Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern programming systems are among the most difficult to implement efficiently, even individually. A new implementation of the Smalltalk-80 system, hosted on a small microprocessor-based computer, achieves high performance while retaining complete (object code) compatibility with existing implementations. This paper discusses the most significant optimization techniques developed over the course of the project, many of which are applicable to other languages. The key idea is to represent certain runtime state (both code and data) in more than one form, and to convert between forms when needed.

*Smalltalk-80 is a trademark of the Xerox Corporation.

BACKGROUND

The Smalltalk-80 system is an object-oriented programming language and interactive programming environment. The Smalltalk-80 language includes many of the most difficult-to-implement features of modern programming languages: dynamic storage allocation, full upward funargs, and call-time binding of procedure names to actual procedures based on dynamic type information, sometimes called *message-passing*. The interactive environment includes a full complement of programming tools: compiler, debugger, editor, window system, and so on, all written in the Smalltalk-80 language itself. A detailed overview of the system appears in [SCG 81]. [Goldberg 83] is a technical reference for both the non-interactive programmer and the system implementor; [Goldberg 84] is a reference manual for the interactive system.

Special Difficulties

The standard Smalltalk-80 system implementation is based on an ideal *virtual machine* or v-machine. The compiler generates code for this machine, and the implementor's documentation describes the system as an interpreter for the v-machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure activations just like any other data objects.

The Smalltalk-80 language approaches programming with generic data types through *message-passing* and dynamic typing. To invoke a procedure (*method* in Smalltalk-80 terminology), a message is sent to a data object (the *receiver*), which selects the method to be executed. This means that a method address must be found at runtime. At a given lexical point in the code, only the message name (*selector*) is known. To perform a *message-send*, the data type (*class*) of the receiver is extracted, and the selector is used as a hash index into a table of the *message dictionary* of the class, which maps selectors to methods. The task of *method-lookup* is complicated by the *inheritance* property of classes -- a class may be defined as a *subclass* to another, inheriting all of the methods of the superclass. If the initial method-lookup fails, the lookup algorithm tries again using the message dictionary of the superclass of the receiver's class, continuing in this way up the class hierarchy until a method corresponding to the selector is found or the top of the inheritance hierarchy is reached.

The Smalltalk-80 language uses the organization of objects into classes to provide strong information hiding. Only the methods associated with a given class (and its subclasses) can access directly the state of an instance of that class. All access from "outside" must be through messages. Because of this, a Smalltalk-80 program must often make procedure calls to access state where languages such as Pascal could compile a direct access to a field of a record. This makes the performance of the method-lookup algorithm even more critical.

Implementation Outline

The purpose of the research described here was to build a Smalltalk-80 system with acceptable performance on a relatively inexpensive, microprocessor-based computer; specifically, to discover how to implement the basic data and code objects of the Smalltalk-80 system in a way that still conformed to the v-machine specification, but were more suitable for conventional hardware. (As of early 1982, the only implementations that ran at acceptable speed were on non-commercial, user-microprogrammable machines, as described in [Krasner 83] [Lampson 81].) The system specification in [Goldberg 83] includes the definition of internal data structures and object code representation for the virtual machine. Indeed, much of the system code depends on these definitions. We chose to take these definitions as given, rather than alter the system code.

This was motivated partly by a desire to retain object-code portability, and partly by a desire not to complicate the description of the Smalltalk-80 machine model.

The single principle that underlies all the results reported here is *dynamic change of representation*. By this we mean that the same information is represented in more than one (structurally different) way during its lifetime, being converted transparently between representations as needed for efficient use at any moment. An important special case of this idea is *caching*: one can think of information in a cache as a different representation of the same information (considering contents and accessing information together) in the backup memory. In the implementation described in this paper, we applied this principle to several different kinds of runtime information in the Smalltalk-80 system.

* We dynamically translate v-code (i.e., code in the instruction set of the v-machine) into code that executes directly on the hardware without interpretation, the native code or *n-code*. Translated code is cached: it is regenerated rather than paged.

* We represent procedure activation records (*contexts* in Smalltalk-80 parlance) in either a machine-oriented form, when they are being used to hold execution state, or in the form of Smalltalk-80 data objects, when they are being treated as such.

* We use several different caches to speed up the polymorphic search required at each procedure invocation. In the best case, which applies over 90% of the time, a Smalltalk-80 procedure invocation requires only one comparison operation in addition to a conventional procedure linkage.

