active context,

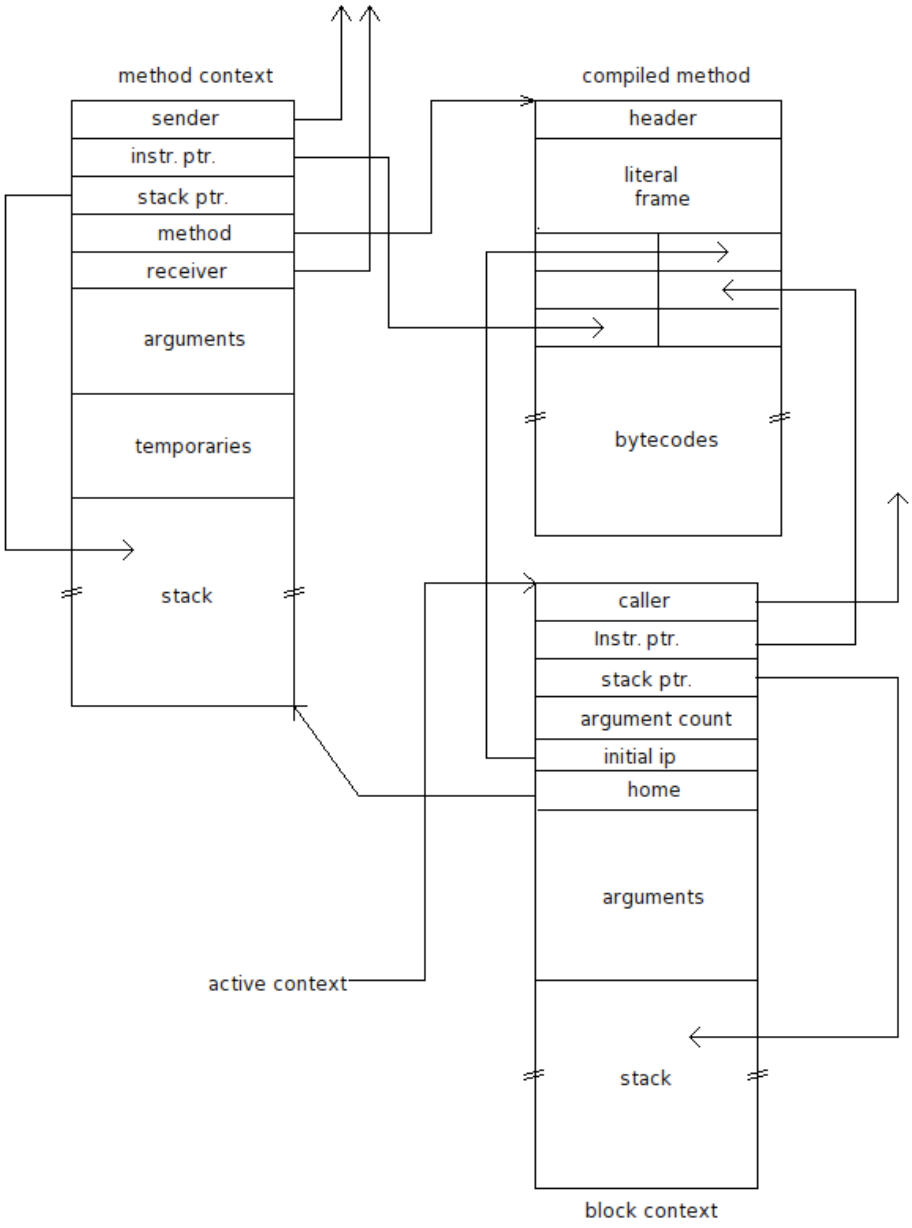

Figure 3. A block context in relation to its home context

(1) the *caller* field (which is what the sender field of a block context is known as) is set to point to the context from which value was sent, (2) the instruction pointer is loaded with the initial instruction pointer, (3) the stack is initialised with any block arguments, and (4) the home context is set to (*a*) the home context of the caller, if the caller is a block context, or (*b*) the caller, if the caller is a method context.

Other control primitives provide computed selectors in message sends, and semaphore-based multi-processing.

The input primitives allow the states of all the input devices to be communicated to a process, by signalling a semaphore whenever the state of a device changes. In addition, the values of the millisecond timer and the real-time clock can be polled. The principal output primitive invokes the raster operation BitBlt, and allows the screen, cursor and other bitmaps to be changed. Optional higher-level primitives provide for line drawing and text display. Also, a primitive exists for taking a snapshot (saving the current state of the Virtual Image).

The system primitives allow object pointers to be compared, the class of an object to be found, and various housekeeping functions to be performed.

The specification of the Virtual Machine

The Smalltalk-80 Virtual Machine has been defined operationally by Goldberg and Robson in Part Four of *Smalltalk-80: The Language and its Implementation* GOLD83. The definition takes the form of an interpreter and object memory for the Virtual Machine, written in the Smalltalk language. A 16-bit segmented real-memory architecture was chosen as the basis for the definition, and an object memory for this architecture defined in the Smalltalk language.

An instance of class Interpreter contains methods to emulate a bytecode interpreter and the primitives. It sends messages to an instance of RealObjectMemory, which emulates an object-based memory. RealObjectMemory in turn sends messages to an instance of RealWordMemory, which has an interface like that of a conventional segmented store. The protocol of RealWordMemory contains methods for fetching and storing bits, bytes and words in a memory with 16 or fewer segments of 64K words apiece.

The methods of RealObjectMemory translate messages from the instance of Interpreter into messages to the instance of RealWordMemory. *En route*, RealObjectMemory performs reference counting, compaction and mark-sweep garbage collection, handles the allocation of chunks, and provides access to the fields of objects. The object table is stored in one of the segments, together with the free pointer list. Each remaining segment has its own free chunk lists.

The instance of Interpreter performs bytecode execution, including the execution of primitives. The "registers" of the Virtual Machine are represented by instance variables of Interpreter.

The following parts of the Virtual Machine are specified in English, rather than in Smalltalk:

- Floating point primitives; however, all arithmetic is specified to be conformant with IEEE standard 754 (single precision) STEV81
- All input and output primitives (except BitBlt)
- The event buffer (see next chapter) and associated access routines

The Smalltalk code was written to demonstrate *what* an implementation of the Virtual Machine should do, rather than *how* it should do it. Because of this, no great effort was expended in making the code especially efficient, and an implementor should take care not to copy the code too literally.

Without sensibility no object would be given to us.

Immanuel Kant, Critique of Pure Reason, vol. III

The type object of objects that are type objects
is a special object of type "type".

J. K. Bennett, A Comparison of Four Object-Oriented Systems (1982)

3. An implementation of the Virtual Machine in a high-level language

Why use a high-level language?

It was said in the introduction that high-level language implementations of the Virtual Machine usually suffer from poor performance. Given this, it is necessary to explain why a high-level language was used for an initial implementation.

The alternatives to the use of a high-level language are assembly language and microcode. Information about the assembly language of the PERQ is proprietary, and is not available to the general user. Therefore the only other option was to use microcode PERQ: microcode [ICL82b].

Writing microcode is an incredibly laborious activity, compared to writing a program in a high-level language. Apart from the overwhelming morass of detail that the programmer has to deal with, little is available in the way of support software. Code must be written to handle floating point arithmetic, perform transfers to and from the disk, control the screen, and to read the keyboard and tablet. Debugging microcode, which has to be done via an "umbilical cord" to another machine, is especially difficult. Chapter 4 shows how an insubstantial fragment of high-level language expands into dozens of lines of intricate microcode.

There are other, more positive reasons for using a high-level language Wirfs-Brock Design decisions for implementors [WIRF83]: first, the formal definition of the Virtual Machine is written in a high-level language (Smalltalk), and transliterating from one high-level language to another is easier than translating from a high-level language to a very low-level language. Second, a program written in a high-level language is generally more portable than the same program written in a low-level language. Third, writing a "disposable" program in a high-level language enables one to learn about the Virtual Machine quickly, while at the same time obtaining a (hopefully usable) initial implementation, that can form the foundation for an efficient implementation. An intimate knowledge of the Virtual Machine is required before attempting to write a microcode version; this knowledge can be gained while building an initial implementation.

The *only* disadvantage in using a high-level language is poor performance. While it is usually the case that a large program, written in a system-software language, and compiled by a good optimising compiler, will be as efficient as an equivalent program in assembly language, this is unlikely to be so with the Virtual Machine. The run-time behaviour is such that a large proportion of the time is spent executing small portions of the code. These critical sections so dominate the execution speed that a careful choice of instructions will lead to a significantly better performance. Similarly, a shrewd allocation of registers will yield better results than that of most compilers, which can only make use of general allocation mechanisms. Naturally, when

one is able to design the instruction set of the computer to match the task at hand—which is the facility that microcoding provides—one can easily outperform an implementation in high-level or assembly language.

The programming environment

The operating system available on the PERQ was PNX, release 1.5, a derivative of UNIX system III. Most of the UNIX software tools were available, with the notable exceptions of an execution profiler and a debugger. The major extension to UNIX incorporated into PNX involves the use of the bit-mapped display. PNX includes a *window manager* which allows a number of *virtual terminals* to coexist on the screen, each in a separate window, and controlled by a separate UNIX process (see Fig. 4).

Windows are usually created by giving a command to the window manager, and any window can be brought to the "foreground" by another command. There exists a mechanism for associating the input devices (keyboard and four-button puck) with any visible window.

The high-level languages available were C and Pascal. C is a natural choice for an implementation which hopes to be portable across UNIX systems, and it also provides a rich set of operators, and good access to memory. Other C features deemed useful were:

- A macro preprocessor, which allows one to retain the call structure of the Smalltalk Virtual Machine, yet does not impose any run-time overheads
- Separate compilation, which saves on development time
- The ability to share a single declaration file among separate modules
- Easy conversion between data types

It is also easier to call UNIX/PNX system functions from C than from Pascal, because the principal language of UNIX is C.

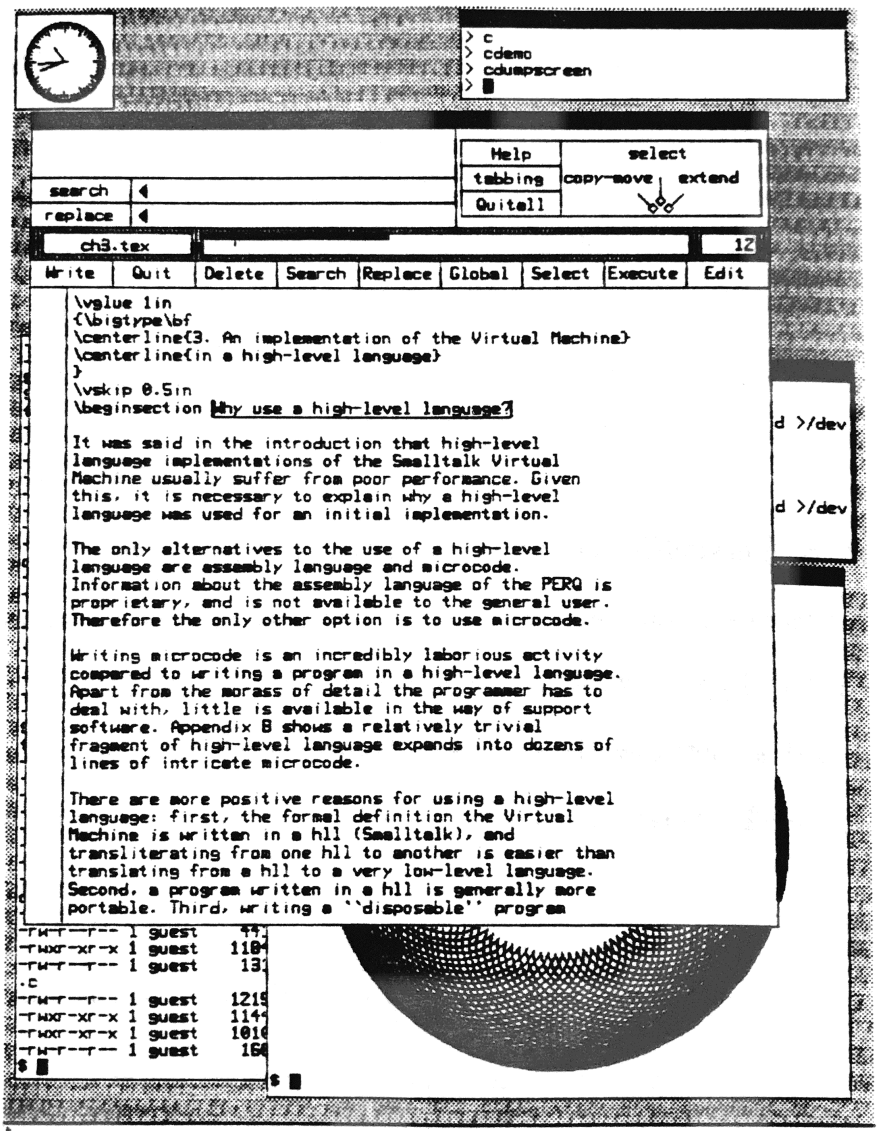

Figure 4. An example of the PERQ/PNX window manager in use

Controlling the hardware

Input and output are treated very differently in the Virtual Machine. Most input is *event-driven*, and the Virtual Machine maintains an *event buffer*. Whenever the state of an input device changes, a new event is placed in the buffer, and a pre-designated semaphore is signalled. The process waiting on this semaphore may then invoke a primitive to determine the nature of the event. There are four primary types of events:

- A change in the x-coordinate of the pointing device
- A change in the y-coordinate of the pointing device

- A bi-state switch coming on
- A bi-state switch going off

A bi-state switch is a key on the keyboard, a button on the puck, or one of five extra buttons (not present on the PERQ implementation). There were two small problems to be overcome with the keyboard. First, there is no mechanism in PNX for detecting the release of a key; the simple solution was to generate a key-release event immediately after a key-depression event. Also, access to the keyboard takes place in "raw" mode, i.e., with no filtering of characters by the operating system. This is required because the operating system would otherwise try to perform special actions (such as halting the program) whenever any special control key (such as control-C) was pressed, whereas the Virtual Machine requires all standard ASCII characters to be available. Furthermore, all keystrokes are to be echoed by the Smalltalk process that monitors the keyboard, rather than by the operating system.

Input from the pointing device was straightforward. However, it was disappointing to find that all input devices had to be polled, and that there was no interrupt mechanism available, based on UNIX *signals*.

A group of pointing-device or bi-state events must have a millisecond timestamp. The timestamp is recorded as another kind of event, and can take one of two forms. If the time interval between events is greater than 4095 ms, the timestamp must be in absolute time, measured since the last rollover of the millisecond clock, otherwise the interval itself is given as the timestamp.

Non-event-driven input routines must be provided to:

- Poll the coordinates of the pointing device
- Read the value of the real-time clock (which must be measured in seconds since the midnight previous to January 1, 1901)
- Read the value of the millisecond clock
- Designate a semaphore, and a value of the millisecond clock at which the semaphore is to be signalled

None of these proved troublesome. However, reading the value of the real-time and millisecond clocks has to be performed via the UNIX system calls *time* and *ftime*. Benchmarks showed that a single call to either of these took over 700 μs, a not insubstantial period considering that a call has to be made for every group of input events. To keep these overheads down to an acceptable level, it was decided to poll the input devices every 20 ms or so.

The only output device which has to be present is the display, and some mechanism for placing a cursor on the display. While accessing the cursor was straightforward, access to the 768×1024 portrait-format screen was the cause of many headaches. The Smalltalk-80 system requires that the display be an instance of class DisplayScreen, and consequently reside in the object memory. Furthermore, a primitive must be

provided that takes an instance of DisplayScreen as an argument, and thereafter uses it as the display image. The screen should then reflect any changes in the Bitmap that is a component of the DisplayScreen.

Unfortunately, although the PERQ display is refreshed from a bitmap in main memory, direct access to this bitmap (via a C pointer) is not possible under PNX, due to the restrictions placed on the virtual address space of each process. The only technique available for changing the display is to use the provided system call *wrasop*, which performs a raster operation between bitmaps, similar to BitBlt, but not as general. Operations can be performed between bitmaps in the process store, or between store and screen, or between screen bitmaps. Use of a display bitmap (the whole screen, or a window) is indicated by passing a null pointer as the base address of the bitmap to the *wrasop* procedure.

However, BitBlt cannot be implemented using *wrasop*, because *wrasop* requires bitmaps to be a multiple of 64 bits wide, and aligned on a quadword boundary (words are 16 bits wide). It is impossible to guarantee this for the bitmaps in the object memory, as they can be any multiple of 16 bits wide, and begin at any word boundary. Additionally, *wrasop* is not very general: it provides only eight of the sixteen possible combination rules between source and destination bitmaps, and does not provide the halftoning capability of BitBlt. The solution chosen was to implement BitBlt in C, by translating from the Smalltalk specification, and to refresh the whole screen as many times a second as was considered prudent. The refresh rate is a balance between friendliness and efficiency: too few refreshes a second make the screen display look jerky, while every refresh decreases the processor time available for other computations. Since it was found that a whole-screen refresh could be done (using *wrasop*) in about 25 ms, it was decided to refresh the screen about 10 times every second. A 25% loss of performance seemed acceptable for an initial implementation.

To enable a refresh to be done using *wrasop* (which is much faster than any other method because of the PERQ's hardware support for raster operations), it was necessary to ensure that the DisplayScreen was of a suitable width and location. This was done by allocating a chunk of object memory big enough for the display, prior to loading a snapshot. This ensured that the display bitmap was in a guaranteed location in memory.

Finally, a decision had to be made whether or not to refresh the screen via the window manager. The alternatives were to restrict the display to lie within a PNX window, or to use the whole screen, bypassing the window manager

The benefit of using the window manager is that the Smalltalk process is just like any other process in the system, and therefore one can easily move between Smalltalk and other tasks. The Smalltalk window can be moved around on the screen, obscured or made visible, and easily halted in the case of an error arising (this is especially important, because the keyboard is in "raw" mode, and the usual halt signal, control-

C, does not work). The disadvantage is that each refresh of the display takes twice as long—the window manager maintains an off-screen copy of the window and refreshes the *whole* window using the off-screen copy whenever a visible part is changed—and uses extra memory. For initial testing, it was decided to go via the window manager; happily it is simple to change this.

The PNX operating system which was available for the PERQ (release 1.5) can only access 1 megabyte of primary memory. While at first it might be thought that this would be sufficient (since the Smalltalk Virtual Image requires 600–750 Kb of memory), one has to bear in mind that the object table occupies 128 Kb, the PNX operating system requires approximately 200 Kb of memory, and that the display bitmap takes 96 Kb. When one adds this to the 100 Kb required by the C implementation of the Virtual Machine, one can see that more than a megabyte is required in total. Since PNX implements a demand-paged virtual memory system, one might also think that this would not be a problem, so long as the working set of the system was below the 1 megabyte limit. Unfortunately, PNX 1.5 has a page size of 128 Kb, which means that any working set which requires eleven or more different pages will cause thrashing, resulting in a severe degradation of performance. Fortunately, the second version of PNX (which requires a hardware upgrade to the PERQ used for the initial implementation) has a main memory threshold of 2 Mb, which is more than adequate.

A minor problem also occurs in the byte-addressing scheme of the PERQ when considering the object memory. The standard Virtual Image has its bytes packed into 16-bit words, with the more significant eight bits representing the "lower" byte in memory. However, the PERQ has the more significant eight bits of a word as the "higher" byte. The addressing transformation devised to convert to a byte offset in object memory was simply to invert the least significant bit of the address, using an exclusive-or operation. (This assumes that byte objects begin on a word boundary.)

Writing the program

As mentioned in Chapter Two, the formal specification of the Virtual Machine is a program written in Smalltalk GOLD83. Although this program is not particularly efficient, on the whole it is a rigorous specification, and a suitable base for an initial implementation. The first step in implementing the Virtual Machine was to gain a thorough understanding of the formal specification. To aid in this, the whole of the Smalltalk code of the Virtual Machine was entered into a set of text files. Then, a UNIX shell script was constructed to generate an index, which showed where methods were called and instance variables used; this was helpful when simplifying the code. Another use of the Smalltalk methods was to include them in the final program as comments, illustrating the purpose of the C code by juxtaposing it with the Smalltalk code.

The second stage involved deciding how closely the implementation should follow the form of the specification. Most of the decisions were made on the basis that memory space should be exchanged for speed. Thus it was decided to implement many of the object memory routines as macros, which saved on function call overheads, while at the same time preserving the modularity afforded by the call structure. A major simplification was to abandon the segmentation in the definition of RealObjectMemory, and store the whole of the object memory in a single large array. The object table was placed in a separate array, and the free chunk list heads were also given their own array.

A number of micro-level benchmarks were written in C to determine the relative costs of using different control structures. As a result of this, a number of local optimisations were made in an attempt to speed up the code.

One such optimisation was to replace large multi-way branches with dispatch tables, giving the address of C functions. For example, the Smalltalk code for dispatching arithmetic primitives was translated from:

```
dispatchIntegerPrimitives
    primitiveIndex = 1 ifTrue: [↑self primitiveAdd].
    primitiveIndex = 2 ifTrue: [↑self primitiveSubtract].
    primitiveIndex = 3 ifTrue: [↑self primitiveLessThan].
```

etc., to the following C code:

```
BOOL (*dispPrim[])() = {
    primFail,    /* no primitive 0 */
    prAddSI, prSubSI, prLTSI, ...

...(*dispPrim[primIndex])()...
```

Another optimisation was to change the functions which emulated the primitives so that they returned

a boolean value indicating whether a primitive succeeded or failed. In contrast, the Smalltalk code maintains a variable, success, which is repeatedly tested to determine the outcome of a primitive. For example, the Smalltalk code for primitiveAdd is

```
primitiveAdd
    | integerReceiver integerArgument integerResult |
    integerArgument ← self popInteger.
    integerReceiver ← self popInteger.
    self success
        ifTrue: [integerResult ← integerReceiver + integerArgument.
                 self success: (memory isIntegerValue: integerResult)].
    self success
        ifTrue: [self pushInteger: integerResult].
        ifFalse: [self unPop: 2]
```

whereas the C equivalent uses control structures rather than a success variable:

```
BOOL prAddSI()     /* add SmallIntegers */
{
    SIGNED intArg, intRcvr;
    int intRes;
    OOP intPtr = popStack;

    if (isInt(intPtr)) {
        intArg = intVal(intPtr);
        intPtr = popStack;
        if (isInt(intPtr)) {
            intRcvr = intVal(intPtr);
            intRes = intRcvr + intArg;
            if (isIntVal(intRes)) {
                push(intObj(intRes));
                return FALSE; /* didn't fail */
            }
        }
        unPop(2);
    } else
        unPop(1);
    return TRUE; /* failed */
}
```

In total, the program constitutes about 4500 lines of C, structured as 15 modules and 12 declaration files, and took about five weeks to write. Of this, nearly two weeks were spent in implementing and testing the input and output modules, including BitBlt. Because these modules had to be written from scratch (except the routine for BitBlt), it was felt that obtaining functional code for these operations should be done first of all, since their implementation was the most uncertain aspect of the whole effort. Fortunately, the floating point arithmetic on the PERQ is to IEEE standards Stevenson proposed standard floating-point IEEE [STEV81]; this made implementation of the floating point primitives straightforward.

On the whole, writing the program was a pleasant, painless task. This success can be attributed to three factors:

- A concerted and deliberate effort was made to understand the Smalltalk code for the Virtual Machine, and to anticipate any serious difficulties *before* coding began.
- The specification of the Virtual Machine is well-written, well-documented, and without any serious flaws. In general it is easy to understand the intent behind the code, so much so that minor typographical and coding errors soon make themselves apparent.
- The programming environment, a multi-window version of UNIX on the PERQ, was most conducive to high productivity. The PERQ screen is able to display 66 lines of text, which enables one to edit a program in one window, compile it an another, and run it in a third, while having any of the three visible at the press of a button. The UNIX tools were used very frequently, and the C compiler proved to be most dependable. The Smalltalk code was entered onto a DEC *VAX*-11/750 (running UNIX 4.1 BSD), and manipulated there using programs written entirely in the UNIX command language. The code was later copied to the PERQ, and merged into the C source code using more UNIX tools, and the PNX puck-driven editor *spy*.