* Using the techniques in [Deutsch&Bobrow 76], we represent reference count information for automatic storage management in a way that eliminates approximately 85% of the reference counting operations required by a standard implementation.

CODE TRANSLATION

Targeting code to a portable v-machine has been used in other language implementations. Usually v-code targeting is used only to avoid having multiple (one per target machine) code-generation phases of the compiler; a secondary benefit is that v-code is usually much more compact than code for any real machine. Since the Smalltalk-80 compiler is just one tool available in the same interactive environment used for execution, and other tools besides the compiler must be able to examine the machine state, the v-machine approach is even more attractive in reducing the cost of rehosting.

PERFORMANCE ISSUES

To rehost the system, an implementor must emulate the v-machine on the target hardware, either in microcode or in software. This normally incurs a severe performance penalty arising from several factors.

* Processors have specialized hardware for fetching, decoding, and dispatching their own native instruction set. This hardware is typically not available to the programmer (although it may be available at the microprogram level), and therefore not useful to the v-machine interpreter in its time-consuming operation of instruction fetching, decoding, and dispatching.

* The v-machine architecture may be substantially different from that of the underlying hardware. For example, many v-machines, including both the P-system and Smalltalk-80 v-machines, use a stack-oriented architecture for convenience in code generation, but most available hardware machines execute register-oriented code much more efficiently than stack-oriented code.

* The basic operations of the v-machine may be relatively expensive to implement, even though the overall algorithm represented by a v-code program may not be much more expensive than if it were implemented in the hardware instruction set. For example, even though a naive interpreter for the Smalltalk-80 v-code must perform reference counting operations every time it pushes a variable value onto the stack, a sequence of several instructions often has no net effect on reference counts.

If the v-code were translated to n-code after normal compilation of a source program to v-code, the interpreter's overhead could be eliminated and some optimizations become possible. One technique for eliminating part of the overhead of interpretation is *threaded code* [Bell 73] [Moore 74]. In this approach, v-code consists of an actual sequence of subroutine calls on runtime routines. This technique does reduce the overhead for fetching and dispatching v-code instructions, although it does not help with operand decoding, or enable optimizations that span more than one v-instruction. We prefer to translate v-code to in-line n-code in a more sophisticated way.

Naive translation from v-code to n-code is a process something like macro-expansion. In fact, [Mitchell 71] observed that a translator can be derived very simply from an interpreter by having the interpreter save its action-routine code in a buffer rather than executing it. If the computation performed by individual action routines is small relative to the computation needed for the interpreter loop, the benefit of even this simple kind of translation will be great.

Translation-time can also be considered an opportunity for peephole optimization or even mapping stack references to registers [Pittman 80]. Translation back-ends for portable compilers have been implemented [Zellweger 79].

DYNAMIC TRANSLATION

Because the Smalltalk-80 v-code is a compact representation that captures the basic semantics of the language, n-code will typically take up much more space than v-code. (In the implementation discussed in this paper, n-code takes about 5 times as much space as v-code.) This would place severe stress on a virtual memory system if the n-code were being paged. However, since n-code is derived algorithmically from v-code, there is no need to keep it permanently: it can be recomputed when needed, if this is more efficient than swapping it in from secondary storage. This leads us to the idea of translating at runtime. (The idea of dynamic translation appears in [Rau 78], where it is applied to translation from v-code to microcode.) When a procedure is about to be executed, it must exist in n-code form. If it does not, the call faults and the translator takes control. The translator finds the corresponding v-code routine, translates it, and completes the call. Since, as mentioned earlier, the translation process is more akin to macro-expansion than compilation, translation time for a v-code byte is comparable to the time taken to interpret it.

We consider the translation approach, and dynamic translation in particular, to be the most interesting part of our research, since it motivated the work on multiple state representations described below. A later section of this paper presents the experimental results that support our contention that dynamic translation is an effective technique in a substantial region of current technological parameters.

MAPPING STATE AT RUNTIME

Since the definition of the Smalltalk-80 v-machine makes runtime state such as procedure activations visible to the programmer as data objects, an implementation based on n-code must find a way to make the state appear to the programmer as though it were the state of a v-machine, regardless of the actual representation. The system must maintain a mapping of n-machine state to v-machine state; in particular, it must keep the v-code available for inspection.