Testing the program

The only satisfactory test of an implementation of the Virtual Machine is whether or not it will run the Virtual Image. Unfortunately, the Virtual Image was not available, so initial tests had to be devised using some other scheme. Testing a Virtual Machine is very much like commissioning a new processor; one has to methodically construct a number of test images, each building on the results of the last. Because of the complex interaction between the elements of a test image, the first tests must be very simple if one is not to flounder when an error occurs. However, one can also be reasonably certain that when one has tested a particular feature thoroughly, it is not likely to cause any future problems, because of the limited interaction between elements of the Virtual Machine. Hence, although there a large number of bytecodes and primitives to be tested, many can be certified correct independently of the operation of others. Furthermore, because most of the code makes repeated use of a few macros and routines (especially those for accessing contexts and the object memory), once these are operational the remaining parts of the program are likely to contain few difficult-to-find bugs.

As was said earlier, the input and output modules were tested thoroughly before work began on the rest of the Virtual Machine, so that it was safe to assume that there were no problems within these modules. The second step, after the rest of the program was written and successfully compiled, was to construct a minimal image. While writing the code, tracing and debugging statements were placed at strategic points, which could be enabled by compile-time switches (C makes this very easy). For initial testing, all such code was enabled. Furthermore, a simple single-step facility was included in the main bytecode dispatch loop to enable one to inspect the contents of any object, or the state of the "registers" of the Virtual Machine, between execution of bytecodes.

A simple image format was devised, which could be edited and inspected using a standard text editor, and a minimal image was constructed. The minimal image (see Fig. 5) consists of twelve objects; these constitute a single process, with an active context, a receiver and its class, and a single method which calls the quit primitive. The next step was to add a call to the snapshot primitive, testing that images could be saved as well as loaded.

Once these two essential primitives were operational, a more sophisticated test could be devised. The test included at least one bytecode from each of the four major categories, and the invocation of several arithmetic primitives. The test consisted of sending a message which invoked a method to calculate the factorial of a small integer, 5. The factorial method used an iterative algorithm, which involved executing both conditional and unconditional branches. The image is shown in Fig. 6. After each bytecode was executed, the object memory was inspected to check that all the required operations were being performed

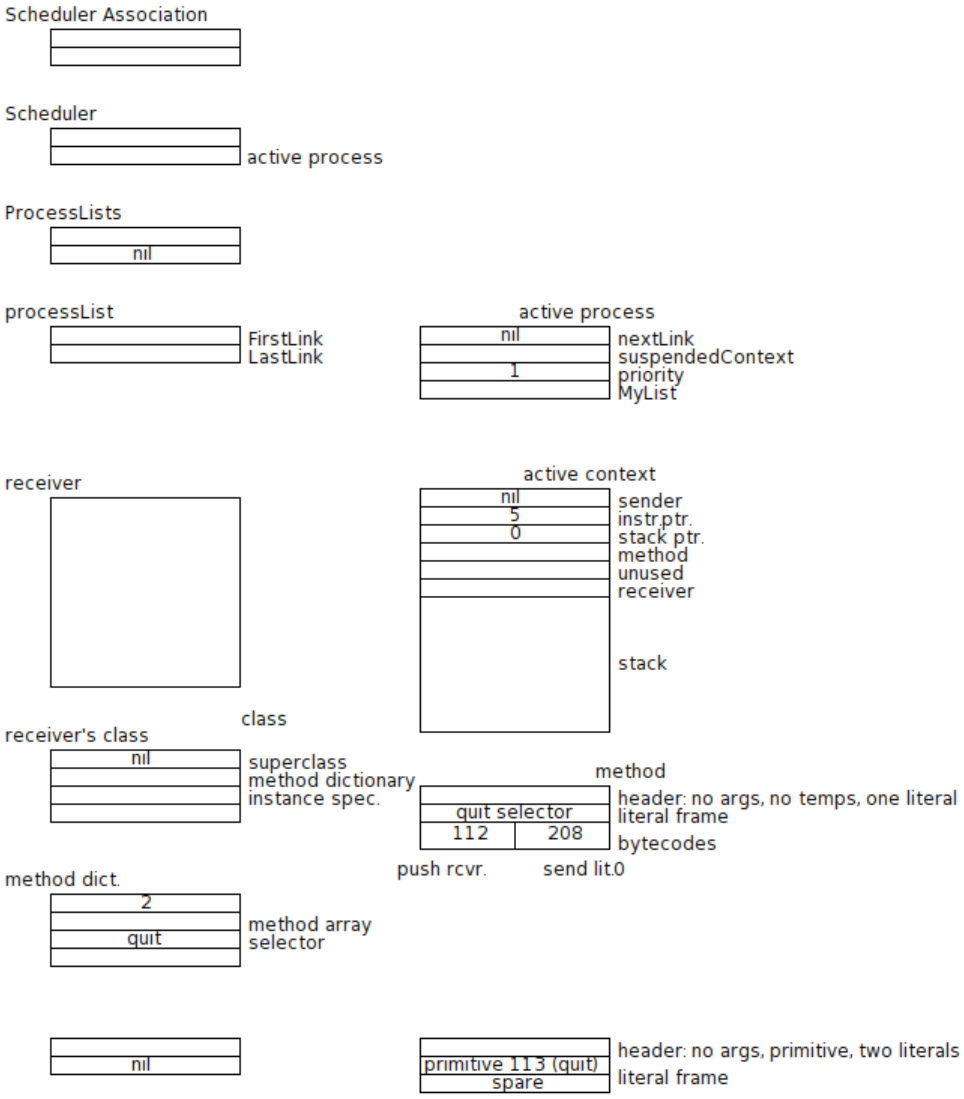

Figure 5. The minimal virtual image

correctly. Additionally, the image that was saved as a result of the snapshot primitive was carefully inspected to check that reference counts and free lists were correct.

Having successfully compiled the program, these simple tests took about two days to complete successfully, mainly because of the difficulty in creating consistent images. Machine code programming in Smalltalk is not easy!

Performance of the implementation

The Virtual Machine requires 100 Kbytes of store, in addition to the store allocated to the object table and object memory. The excessive code size is due to the extensive use of macros. (There was no chance of fitting *any* implementation into 1 Mb, so that saving an extra few tens of kilobytes seemed unprofitable.)

To gain an estimate of final performance (assuming that enough store became available), the test image which evaluated 5 factorial was altered to send the same message 1000 times. A total of 74009 bytecodes were executed, made up of 46005 (62.2%) stack bytecodes, 11001 (14.9%) jump bytecodes, 1000 (1.35%) full message send bytecodes, 15001 (20.3%) special send bytecodes, 1000 (1.35%) return bytecodes, and 2 send bytecodes which invoked the snapshot and quit primitives. This benchmark (without the tracing and debugging code) was timed at 29.4 seconds, which represents an average speed of 2500 bytecodes per second. A better test would have had a slightly different mix of bytecodes (Ungar Patterson Berkeley Smalltalk [UNGA83], Falcone Analysis at Hewlett-Packard [FALC83b]), with more full message sends. Due to the fact that full message sends are slow, a more realistic performance estimate would be around 2000 bytecodes every second. While in absolute terms this represents a slow implementation, it proves that a workable implementation is feasible. Furthermore, there are many potential speed-ups available which were not incorporated. Most of the speed gains that were made in the current implementation were due to good use of the implementation language, rather than algorithmic changes. A list of potential optimisations is given at the end of this chapter.

As an exercise, the program was ported to a DEC *VAX*-11/750, to gain some idea of the performance of the code with respect to other, earlier implementations on similar machines. Because there were no suitable input and output devices present, the calls to input and output routines became null operations. Once this was done, the program compiled and ran without problems. The same benchmark ran on an otherwise unloaded machine (i.e., there were no other users) in 24.2 seconds, approximately 20% faster than on the PERQ. Re-running the C benchmarks on the *VAX* showed a similar performance gain, except when floating point arithmetic was involved, for which the *VAX* was much faster. The port was very simple, and the factorial benchmark was operational on the day after the code was copied to the *VAX*.

How does the performance compare with other first implementations? In speed terms, it is very promis-

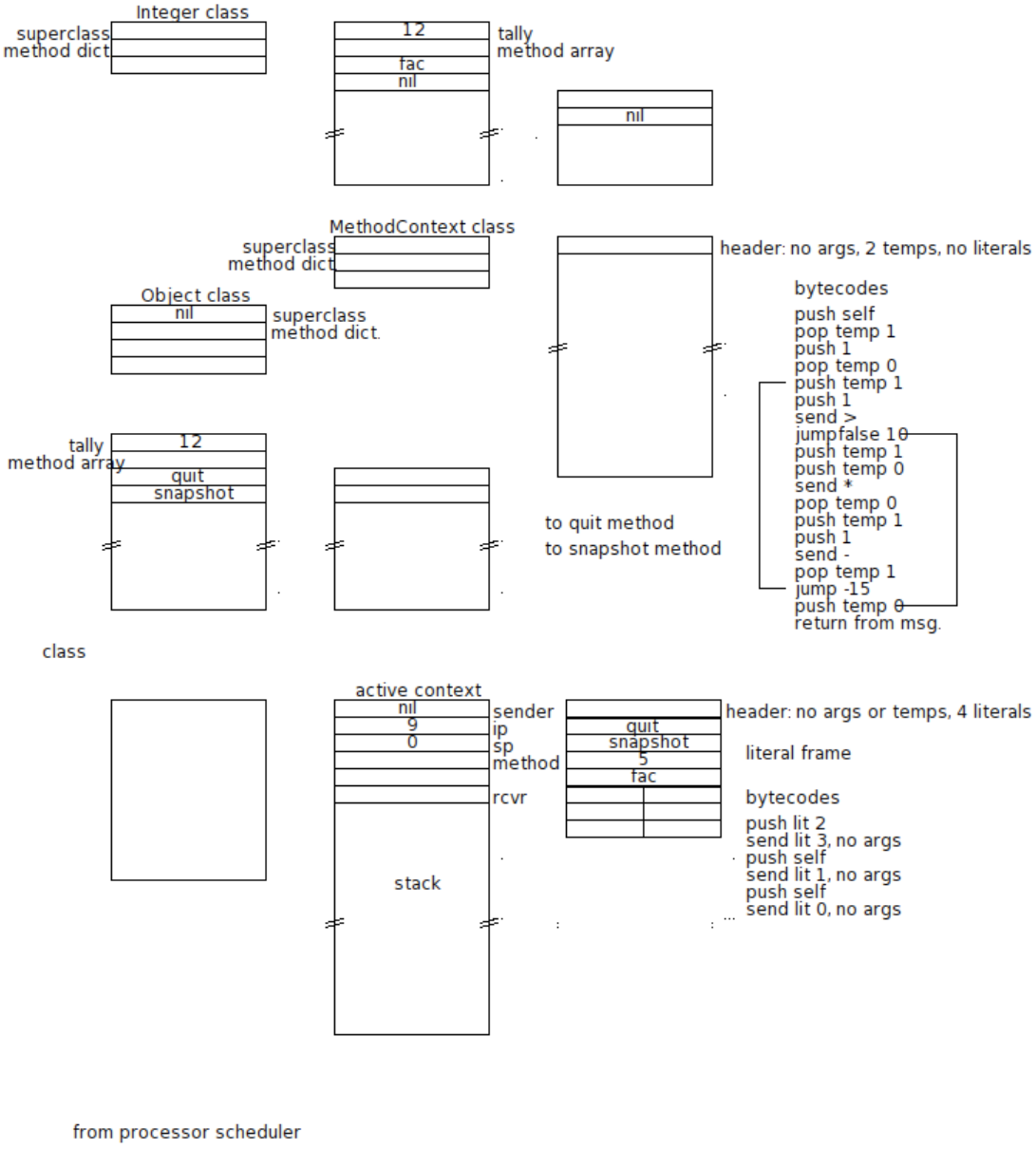

Figure 6. A portion of the virtual image that executes factorial 5

(See Fig. 5 for those objects not shown.) The Smalltalk program is:

```
factorial
| f count |
count ← self.  f ← 1.
[count>1] whileTrue:[f ← f * count.  count ← count−1].
↑f
```

ing: An initial implementation by Hewlett-Packard Falcone Stinger Smalltalk Hewlett-Packard [FALC83a] on a *VAX*-11/780 in C under UNIX 4.1 BSD ran at 1000 bytecodes per second, and was improved after several months effort to 5000 bytecodes per second. Following the comments of Ungar and Patterson UNGA83, one can estimate that the initial implementation can be speeded up to about 10000 bytecodes per second by the use of algorithmic optimisations. This is broadly in line with the DEC implementation Ballard Shirron Design Implementation VAX/Smalltalk-80 [BALL83] on a *VAX*-11/780 (under VMS), at 15000–25000 bytecodes per second, and the Berkeley implementation (also on a *VAX*-11/780) at 12000 bytecodes per second.

While a four-fold performance improvement is thought possible UNGA83, the resulting performance will be still lack-lustre, when compared to the implementation on the *Dorado* Deutsch Dorado Smalltalk [DEUT83] at an amazing 400000 bytecodes per second. In the final section of this chapter we will outline how a four-fold improvement in performance can be achieved, and in the next chapter we will show that a completely different approach will yield a five-fold improvement over the best high-level language implementation on the PERQ.

Optimisations

In implementing the Virtual Machine, other people have devised and discovered many useful tricks and changes which improve performance. These are collected and summarised below. The changes required in the current implementation to accommodate these improvements range from the trivial to the extreme; from alterations that can be done in an afternoon, to many weeks work. The first three are of the trivial sort, and have already been implemented:

- In the formal specification, the maximum chunk size to have its own separate free list is 20 words. By increasing this limit to 40 words, chunks for large contexts are moved from a generic free list to one of a specific size. Since contexts account for most of the allocation requests, this speeds up message sends.
- In the Smalltalk code, several instances of SmallInteger (including stack and instruction pointers) are fetched and stored using the routines fetchInteger:ofObject: and storeInteger:ofObject:withValue:, which check that the object pointers they are handling are indeed SmallIntegers. These tests are unnecessary, and the use of smaller routines can save execution time when manipulating important integers, such as the stack and instruction pointers.
- When transferring the arguments from one context to another in a message send, the Smalltalk code uses general routines which perform reference counting. Since the arguments are moved and not copied, their reference counts do not change, and the extra code can be avoided.

Here are other conceptually simple changes:

- The code for bytecodes which push non-reference-counted objects (true, false, nil, etc.) can be changed so that it does not check whether the objects should have their reference counts increased.
- Special free lists can be maintained for the exclusive use of contexts and Points (these account for 98% of allocation requests FALC83a, pp. 95–6).
- To speed access to the active context, the instruction pointer, stack pointer and active context pointer can be cached as absolute addresses rather than offsets and object pointers. Additionally, C structure casts can be used for fast access to fields of contexts, compiled methods, etc. FALC83a, p. 82
- The tests performed in the main bytecode dispatch loop to determine whether the input devices should be polled, or a process switch checked for, can be condensed into a single decrement and test UNGA83, p. 196.
- The code for BitBlt can be improved in two ways: first special cases (such as area fill) can be detected quite easily, and result in the execution of special-purpose fast code; second, the main loop of BitBlt consists of a multi-way branch inside a `for` loop. The same alternative in the branch is chosen every time, within a single call to BitBlt. By commuting the branch and the loop (i.e., having a branch which selected one of 16 special-purpose loops), the primitive would be much quicker.
- The implementation of the method cache follows the formal specification quite closely, differing only in the choice of the hash function. Many better (but more complicated) schemes are available for the method cache FALC83a, p. 87; BALL83, pp. 147–8; UNGA83, p. 202.

There are a number of optimisations which concern the object table, and speed up object access at the expense of more memory;

- The size and class fields of an object can be moved from the object's chunk to the object table; this saves an indirection when accessing these fields.
- Expanding reference counts to 16 or 32 bits obviates the need for overflow checks UNGA83, p. 192.
- Splitting the components of the object table into separate arrays speeds indexing if the sizes of the elements of the arrays are powers of two UNGA83, p. 192.
- If the most significant bit of an object pointer becomes the SmallInteger tag bit, a single signed comparison can discriminate between ordinary and non-reference-counted objects (SmallIntegers and nil, false, true, etc.) UNGA83, p. 192.

The remaining optimisations try to save on reference counting overheads:

- The majority of contexts are deallocated when they return; only a few do not. If one can detect

when a context cannot be deallocated on return, the majority of context reclamation can be speeded up by not having to explicitly test reference counts. Fortunately, there is a way of doing this: the only occasions on which a reference is created to the active context (except during a message send) are when the pushActiveContext bytecode is executed, or when the context is saved on a process or semaphore queue. If neither of these occur during the lifetime of a context, it can be reclaimed immediately upon return, without having to test the reference count BALL83, p. 146.

• The number of reference count operations for full sends and returns can be cut six-fold by using a different stack management strategy. One can avoid sweeping the area above the top-of-stack when a context is reclaimed, by banishing references to objects from that area. Also, the return value can be moved, rather than copied, when a method returns; this saves another reference count operation UNGA83, pp. 193–5.

In all, these optimisations can lead to a 400–500% increase is speed UNGA83, p. 190.

Plan to throw one away; you will anyhow.

F. P. Brooks, Jr., The Mythical Man-Month (1975)

4. Implementing the Virtual Machine in microcode

Chapter 3 described a high-level language implementation of the Virtual Machine, stated its present performance, and estimated its potential performance. At best, the speed of a high-level language implementation on the PERQ is unlikely to surpass 15000 bytecodes per second. When one considers that microcoded implementations have done much better than this mccall Smalltalk-80 benchmarks [MCCA83], it seems worthwhile to investigate the problems and benefits of microcoding the Virtual Machine on the PERQ. The only other microcoded implementation which has been described in the literature in detail is that done on the Xerox *Dorado* DEUT83. Coincidentally, this implementation also happens to be the fastest, executing on average 400000 bytecodes every second. This chapter will compare the *Dorado* implementation with a possible PERQ implementation, and describe the problems involved in microcoding the PERQ. We will also attempt to estimate the speed of such an implementation.

Architecture of the *Dorado*

To enable the two implementations to be compared, one must consider the underlying hardware in some detail. A program in microcode communicates with the hardware at the lowest possible level, and therefore is intimately tied to the hardware to a degree that is not possible when written in assembly code or high-level language.

The *Dorado* is a high-performance personal computer, designed and built at Xerox *PARC*. It has been described in detail in Lampson processor high-performance personal computer [LAMP80], Lampson instruction fetch unit [LAMP81], Clark memory system [CLAR81] and Pier Dorado retrospective [PIER83]. The salient features of the *Dorado* are:

- It is a pipelined microprogrammable machine, with a 60 ns microcycle
- It has a fast pipelined main memory cache, which has a low latency and high throughput, and usually achieves high hit rates
- Because its design was a follow-on from the Xerox *Alto*, it supports an *emulation mode* in which it can execute *Alto* object code
- Instructions are fetched by a fast, semi-autonomous unit, which facilitates high-speed instruction decoding
- It has a large main memory, with hardware support for paged virtual memory
- Input and output is performed by *microtasks* which are written in microcode and timeshare the CPU with the bytecode interpreter

The features which affect the implementation of the Smalltalk Virtual Machine will be described in more detail:

The *Dorado* processor LAMP80 is constructed using ECL technology, and has a 60 ns microcycle time. The microinstructions are 36 bits wide (34 data bits and 2 parity bits), and the processor has a 4 Kword microinstruction store. Each microinstruction is very tightly encoded, and for implementations of most languages (but not Smalltalk) a macroinstruction is frequently executed by a single microinstruction. This indicates that each microinstruction can be quite powerful. The processor has a throughput of one microinstruction per microcycle, using a three-stage pipeline. Contained within the processor are:

- 256 general purpose 16-bit registers
- 4 hardware stacks, each 64 levels deep
- A 32-bit barrel shifter/masker, and
- A 16-bit ALU, with symmetric data inputs

A major feature of the *Dorado* is its IFU LAMP81. The IFU is able to fetch bytecodes, decode them into instructions and operands, and provide the processor with suitably formatted data, including the microinstruction dispatch address. In the absence of conditional branches it performs bytecode fetches autonomously, simultaneously maintaining the bytecode instruction pointer; the processor only has to intervene when a conditional branch is not taken. The decoding is performed via table look-up in a 1024-entry RAM; this allows four separate instruction sets to coexist, with high-speed switches between them performed by simply changing the value of a register.

The *Dorado* memory system CLAR81 supports a paged virtual address space with 28-bit virtual addresses. Addresses contain a reference to one of 32 *base registers*, and a 16- or 28-bit offset. A real address is computed by adding the offset to the base. The main store consists of cheap, but slow, MOS dynamic RAM. To make up for the lack of speed of the main store, there is a large (8–32 Kbyte) fast cache, which can deliver a memory word every microcycle. The cache operates on the *write-back* principle rather than the *write-through* principle: this means that an entry in the cache that has been altered (a *dirty* entry) is only written to the main store when the cache entry is flushed, so that another entry may take its place. In contrast, a write-through cache writes the cache entry to store every time it changes. The write-back mechanism saves considerably on store requests. Typical hit rates for the *Dorado* cache are above 99%; when a cache miss occurs, a 16-word block is fetched from the main store and placed in the cache.

The *Dorado* follows the *Alto* (Thacker Alto personal computer [THAC79], Wadlow Xerox Alto Byte [WADL81]) in performing input and output in microcode, by switching at high speed between a set of fixed-priority tasks. The *Dorado* has 16 microtasks, with the bytecode interpreter

always having the lowest priority (because it does not have a crisis time). Task switching does not impose any time overhead, because each task has a separate set of task-specific registers, which are addressed by a single *task register* containing the current task number. Thus task switching can take place at every microcycle, by changing the value of the task register. The peripherals attached to a *Dorado* usually comprise a keyboard, mouse, high-resolution bit-mapped display, Ethernet interface, and 80 Mb disk.

Architecture of the PERQ

Like the *Dorado*, the PERQ is a microcode engine ICL82b. It has a basic cycle time of 170 ns, almost three times slower than that of the *Dorado*. There are also several important features of the *Dorado* which are absent from the PERQ. The data paths to and from store are 16 bits wide, but most of the internal processor data paths are 20 bits wide, to enable efficient address calculation (see Fig. 7). The PERQ is limited to 1 Mword (2 Mbytes) of main memory.

The microinstructions are 48 bits wide; early PERQs were limited to 4 Kwords of control store, but modern ones have 16 Kwords. When not accessing main memory, the PERQ usually executes one microinstruction per microcycle. Contained within the processor are:

- 256 general purpose registers, each 20 bits wide; the register file is dual-ported
- A 16-level hardware stack (also 20 bits wide)
- A 16-bit shifter and masker
- A 20-bit ALU with asymmetric inputs
- An 8-byte file for bytecodes

Instruction fetch and decoding must be done explicitly in microcode; there is no autonomous IFU. Also, there are no independent microtasks: input and output drivers are activated as the result of microcode-level interrupts, and the necessary context changes must be performed explicitly in microcode.

Like the *Dorado*, the PERQ has a slow memory system built from MOS RAM. Unlike the *Dorado*, there is no cache to compensate for this. The memory system can *at best* accept one request every four microcycles, provided that the request arrives at the correct time. Each store cycle is composed of four microcycles, and requests are executed immediately only if they arrive in the correct minor cycle; any requests issued at other times are held up until the appropriate cycle. Additionally, one cannot start two store requests in successive cycles: an explicit pause must be placed between them. If this is not done, the second request is ignored. This is in contrast to the memory system of the *Dorado*, which can accept a memory request (fetch or store) in every microcycle.