How can we guarantee that all attempts to access a quantity requiring representation mapping are detected? The structure of the Smalltalk-80 language guarantees that the only code that can access an object of a given class directly is the code that implements messages sent to that class. Thus, the only code that can directly access the parts of an object requiring mapping is code associated with that object's class. Recall that all the code in the Smalltalk-80 system is written in the Smalltalk-80 language, hence compiled into v-code. When we translate a procedure from v-code to n-code that is associated with a class whose representation may require mapping, we generate special n-code that calls a subroutine to ensure that the object is represented in a form where accesses to its named parts are meaningful.

The most obvious quantity requiring mapping is the return address (PC) in an activation record, which refers to a location in the n-code procedure rather than in the v-code. Although there is no simple algorithmic correspondence between the v-PC and the n-PC values, the v-PC need only be available when a program attempts to inspect an activation as a data object. At that moment, the system can consult (or compute) a table associated with the procedure that gives the correspondence between n- and v-PC values.

We can greatly reduce the size of the mapping tables for PC values by observing that the PC can only be accessed when an activation is suspended, i.e., at a procedure call or interrupt/process-switch. If we are willing to accept somewhat greater latency in a Smalltalk-80 program's response to interrupts, we can choose a restricted but sufficient set of allowable interrupt points, and only store the mapping tables for those points. This is what our implementation does: interrupts are only allowed at, and PC map entries are only stored for, all procedure calls and backward branches (the latter since interrupts must be allowed inside loops).

MULTIPLE REPRESENTATIONS OF CONTEXTS

As mentioned earlier, the format of procedure activation records are part of the Smalltalk-80 v-machine specification. Contexts are full-fledged data objects; they have identifiable fields which can be accessed and they respond to messages. A context is created for every message-send. There is also syntax in the language for creating contexts whose activation is deferred, called *block contexts* in Smalltalk-80 terminology, which correspond to the *functionals*, *closures*, or *funargs* of other languages. Most control structures in the Smalltalk-80 system are implemented with block contexts.

The fact that contexts are standard data objects implies that they must be created like data objects, i.e., allocated on a heap and reclaimed by garbage collection or reference counting. Unfortunately, conventional machines are adapted for calling sequences that create a new activation record as a stack frame, storing suspended state in predefined slots in the frame. Actually implementing contexts as heap objects results in a serious performance penalty.

Measurements show that even in Smalltalk-80 programs, more than 85% of all contexts behave like procedure activations in conventional languages: they are created by a call, never referenced as a data object, and can be freed as soon as control returns from them. (Note that any context in which a block context is created does not satisfy this criterion.) Such contexts are candidates for stack-frame representation. (An unpublished experimental implementation of an earlier Smalltalk system used linear stacks, but did not deal properly with contexts that outlived their callers.)

Stack allocation of contexts solves one of the two major efficiency problems associated with treating contexts like other objects, namely the overhead of allocating the contexts themselves. [Deutsch&Bobrow 76] shows how to solve the other problem, of reference counting operations apparently being required on every store into a local variable. With these two problems solved, we can use the hardware subroutine call, return, and store instructions directly.

Our system has several types of context representations. A message-send creates a new context in a representation optimized for execution: a frame is allocated on the machine's stack (with some spare slots) by the usual machine instructions. In the simple case, where no reference is ever made to the context as a data object, the machine's return instruction simply pops the frame off the stack when control returns from the context. This kind of context, which lives its life as a stack frame, we call *volatile.*

At the other extreme, we store contexts in a format compliant with the virtual machine specification, which can be manipulated as data items. We call this representation *stable.*

The third representation of a context, called *hybrid,* is a stack frame that incorporates header information to make it look partly like an ordinary data object. A volatile context is converted to hybrid when a pointer is generated to it. Since this makes it possible for programs to refer to the context as an object, we fill in slots in the frame corresponding to the header fields in an ordinary object. This pseudo-object is tagged as being of a class we name "DummyContext." A block of memory is allocated, and its address is stored in the context in case the context must be stabilized in the future. Since there may be pointers to this context, it cannot be returned from in a normal way, so the return address is copied to another slot in the frame and replaced with the address of a clean-up routine that stabilizes the context on return.