The store is 64 bits wide; a memory request can be for one, two, three or four 16-bit words, provided

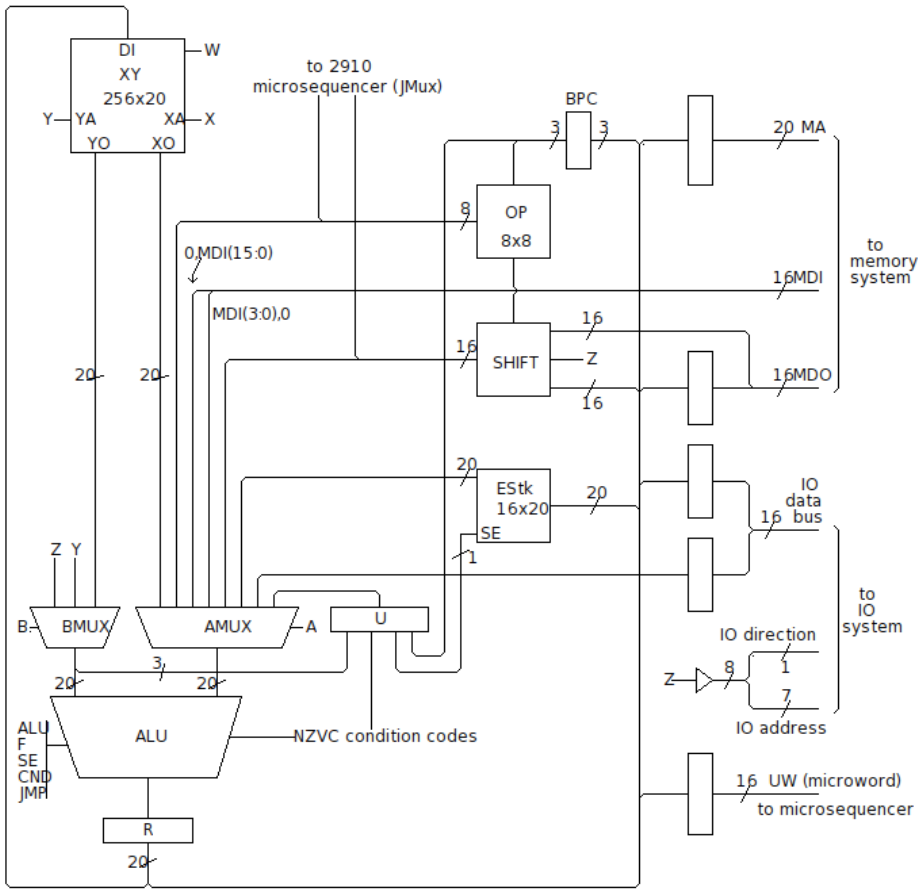

Figure 7. The architecture of the PERQ processor

that they all come from the same 64-bit memory word. When the data has arrived from main memory (which takes three microcycles), one microcycle is required to place each 16-bit word in a register.

The Smalltalk Virtual Machine on the *Dorado*

Most of the features present in the architecture of the *Dorado* were exploited in the implementation of the Virtual Machine:

One of the most useful features of the *Dorado* to the implementors of the Smalltalk Virtual Machine was a mature assembly-level language, furnished by the *Alto* emulator. In fact, this was doubly beneficial because Smalltalk had already been implemented on the *Alto*, and some code could be transferred directly. Furthermore, because they had a BCPL compiler which targetted code for the *Alto*, they were free to move sections of the program between BCPL, *Alto* code, and *Dorado* microcode. One of the few deficiencies of the *Dorado* made this important; without the *Alto* emulator it might have been much more difficult to fit the Virtual Machine and associated support into the 4 Kwords of control store. As it was, 1300 words were already tied up with the input/output system; the *Alto* emulator used up another 700. Therefore, at a cost of about 20% of the control store, they were free to move as much of the program as necessary from microcode to *Alto* code. In addition to this, there were a substantial number of unused *Alto* opcodes—in the Smalltalk Virtual Machine these were defined to perform frequently required operations (such as reading the object table entry associated with an object pointer).

The IFU is perfectly matched to the requirements of the Virtual Machine: so much so that one suspects that the design of the Virtual Machine was strongly influenced by the architecture of the *Dorado's* IFU, and *vice versa*. It is capable of prefetching up to six bytecodes, and decoding each bytecode to provide: a microcode address; the number of additional bytes required by the instruction; and a 4-bit parameter which the processor can use for any purpose. All this is done by table look-up; the table has enough space for four separate bytecode instruction sets, and switching between the instruction sets can be done "on the fly" under processor control. This switching mechanism was used to alternate between Smalltalk bytecode interpretation and *Alto* emulation.

The design is such that there is no delay between macroinstructions; the last microinstruction of each macroinstruction contains a field which informs the processor that it should ask the IFU for a new dispatch address. Under certain conditions, the IFU can also autonomously execute unconditional branches in the bytecodes; processor intervention is only required for other branches, message sends and returns, and when switching between instruction sets. The result of all this is that the Virtual Machine spends only 2.4% of its time waiting for the IFU to supply a microcode address.

Although the *Dorado* provides support for a paged virtual address space, the Virtual Machine makes no

use of it. Because a typical Smalltalk bytecode can make many references to memory, restarting a bytecode after a page fault is potentially very complicated. Instead, the implementors decided to map virtual addresses directly onto real addresses.

The *Dorado* memory cache plays an important rôle in the implementation of the Smalltalk Virtual Machine. The principal feature of the cache is that it is fast: provided that the desired word is in the cache, a memory fetch takes two cycles, and a memory store only one. This means that large amounts of data (up to 32 Kb) can be accessed almost as fast as data in the registers. Because of this, much information was placed in memory rather than in registers or on a hardware stack: the active and home context stacks reside in memory, as do the free chunk list heads (in fact, indexing into registers is *slower* than indexing into memory). This is possible only because the of the speed and efficiency of the cache. As Deutsch states DEUT83:

> "If the cache had been only half as fast—taking 3 to 4 microinstructions to fulfill a request—
> it would probably have been better to store some of the current context in registers."

Of the 256 general-purpose registers present in the *Dorado* processor—which are in 16 groups of 16—the Virtual Machine only uses three groups. One reason for this is that only a single group is accessible at any given time; switching between groups requires extra microinstructions. In addition to these three groups, a number of the base registers are set up to point to frequently used objects, such as the active and home contexts, self, and the current method.

Microcoding the PERQ

As can be seen from the above description, a great deal of hardware is present in the *Dorado* which directly assists the rapid execution of bytecodes. Unfortunately, the PERQ is devoid of most of this useful hardware:

- The PERQ does not have an independent instruction fetch unit. Bytecodes must be loaded into the OP file (see Fig. 7) by explicit request (a 64-bit memory word is fetched to fill the file), and thereafter single bytes can be fetched from the file with no extra delay, until the file is empty. There is a form of microinstruction which can cause a branch to one of 256 microcode addresses, based on the value of the byte returned from the OP file, but the addresses must be spaced at four-word intervals, with the first on a four-word boundary. By comparison, the *Dorado* IFU RAM entries can contain any dispatch address. This means that on the PERQ there will be a short delay between bytecodes while the byte is fetched and decoded, and that there will be a significant interruption when the OP file has to be refilled.
- When comparing the PERQ and the *Dorado*, the most apparent difference lies in their memory systems. Whereas the *Dorado* has a fast, pipelined cache, designed so that the programmer does not

have to consider the timing of the main store, the PERQ insists that the microprogram synchronise with the slow main memory. For this reason, it would probably be faster to cache the contents of the active and home contexts in registers, but for the fact that indexing into the register block (for access to the context stack) is impossible (there is no addressing mode to support this). An alternative approach would be to place the context stack in the hardware stack; however, the 16-level hardware stack on the PERQ is not big enough to cache the stack of a large context. One would either have to place large context stacks in memory, or have some complicated stack management strategy that kept track of the locations of different parts of the stack. In any case, whenever the context changed, a series of memory transfers would be required to save the outgoing context and load the new context; fortunately the PERQ provides a mechanism for transferring a 64-bit memory word in two memory cycles.

• The registers in the PERQ are all equally accessible—at least this is better than the *Dorado*—but there are no indexed addressing modes for fast access to arrays in memory; one has to explicitly perform an addition to get the effect of indexing. On the *Dorado* one has the option of setting up base registers to access fields of commonly used objects. Happily, the ALU is a full 20 bits wide, and address calculations can be done reasonably quickly, whereas address calculations on the *Dorado* (using a 16-bit ALU) are slow because double precision arithmetic is required.

• Another weakness of the PERQ hardware is that the shifter is only 16 bits wide, compared to the *Dorado's* 32-bit shifter.

• The final deficiency, and to an implementor possibly the most serious, is that there is no stable base to work from. The *Dorado* implementors had the *Alto* emulator from which they could start, and the freedom to move sections of the program between BCPL, *Alto* code and microcode. An implementor using the PERQ would have to start from scratch, and write entirely in microcode. One consolation would be that the 16 Kword control store would leave lots of room for manœuvre.

An example of PERQ microcode

In this section we will present a small portion of an implementation of the Virtual Machine. First, we will give the original Smalltalk code, taken from the formal specification GOLD83, pp. 585–686. Then we will present equivalent C code, as in a naïve implementation; then a better example in C; and finally a version in PERQ microcode.

Smalltalk code

We have chosen to demonstrate code for the bytecode which pushes the first temporary variable of the active context (temporary 0) onto the stack (bytecode 16). The example has been chosen for its simplicity: as long as no reference count falls to zero, it only accesses object pointers in the active and home contexts. We take up the story in the Smalltalk bytecode dispatch loop, and assume that the current bytecode has already been fetched: (Note that in the following methods self refers to an instance of Interpreter.)

```
dispatchOnThisBytecode
    (currentBytecode between: 0 and: 119) ifTrue: [↑self stackBytecode].
    ...

stackBytecode
    ...
    (currentBytecode between: 16 and: 31)
        ifTrue: [↑self pushTemporaryVariableBytecode].
    ...
```

Then, to execute the actual bytecode:

```
pushTemporaryVariableBytecode
    | fieldIndex |
    fieldIndex ← self extractBits: 12 to: 15 of: currentBytecode.
    self pushTemporaryVariable: fieldIndex

pushTemporaryVariable: temporaryIndex
    self push: (self temporary: temporaryIndex)
```

The more important auxiliary methods are:

```
push: object
    stackPointer ← stackPointer + 1.
    memory storePointer: stackPointer
        ofObject: activeContext
        withValue: object
```

```
temporary: offset
  ↑memory fetchPointer: offset + TempFrameStart
     ofObject: homeContext
```

The routines called in the object memory (i.e., methods in the class RealObjectMemory, of which memory is an instance) are as follows:

```
fetchPointer: fieldIndex ofObject: objectPointer
  ↑self heapChunkOf: objectPointer word: HeaderSize + fieldIndex

storePointer: fieldIndex
      ofObject: objectPointer
      withValue: valuePointer
  | chunkIndex |
  chunkIndex ← HeaderSize + fieldIndex.
  self countUp: valuePointer.
  self countDown: (self heapChunkOf: objectPointer word: chunkIndex).
  ↑self heapChunkOf: objectPointer word: chunkIndex put: valuePointer
```

These routines form part of the interface to the bytecode interpreter. The following methods are internal to the object memory manager:

```
heapChunkOf: objectPointer word: offset
  ↑wordMemory segment: (self segmentBitsOf: objectPointer)
     word: ((self locationBitsOf: objectPointer) + offset)

heapChunkOf: objectPointer word: offset put: value
  ↑wordMemory segment: (self segmentBitsOf: objectPointer)
     word: ((self locationBitsOf: objectPointer) + offset)
     put: value
```

The segment:word: routines fetch and store a word from the real memory. (wordMemory is an instance of RealWordMemory.) The countUp: routine increments an object's reference count:

```
countUp: objectPointer
  | count |
  (self isIntegerObject: objectPointer)
     ifFalse:
       [count ← (self countBitsOf: objectPointer) + 1.
        count < 129 ifTrue: [self countBitsOf: objectPointer
                                 put: count]].
```

```
↑objectPointer
```

The countDown: method decrements an object's reference count, and deallocates the object if its reference count falls to zero. For simplicity, we will not present the code for deallocation:

```
countDown: objectPointer
  | count |
  (self isIntegerObject: objectPointer)
     ifTrue: [↑objectPointer]
     ifFalse:
        ...
        count ← (self countBitsOf: objectPointer) – 1.
        count < 127
           ifTrue: [self countBitsOf: objectPointer
                       put: count].
        ...
```

Note that the Smalltalk code uses a maximum count value of 128 in an 8-bit field, rather than 255. There only reason for this restriction seems to be that on most machines a test of an 8-bit quantity for the presence of a sign bit is marginally faster than comparison with a constant. To be consistent, we will continue using this scheme, although individual implementors may find that the use of 255 incurs little or no extra penalty.