When a message is sent to a hybrid context, the send fails (there are no procedures defined for the DummyContext class), and a routine is called to convert the hybrid context to the stabilized form. At this point PC mapping comes into play; the n-PC in the activation is converted to a v-PC for the stabilized representation. Pointers to the hybrid context are switched to refer to the stable context (this is simple in our system, which uses an indirection table for all objects). After the context has been stabilized, the failed message is re-sent to the stable form.

A stable context is not suitable for execution. Before a stabilized context can be resumed, it is reconstituted on the stack as hybrid. Again, this means that the n-PC must be reconstructed from the v-PC. Usually the v-PC does not change during the stable period, so our system includes a one-element cache in each n-code procedure for the most recent v-PC/n-PC pair, to avoid having to run the mapping algorithm.

Block contexts are "born" in stable form, since the whole purpose of closures is to provide a representation for an execution context which can be invoked later.

IN-LINE CACHING OF METHOD ADDRESSES

Message-passing is applied down to the simplest operations in Smalltalk. The system provides a variety of predefined classes: the most basic operations on elementary data types (such as addition of integers) are performed by *primitives* implemented

by the kernel of the system, rather than by Smalltalk routines, but there is no distinction drawn at the language level. Since message-sends are so ubiquitous, they must be fast; the operation of method-lookup is both expensive and critical.

All existing Smalltalk-80 implementations accelerate method-lookup by using a *method cache*, a hash table of popular method addresses indexed by the pair (receiver class, message selector). This simple technique typically improves system performance by 20-30%. More extensive measurements of this improvement appear in [Krasner 83].

Further performance improvements are suggested by the observation of dynamic locality of type usage. That is, at a given point in code, the receiver is often the same class as the receiver at the same point when the code was last executed. If we cache the looked-up method address at the point of send, subsequent execution of the send code has the method address at hand, and method-lookup can be avoided if the class of the receiver is the same as it was at the previous execution of this particular send. Of course, the class of the receiver may have changed, and must be checked against the class corresponding to the cached method address.

In the implementation described here, the translator generates n-code for sends *unlinked* -- as a call to the method-lookup routine, with the selector as an in-line argument. The method-lookup routine *links* the call by finding the receiver class, storing it in-line at the call point, and doing the method-lookup (like other implementations, it uses a selector/class-method cache). When the n-code method address is found, it is placed in-line with a call instruction, overwriting the former call to the lookup routine. The call is then re-executed. (Of course, there may be no corresponding n-code method, in which case the translator is called first.) Note that this is a kind of dynamic code modification, which is generally condemned in modern practice. The n-method address can just as well be placed out-of-line and accessed indirectly; code modification is more efficient, and we are using it in a well-confined way.

The entry code of an n-code method checks the stored receiver class from the point of call against the actual receiver class. If they do not match, relinking must occur, just as if the call had not yet been linked.

Since linked sends have n-code method addresses bound in-line, this address must be invalidated if the called n-code method is being discarded from memory. The idea of scanning all n-code routines to invalidated linked addresses was initially so daunting that we almost rejected the scheme. However, since n-code only exists in main memory, invalidation cannot produce time-consuming page faults. Furthermore, since the PC mapping tables described earlier contain precisely the addresses of calls in the n-code, no searching of the n-code is required: it is only necessary to go through the mapping tables and overwrite the call instructions to which the entries point. (A scheme similar to this may be found in [Moon 73].)

For a few special selectors like +, the translator generates in-line code for the common case along with the standard send code. For example, + generates a class check to verify that both arguments are small integers, native code for integer addition, and an overflow check on the result. If any of the checks fail, the send code is executed. This is a space-time tradeoff justified by measurements that indicate that the overwhelming majority of arithmetic operations involve only small integers, even though they are (in principle) polymorphic like all other operations in the language.

EXPERIMENTAL RESULTS

Three aspects of our results deserve experimental validation: the use of stable and volatile context representations, the use of the one-element in-line cache and linked sends for accelerating method-lookup, and the technique of v-code to n-code translation (specifically, dynamic translation).

CONTEXT REPRESENTATIONS

The dramatic drop in reference counting overhead obtained by treating contexts specially has been documented elsewhere (e.g., [Krasner 83], section 19). We also obtain a striking efficiency improvement by allocating contexts on a stack, and by keeping their contents in execution-oriented form. Offsetting these advantages, in our implementation there is an added overhead of converting contexts between volatile/hybrid and stable forms, and of ensuring that a context accessed as a data object (either by sending it a message or directly while running a method implemented in a context class) is in stable form.