Equivalent code in C

This code is taken directly from the implementation described in chapter 3. The first thing to notice is that bytecode dispatch is more efficient due to the use of a dispatch table.

```
int (*dispTable[])() = { ... pTVar, ... };
       /* the dispatch table */
...
(*dispTable[currentBC])();    /* bytecode dispatch */
...

pTVar()     /* push Temporary Variable */
    { push(temp(currentBC & 15)); }
```

The auxiliary routines are defined as macros:

```
#define push(obj)     storePtr(++stackPtr, activeContext, (obj))
#define temp(offset)  fetchPtr((offset)+TEMP_FR_START,
                                    homeContext)
```

Access to the object memory also takes place via macros:

```
#define fetchPtr(i, oop)    HCword(oop, HDR_SIZE + (i))
#define storePtr(i, oop, val)  {
    WORD oop1 = (oop), val1 = (val);
    WORD chunkInd = HDR_SIZE + (i);
    countUp(val1);
    countDown(HCword(oop1, chunkInd));
    HCwordPut(oop1, chunkInd, val1);
}
```

Note the use of temporary variables in order to prevent multiple expansion of macros, and extra parentheses to avoid unwanted side-effects.

The object memory and object table have the following structure:

```
struct ot_entry {
    BOOL     Stuck : 1;
    unsigned Count : 7;
    BOOL     Odd   : 1;
    BOOL     Ptrs  : 1;
    BOOL     Free  : 1;
    ADDR     Loc   : 20;
} ot[32*1024];

WORD om[OM_SIZE];  /* OM_SIZE=400*1024, e.g.*/
```

where `BOOL` and `ADDR` are defined to be `unsigned int`, and a `WORD` is an `unsigned short` (16 bits). Object memory access is performed by the following macros:

```
#define HCword(oop, offset)   (om[loctn(oop) + (offset)])

#define HCwordPut(oop, offset, val)
                          { HCword(oop, offset) = (val); }
```

Reference counting is done using macros and a recursive function:

```
#define countUp(oop) {
    WORD oop1 = (oop);
    if (!isInt(oop1) && !stuck(oop1))
        ++count(oop1);
}

#define countDown(oop) {
```

```
    WORD oop1 = (oop);
    if (!isInt(oop1) && !stuck(oop1))
        if (--count(oop1) == 0)
            /* deallocate objects recursively */
            ...
}
```

The `stuck`, `count` and `loctn` macros access the fields of the object table. The `isInt` macro tests whether an object pointer is tagged as a SmallInteger:

```
#define isInt(oop) ((oop)&TAG_BIT)
```

There are a number of optimisations which can be made when implementing a faster version in C. The most obvious is to cache the pointers to home and active contexts and the stack pointer as absolute addresses, thereby saving a number of indirections. Another time-saver is to have a separate routine for each bytecode; this only saves on one addition in this case, but can be significant when pushing a constant which is not reference counted, for example. Also, by loading the object table entry for an object only once, one can save on the fetching, shifting and masking which must inevitably occur when one uses C bit fields. Finally, placing the SmallInteger tag bit in the most significant position can substantially speed up testing object pointers, as can placing the reference count overflow bit in the same position. All SmallIntegers will then have negative object pointers, and an object table entry with a stuck reference count will also be negative.

Thus an efficient version of the same code might be (shown after any macros have been expanded):

```
pTVar0()
{
    register SIGNED ot_entry, old_oop;
    register SIGNED temp = homeContext[TEMP_FR_START];

    if (temp >= 0                /* not an integer pointer */
       && (ot_entry = ot[temp]) >= 0)         /* not stuck */
                               /* increase reference count */
            ot[temp] = ot_entry + 0x0100;

         /* increment stack pointer and fetch top of stack */
               /* then test whether its an integer pointer */
    if ((old_oop = *++stackPtr) >= 0
       /* test whether the old top of stack is an int.ptr. */
        && (ot_entry = ot[old_oop]) >= 0) {
           /* decrement ref.count and check for zero count */
```

```
            if ((ot_entry -= 0x0100) < 0x0100)
                /* recursively free object */
                ...
            ot[old_oop] = ot_entry;       /* replace ref.count */
        }
        *stackPtr = temp;   /* place temp.var. on top of stack */
    }
```

where `SIGNED` is a `short int` (16-bit). The next section will present a literal translation of this function into microcode, and will calculate how long it takes to execute.

An example of PERQ microcode

The architecture of the PERQ CPU was described earlier; it should be reasonably clear what each microinstruction is doing. The syntax of the microprogram does not conform exactly to that used by ICL; this is to make it more readable.

As before, we will start with bytecode fetch and dispatch. The description of the hardware stated that bytecodes could be taken from the OP file; the OP file has its own 4-bit *Bytecode Program Counter* (BPC), which is incremented whenever a bytecode is fetched from the OP file. Before a bytecode is fetched, the overflow bit of the BPC (bit 3) should be tested: if it is set, then the OP file has been exhausted, and should be refilled. To do this, we maintain an instruction pointer (IP) in one of the general-purpose registers, and perform a `LoadOP` instruction, which fills the OP file with 8 bytes from memory. The bytecode dispatch is performed by the `NextInst` instruction, which transfers control to one of 256 locations, depending on the value of the next byte in the OP file.

```
        If BPC[3] Goto (REFILL)                         0
            ! if the OP file is empty, refill it
        NextInst (BYTECODETABLE)                        1
            ! branch to one of 256 locations. The
            ! first location is at BYTECODETABLE.
REFILL:
        Fetch4, IP := IP + 8                            3*
            ! Increase instruction pointer, and
            ! fetch next 4 words (8 bytes) of program
        LoadOP                                          012301*
            ! and place them in the OP file
        ReviveVictim                                    2
            ! This "revives" the instruction after
            ! the test of BPC, in this case NextInst.
```

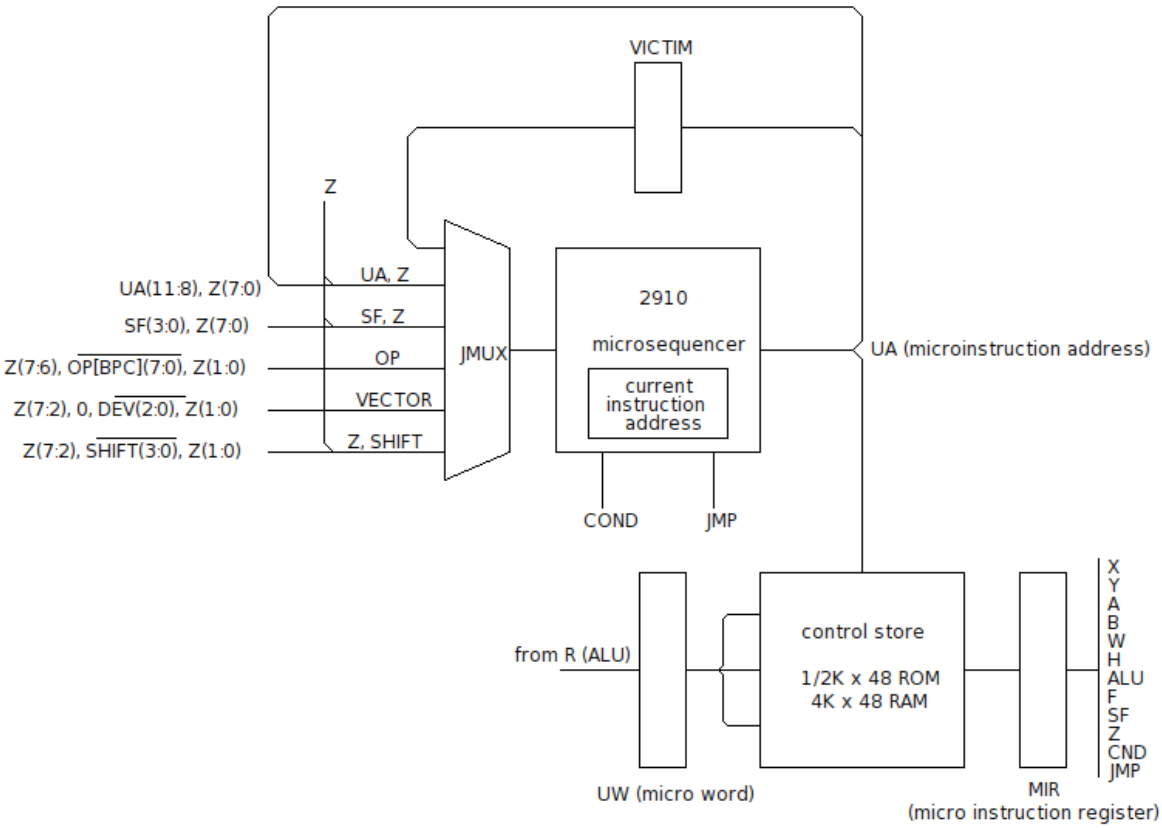

Figure 8. The PERQ microinstruction fetch and dispatch unit

The function of `ReviveVictim` instruction is to re-execute the instruction that followed a test of BPC[3], and was aborted because OP was empty (see Fig. 8).

The next section of code resides in the dispatch table; it cannot contain more than four instructions, because of the pre-determined, fixed spacing of the table. Because store fetches take a number of cycles, we initiate the fetch before jumping out of the table; the code at the destination of the jump will read the data value when it arrives.

```
Push (Home + TEMPFRAMESTART)                    2
    ! push the address of the temporary
    ! variable onto the hardware stack
```

```
        Fetch (TOS)                                     3*
            ! Initiate a fetch. The memory address
            ! is taken from the top of stack
        Goto (PUSH)                                     0
            ! Jump out of the dispatch table
        ...
```

Next comes the code at the destination of the jump, which may be shared by all bytecodes that push a value onto the stack. It assumes that the address of the variable to be pushed is on the top of the hardware stack (`TOS`), and that a memory fetch for the variable has been initiated.

It first checks whether the variable is an instance of SmallInteger, and if not, whether its reference count has overflowed; if it has not, then it is incremented.

```
PUSH:
        TOS := MDI                                      12*
            ! store the word from memory on the top
            ! of the hardware stack
        If LT Goto (NO_COUNT_UP)                        3
            ! don't reference count SmallIntegers
        Fetch (OT_BASE + TOS)                           0123*
            ! fetch the first word of the object
            ! table entry for the temporary...
        ot_entry := MDI                                 012*
            ! ...and store it in a register
        If LT Goto (NO_COUNT_UP)                        3
            ! don't increase a stuck count
        Store (OT_BASE + TOS)                           012*
            ! replace the object table entry...
        MDO := ot_entry + #0100                         3*
            ! ...with its ref. count increased
NO_COUNT_UP:
```

The next thing to do is to increment the stack pointer, and to decrement the reference count of the object pointer which is being overwritten.

```
        Fetch (SP := SP + 1)                            0123*
            ! increment the context stack pointer,
            ! fetch the value it points to...
        old_oop := MDI                                  012*
            ! ...and store it in a register
```

```
        If LT Goto (NO_COUNT_DOWN)                      3
            ! don't ref. count SmallIntegers
        Fetch (OT_BASE + old_oop)                       0123*
            ! fetch the first word of the object's
            ! entry in the object table...
        ot_entry := MDI                                 012*
            ! ...and store it in a register
            ! (re-using the register used earlier)
        If LT Goto (NO_COUNT_DOWN)                      3
            ! don't decrease stuck ref. counts
        ot_entry := ot_entry - #0100                    0
            ! decrement the ref. count
        ot_entry & #FF00                                1
            ! test if the ref. count is zero...
        If EQ Call (DE_ALLOC)                           2
            ! ..and call a deallocater if it is.
            ! (There is a five-level microcode
            ! call stack in the PERQ)
        Store (OT_BASE + TOS)                           3012*
            ! replace the word from the object table
        MDO := ot_entry                                 3
NO_COUNT_DOWN:
```

The only thing that remains is to push onto the context stack the value passed from the code in the dispatch table:

```
        Store (SP)                                      012*
            ! address at which to store...
        MDO := Pop                                      3
            ! ...the value passed to PUSH
        Goto (LOOP)                                     0
```

The code at `LOOP` will perform any necessary input polling, and check for process switching.

An estimate of speed

The PERQ CPU nominally executes one microinstruction in every microcycle. However, access to main memory must be synchronised with the main memory cycle, which is four microcycles long. We will denote the four minor cycles t0–t3. At the right hand side of the microcode we show which memory minor cycles a particular microinstruction executes in. An asterisk denotes that synchronisation is enforced by the PERQ hardware. The (much simplified) rules for determining these are:

- Fetch requests are held until t3, and the data becomes available in the following t2.
- Store requests are held until t2, and the data must be presented in the following t3.

There are many such rules concerning multi-word transfers, and the timing of successive memory requests; they occupy nearly four pages of fine print in the PERQ microcode manual ICL82b. The cycle numbers given in the example above assume that none of the conditional branches are executed; those given for the OP file refill are merely to illustrate how long a refill will take.

Assuming that none of the branches are taken, this section of microcode takes 49 microcycles to execute. After every eight bytecodes, the OP file will have to be refilled, taking another eight or so microcycles. In addition, we have not shown the process switching and input polling code. Most of it will only be executed once every several hundred bytecodes, and therefore only the code which is executed for every bytecode will be important; this should only occupy ten to fifteen cycles.

Thus it seems that we have completely executed a bytecode in about 60–70 microcycles, or 11 μs. But it not safe to assume that this an average speed for bytecodes; while the example we have chosen is the single most common bytecode executed in a typical Smalltalk-80 system (at 6.5% of all bytecodes executed; 43–44% of all bytecodes executed are push bytecodes*), many less frequent bytecodes are much more complicated.Message sends which cause context changes are also common (6–11%), and take much longer to execute. All this suggests that an average rate of 50000–80000 bytecodes per second is possible. If one considers the differences between the *Dorado* and PERQ, this seems a reasonable estimate:

- The raw cycle time of the *Dorado* is three times better than that of the PERQ
- Access to the main store is *much* quicker on the *Dorado*, due to the cache. A request which results in a cache hit is over five times faster than a main store request on the PERQ. Of course, it may be possible to recode the example given so that the context stack is cached in the CPU; however it will be difficult to evaluate whether the management overheads of such a scheme will outweigh any advantages, without actually implementing both schemes and benchmarking the results.

Given these major differences, a 6:1 performance ratio between the *Dorado* and the PERQ seems justified.

* Figures from UNGA83 and FALC83b

As a final note, we should state that while the PERQ does not compare favourably with the *Dorado*, an implementation at 50000 bytecodes per second would be the second fastest known, based on reported performance figures at the time of writing MCCA83. All this leads one to believe that it is not that the PERQ is deficient, but that the *Dorado* outclasses everything else for this sort of task.

A programming language is low level when its programs require attention to the irrelevant.

If two people write exactly the same program, each should be put into microcode, and then they certainly won't be the same.

A. J. Perlis, Epigrams on Programming (1982)

5. Conclusions

The aim of this dissertation has been to show three things: first, that implementing the Smalltalk Virtual Machine on the PERQ by translating the methods of the formal definition into C is straightforward; second, that an improved version of this implementation can be constructed which has reasonable performance; and finally, that an implementation in microcode which has excellent performance is feasible, albeit difficult. We will examine each of these claims in more detail.

An initial implementation

Chapter 3 described how an initial implementation was constructed by translating from the formal definition of the Smalltalk Virtual Machine. The only requirement of this implementation that might prove unsatisfiable is that well over 1 Mb of main memory is required in a host PERQ to avoid severe degradation in performance due to thrashing. Otherwise, the implementation delivers a performance above 2000 bytecodes per second, which would prove barely usable. However, the code is quite portable; a re-implementation on a different machine should be possible with a few days effort, provided that the machine-dependent portions (mainly input and output) were not too problematic. Furthermore, a significant increase in performance is achievable without great additional effort: this is because the speed of the implementation is closely tied to the structure of the object memory macros; carefully changing these macros to be in sympathy with the object code of the host computer could produce dramatic benefits. Unfortunately, it was impossible to do this in a systematic way on the PERQ, because a run-time profiler was unavailable, and the nature of the assembly language is unknown. If one could examine the assembly code produced by the C compiler, it would be possible to compare the execution patterns of different versions of the same macro, and decide which was the fastest.

Improving the initial implementation

Chapter Three also described a number of algorithmic improvements (as opposed to the aforementioned machine-specific enhancements) which can be made to the initial Virtual Machine implementation. Without being able to profile the existing code to determine where most of the time is being spent, it is difficult to make specific recommendations as to which changes will bear the most fruit. However, given the nature of the current implementation, the following changes are not too difficult, and should produce significant benefits:

- Storing pointers to the active and home contexts, the next instruction and the stack pointer as absolute addresses (C pointers) should have the greatest effect of all optimisations, since these are the most frequently used items of information in the state of the Virtual Machine. The majority of changes will be to macros, and are therefore localised to the macro modules, but care will have to taken to ensure that the integrity of these variables is preserved when objects are moved about in memory (due to compaction, for example).
- In a working Smalltalk-80 system, almost all of the available object pointers are in use. Therefore moving the class and size fields of an object from the object memory to the object table will not waste a great deal of memory, and will speed access to these important items of information. The changes required to achieve this are few in number, because of the use of macros to access these fields.
- If memory is plentiful—as it should be on a two megabyte PERQ—one can speed up reference counting substantially by expanding the reference count field of an object table entry to sixteen bits. This saves on expensive shifting and masking when the reference count is accessed, and also obviates the need for overflow checks. In a similar vein, restructuring the object table so that the fields of each entry are in separate arrays will result in speed gains, at the expense of memory space.
- Moving the SmallInteger tag bit of an object pointer from the least significant position to the most significant bit position will result in faster testing of object pointers, because all pointers to SmallInteger objects will be negative, in a sixteen-bit field. Another benefit will be faster access to the object table, because non-integer object pointers will not need any conversion to make them into object table indices.

The stack management strategy mentioned in chapter 3, and described in more detail in UNGA83, can result in dramatic increases in speed. Unfortunately, it would require major changes to the existing implementation. For this reason it is recommended that the abovementioned changes are made and their effects assessed before a commitment is made to drastic restructuring of the existing code. At a guess—and without any hard evidence it can be little more than a guess—the abovementioned changes will result in a 100–200% speed-up.

A microcode implementation

Chapter 4 investigated the issues involved in microcoding the PERQ to emulate the Smalltalk Virtual Machine. Because one does not have an operating system "getting in the way", and one has the freedom to define almost any sort of virtual machine architecture, it is almost certain that the Smalltalk Virtual Machine can be implemented on a PERQ with 16 Kwords of control store. On the other hand, an efficient implementation in 4 Kwords of control store is almost certainly impossible; the shortage of control store would force one to define an instruction set intermediate in complexity between the bytecodes of the Smalltalk Virtual Machine and the PERQ microcode, and write large portions of the bytecode emulator in the intermediate code (much as the *Dorado* implementors used the *Alto* emulator). The program in the control store would interpret the intermediate code, as well as directly executing small, critical sections of the Virtual Machine. On might ask, Why was this task so eminently successful on the *Dorado*, yet so singularly unattractive on the PERQ? As chapter 4 emphasises, there is a great deal of hardware in the *Dorado* which would have to be emulated in microcode on the PERQ.

An interesting property of a microcode implementation is that it should run comfortably within 1 Mb of main memory; over a quarter of a megabyte is saved by not having resident the PNX operating system and window manager. Of course, the principal disadvantage would be that all the kernel code of the operating system would have to be re-implemented. Microcode would be required to perform disk transfers, as well as other input and output, and to support floating point arithmetic.

We can conclude that a microcode implementation would be a major undertaking. The *Dorado* implementation took six person-months; one suspects that an implementation on the PERQ would take twice as long. To be successful within this timescale, the implementor(s) would find it beneficial to develop microcode tools that could detect, if not avoid, the more difficult and subtle aspects of timing interactions with the main store. Another useful tool to enhance performance would be a "peephole" optimiser. Main store requests usually result in a significant delay, a tool that could reorder instructions to minimise the effects of the delay might prove advantageous.

As a final note, we should add that the microcode implementation described is the simplest that one could envisage being attempted. Much more complicated implementations involving larger object pointer spaces and virtual memory are also possible (Kaehler Virtual memory Byte [KAEH81], Kaehler Krasner LOOM [KAEH83]); these are possible topics for future research.

... these objects are not subjective fantasies.

G. Frege, The Foundations of Arithmetic (1884)

Smalltalk is the object of much interest.

S. C. Holden

Appendix A
Some interesting features of the Smalltalk-80 Virtual Machine

When one has become intimately familiar with the operation of the Smalltalk-80 Virtual Machine, a number of interesting details emerge that were not obvious at first sight. Since a reduction in the number of concepts in a system is always a good thing, one naturally asks whether these features are essential to the operation of the Virtual Machine, or whether they can be removed or simplified. This appendix describes a number of characteristics of the Smalltalk-80 Virtual Machine which are in some way unusual, and distinguish the Smalltalk-80 system from conventional systems.

Blocks and block contexts

The concept of blocks with arguments is a recent development in the Smalltalk language Ingalls evolution Smalltalk Virtual Machine [INGA83], and is one of its most unusual and interesting concepts. Blocks were described in chapter 2; to summarise, a block is a piece of code which is not bound to a selector. It is activated by sending it the value message, rather than in response to the method dictionary lookup that follows a message send. Because blocks are considered to be independent objects (even though they are not implemented directly as such), they can be used as arguments to messages, and returned as results. Associated with the code in the block is its *context*, which defines how the identifiers used in the block are bound to objects. For example, in the following method (which we define to be in class Number),

```
plus: n
    ↑[self + n]
```

n must always refer to the object passed as an argument to plus:, even if the method which later sends the value message to the block has a local n. This example also illustrates why contexts cannot be disposed of as soon as they are returned from: a method that sends plus: to an instance of Number will be returned a reference to a block that contains a reference to a local variable of the method for plus:. This local variable is in the context for plus: (on the temporary frame) and therefore the context cannot be disposed of, even though its method has returned. (Conventional languages with code blocks have made such situations illegal [e.g., as in Algol 68] or undefined [by creating a dangling reference], in order to preserve a strict stack discipline for procedure activations.)

An interesting consequence of the ability to refer to a returned context is that a block context can attempt to return *to* a method context that has already been returned *from*. Consider the following example:

```
plusb: n
    ↑[↑self + n]
```

in which we have simply added an extra return within the block. Another object might send the plusb: message to a Number, and then try evaluating the returned block:

```
...
| block |
    ...
    block ← 3 plusb: 5.
    block value.
```

In this situation, the return from within the block (after value has been sent) will try to return to the context for plusb:, which has already been returned from. The Virtual Machine must detect such an occurrence, and inform the context which is trying to return that it cannot. In the definition of the Virtual Machine, this is done by nilling out the instruction pointer when a context returns successfully, and then checking for this condition before returning again. If an illegal return is detected, the erroneous return is signalled by sending a cannotReturn message.

Although the treatment of blocks in the Smalltalk language is such that they are full and equal objects, this is certainly not the case in the Virtual Machine. A block, or more properly a block together with its context, is stored in two parts. The bytecodes belonging to the block are embedded within the bytecodes of the method that contains the text of the block. The context (which contains the arguments to the block, and its own stack) is a separate object of class BlockContext, and is created by invoking the blockCopy: primitive, which determines the bindings of the variables (see Fig. 3). Once the block context has been created, the value: primitive can be used to activate it. By embedding the bytecodes of the block within those of the method, the Smalltalk compiler is able to generate efficient code for the ifTrue: and whileTrue: messages. Instead of actually sending messages for ifTrue: and whileTrue:, the compiler plants code to evaluate the loop/branch condition, followed by the appropriate jump bytecodes; this saves on the creation of a context for each branch and iteration (a not insubstantial saving). The disadvantage of this scheme is relatively minor: apart from a (subjective) lack of elegance, one cannot define ifTrue: and whileTrue: messages in classes other then Boolean.

The alternative—to have blocks as independent objects completely separate from methods—poses a number of problems. Aside from the efficiency considerations outlined above, there is the problem of how to implement infinite loops by sending messages, without infinite stack expansion (the main cycle of the interpreter is such a loop). Furthermore, because the code of each block would be a separate object, there would be a dramatic increase in the demand for object pointers, which in the standard 16-bit system are already in short supply.

Thus it would seem that the peculiar implementation of blocks and their contexts is the most satisfactory in the circumstances.

What happens when a message is not understood

Because there is no way of checking at compile-time that a message will be understood by its receiver, the Virtual Machine detects the absence of a matching selector when performing the dictionary lookup, and sends an error message, doesNotUnderstand, to the sender. Because doesNotUnderstand is defined within class Object, it should always be understood. However, because the user has access to all of the system, he can delete this message from the list of messages understood by class Object. While one might say that a user who does this deserves everything he gets, the published definition of the Virtual Machine goes to some lengths to detect this occurrence, and signal the user (via some unspecified mechanism) as to what has happened. This, together with other ways of crashing the system (the classic one being Processor ← nil), suggests that it would be desirable to limit the things a user can do, especially when the user can invalidate the assumptions under which the Virtual Machine operates.

Stack sizes and stack overflow

The Virtual Machine uses only two different sizes of context, one containing a 12-word stack and the other containing a 32-word stack. The question arises, If the maximum stack size can be computed at compile-time, why isn't the exact context size used for each activation? There are two reasons for not using the exact size of context: first, over 85% of the allocation requests in the system are for contexts FALC83b, and a proliferation of context sizes would lead to an increase in fragmentation. The second reason is that it is *not* possible to calculate the maximum stack size in every case. One of the primitives present in the Virtual Machine is the perform: primitive, which takes a message selector and a list of arguments, and dynamically sends a message constructed from them. Because the size of the argument list can vary from invocation to invocation, it is not possible to compute the size of the stack required to accommodate the argument list. In fact, the only place in the Virtual Machine where the stack is checked for overflow occurs in the code for the perform: primitive. If the argument list cannot fit in the remaining space, the primitive fails, and Smalltalk code must take over (presumably generating a new, larger context, and substituting it in place of the smaller context).

Compiled methods

The most unpleasant aspect about the design of the Smalltalk-80 Virtual Machine is the representation of compiled methods (instances of class CompiledMethod). A compiled method is separated into three parts:

- The first word of the compiled method contains a method header, which is an encoded representation of various vital statistics. This word is represented as an integer pointer.
- The second part of a compiled method is the literal frame, which contains all the object pointers referred to literally in the bytecodes of the method.
- The third part contains the bytecodes which constitute the code of the method.

The unpleasantness stems from the fact that object pointers and bytes (bytecodes) coexist in the same object. This complicates many aspects of the memory manager (e.g., the deallocater must only deallocate objects referred to in the literal frame, and pass over the bytecodes), and means that one cannot create subclasses of CompiledMethod. Furthermore, special primitives have to be provided to create and access compiled methods.

The reason for this representation lies in the limited number of object pointers available in the standard 16-bit system; separating each compiled method into two objects would add several thousand new objects to the system, leaving very few for the user. The author's feeling on this subject is that the sooner 16-bit object pointers are abandoned, the better. The Smalltalk-80 system is bursting at the seams under this artificial

restriction, and any scheme which could overcome it would be a major improvement. At the moment, the most promising candidate is the Large Object Oriented Memory (LOOM) KAEH83, which supports 32-bit virtual object pointers, while only requiring 16-bit pointers for those objects in memory.

The representation of small integers

A side effect of the inadequate object pointer space is that SmallIntegers are also inadequate. For example, while one may create an object with up to 64 K fields, one cannot address all the fields of this object with 15-bit integers. This complicates the array accessing primitives enormously, which must handle both SmallInteger and LargePositiveInteger indices; whereas SmallIntegers are encoded into object pointers, LargePositiveIntegers are byte objects.

Can the design of the Virtual Machine be improved?

On the whole, the Virtual Machine is well-designed, robust, and efficient in space and time. However, as more people come to use the Smalltalk-80 system, and place increasingly severe demands upon it, it is difficult to see the limited size of the system supporting ambitious applications. It is felt that the next natural evolutionary step of the Smalltalk-80 system is to incorporate some scheme of virtual memory. The number of available objects is becoming a problem; the amount of memory required to run a system has always been a problem.

Another step in the evolution of the Virtual Machine might be a reduction in the number of fundamental concepts "wired in" to the Virtual Machine. A particularly good candidate for this is the abolition of mixed-type compiled methods; this would fit in very well with an increase in the object pointer space, and would also make a number of primitives redundant.

The difficulties involved in identifying the fundamental concepts required in the Virtual Machine indicate that the published definition is in some way deficient. A useful exercise would be to create a formal definition of the operation of the Smalltalk Virtual Machine that precisely specified its semantics, without getting bogged down in implementation details. This would enable an implementor to choose alternative algorithms and data representations, and still be confident that his implementation met the criteria required of a Virtual Machine. It would also force the implementors of the Virtual Image to eliminate (or parameterise) all features of the image that were dependent on the implementation decisions made in the Virtual Machine. It is also felt that the clarity of thought and description imposed by a rigorous definition would lead to insights concerning the efficiency of the design of the Virtual Machine, and suggest ways that the structure of the Virtual Machine might be improved.

The final possibility is the implementation of the Virtual Machine in hardware. However, the complexity

which has been highlighted in the preceding sections suggests that this will be a significant undertaking. It is more likely that most good implementations will be done in microcode, which approaches the speed and efficiency of hardware, but without sacrificing the flexibility available in software.

Number thus emerged as an object...

G. Frege, The Foundations of Arithmetic (1884)

To iterate is human, to recurse divine!

L. P. Deutsch

Appendix B
The production of this dissertation

The text

When writing a dissertation that is to be submitted for examination, there comes a time when one considers the mechanics of the production. For some time the author has felt that the process of writing is greatly assisted by the use of computer-based text processing. As a consequence, it was natural to investigate the means available for the production of this thesis. Because of the generous provision of computer facilities for the Advanced M.Sc. course in Computer System Design in the Computer Science Department at the University of Manchester, there was no problem in finding sufficient resources (storage and terminal time) to type in and edit the dissertation on-line. A *VAX*-11/750 running UNIX had been available since the beginning of the course in October 1983, and a capable text editor (in the shape of EMACS) was also present. The only outstanding problem concerned the final stage of text processing, i.e., producing a hard copy of acceptable quality. A few students on the same M.Sc. course found facilities available in the UNIX `nroff/troff` text processors, used in conjunction with dot-matrix and daisywheel impact printers, but the author felt that the limited font capabilities of these devices, together with the severe load that was already placed upon them, made it desirable to find another route. It was with this background, and a personal interest in computer-based text processing, that it was discovered that Donald Knuth's TEX text processing system might be available.

In the spring of 1984, Tony Arnold, of the University Computer Graphics Unit, began experimenting with an old version of the TEX processing system, using the Graphics Unit's VMS *VAX*-11/750. By the summer, he had the latest version of TEX, TEX82, working successfully on that machine, with hard copy produced on a Benson 9211 electrostatic plotter, owned by the University of Manchester Regional Computer Centre (UMRCC). Upon hearing about this, and seeing the high quality of output produced by the system, the author felt that this would be an ideal system for the production of his own dissertation, and the dissertations of two other students on the same course.

There still remained a problem: the Graphics Unit could not provide the terminal or machine time that would be required by three students, because of a shortage of terminals, and production of theses was not seen as a legitimate use of their resources. The only tenable solution was to provide a TEX capability on the M.Sc. UNIX *VAX*.

A copy of part of the TEX system (which is in the public domain) was obtained on the 25th of July 1984, and transferred to the M.Sc. *VAX*. The task facing the author was to change the operating-system dependent parts of VMS-TEX to function correctly under UNIX. (Although TEX is available for VAX/UNIX

systems, it is based on the standard UNIX Pascal compiler, whereas the author wanted to use a new, fully standard Pascal compiler from the University of York. Furthermore, it was unlikely that funds could be found for a copy of the UNIX-TEX system to be ordered and delivered in time.) Even though the TEX system is made up of thousands of lines of Pascal, this first stage was relatively straightforward. All the system-dependent parts of the system are isolated in small "change files", and since we had the change files for VMS, it was straightforward to adapt these to UNIX. After three weeks of intermittent activity, the TEX program produced device-independent (DVI) output, which required transformation into pixel files suitable for the Benson plotter. This transformation still took place on the Graphics Unit *VAX*, and since it required significant amounts of machine time (as well as tying up the only tape deck), it seemed prudent to transfer this stage to the UNIX *VAX*.

On the 21st of August the source code for the transformation program (about 4000 lines of Pascal, and 200 lines of *VAX* assembly language) was copied to the UNIX *VAX*. Unfortunately, this program was not written using change files, and converting it to run under UNIX took over a week of full-time effort (during which the author learned a great deal about VMS Pascal!). Output is still produced using the UMRCC Benson plotter, but all processing is performed on the M.Sc. UNIX *VAX*, and a tape containing the pixel files is carried down to the minicomputer that drives the plotter.

Since the frequency with which files can be printed is quite low, it was decided to implement a rudimentary preview facility, using a PERQ as the display device. This was operating successfully in mid-September, with an added benefit being that hard copy was available using a dot-matrix printer connected to the PERQ. Unfortunately this is *very* slow, because of the low speed of the line connecting the *VAX* and the PERQ. There are two possible developments that might ease the situation. The first is to implement the DVI-to-pixel transformation program on the PERQ, and transmit DVI files (which are much smaller than pixel files) to the PERQ. This should be reasonably simple, because the same Pascal compiler runs on the PERQ as on the *VAX*, and both use the UNIX operating system (with minor differences). The only foreseeable problem is that the the transformation program requires almost seven megabytes of disk storage for font information, which would take up a large part of the thirty megabytes available on the PERQ disk. The other ameliorating development will be the planned installation of an Ethernet local area network, connecting the *VAX* and PERQ. Once this is functional, transmitting pixel images between the machines should not be a problem. (The current scheme takes about 5 minutes to transmit the image of an average page [which is run-length encoded] over a 9600 bits-per-second (bps) serial line. At 10 Mbps, Ethernet will be capable of transmitting the same information in a few seconds.) The final development (which should occur in the near future) will be the connection of a Canon LBP-10 laser printer to the PERQ. Once this is complete, and an

Ethernet installed, a complete TEX capability will then be available entirely within the Computer Science Department.

The diagrams

The diagrams in this dissertation were entered and edited on a PERQ, using the DP drawing package from Carnegie-Mellon University. The diagrams were then transmitted to the aforementioned *VAX*, where they were drawn on a Benson 1302 plotter (connected to the *VAX*).