To evaluate the performance advantage of linear context allocation and volatile representation, we compared our code for allocating and deallocating contexts against code based on a hypothetical design that used the standard object representation for contexts, but did not reference-count their contents. This code appears to take about 8 times as long to execute, which would make it consume 12% of total execution time compared to 1.5% for our present code.

Less than 10% of all contexts ever exist in other than volatile form. Block contexts, which are created in stable form, and their enclosing context, which must be made hybrid so the block context can refer to it, account for two-thirds of these; nearly all of the remainder arise from an implementation detail regarding linking together fixed-size stack segments. In all of our measured examples, the time required for the conversion between the stable and volatile form was under 3% of total execution time.

If the receiver of a message is not a hybrid context, there is no overhead for making the check because it happens as part of the normal method-lookup (recall that hybrid contexts appear to be objects of a special class DummyContext with no associated methods). Only when method-lookup fails is a check made whether the receiver was actually a DummyContext. In the normal operation of the system, messages are only sent to contexts by the debugger and for cleanup during destruction of a process, so the overall impact is negligible.

As discussed above, methods associated with context classes must be translated specially, so that each reference to an instance variable checks to make sure the receiver is in stable form. The time required for this check is negligible.

IN-LINE CACHE AND LINKED SENDS

Independent measurements by us and by a group at U.C. Berkeley confirm that the one-element in-line cache is effective about 95% of the time. Measurements reported in [Krasner 83] indicate that a more conventional global cache of a reasonable size is effective about 85-90% of the time. It may be that an in-line cache tends to lower the effectiveness of the global cache, since most of the lookups that would succeed in the global cache are now handled by the in-line cache, but we have no direct evidence on this point.

Adding an in-line cache to the simple translator described below improved overall performance by only 9%. On a benchmark consisting almost entirely of message sends where the in-line cache is guaranteed valid, the in-line cache only improved performance by 11%. The improvement obtained by adding an in-line cache to the optimizing translator was also about 10%. Our original hand-analysis indicated that the overall improvement should be closer to 20%, and we cannot yet account for the discrepancy. The code produced by the optimizing

translator for the activate-and-return benchmark is a remarkable 47% faster than the code from the simple translator with the in-line cache, suggesting that operations other than the overhead eliminated by the in-line cache still dominates overall execution time.

Dynamic Code Translation

Our implementation of the Smalltalk-80 v-machine is designed to be easily switchable between different execution strategies. We have implemented a straightforward interpreter, a simple translator with almost no optimization, and a more sophisticated translator. Both translators exist in two variants, with and without the in-line cache described above. Switching between strategies simply requires relinking the implementation with a different set of modules; the price in execution speed paid for this flexibility is negligible.

Our first experiment in code translation was a simple translator that does little peephole optimization and always generates exactly 4 n-bytes per v-byte. (The latter restriction eliminated the need for the PC mapping tables described earlier.)

Our second experiment was a translator that does significant peephole optimization. The code it generates keeps the top element of the v-machine stack in a machine register whenever possible, and implements all v-instructions in-line except sends and a few rare instructions like load current context. Even arithmetic and relational operations are implemented in-line, with a call on an out-of-line routine if the operands are not small integers. The resulting code is bulky but fast.

To estimate the space required by translated methods, we have observed that the average v-method consists of 55% pointers (literal constants, message selectors, and references to global variables) and 45% v-instructions. Since our simple translator expands each v-code byte to 4 n-code bytes, the expansion factor for the method as a whole is .55+(.45*4)=2.35. The version of the simple translator that uses an in-line cache simply triples the size of the pointer area, leaving room for a cached class and n-method pointer regardless of whether the pointer is a selector or something else. This expands the total size of methods by a factor of (3*.55)+(4*.45)=3.45. The observed expansion factors for the optimizing translators appear in the table below.

We ran the standard set of Smalltalk-80 benchmarks described in [Krasner 83], section 9, using each of our five execution strategies. The normalized results are summarized in the following table:

Strategy	Space	Time
Interpreter	1.00	1.000
Simple translator, no in-line cache	2.35	0.686
Simple translator with in-line cache	3.45	0.625
Optimizing translator, no in-line cache	5.0	0.564
Optimizing translator with in-line cache	5.03	0.515

The space figure for the optimizing translator without the in-line cache could be reduced at the expense of further slowing the code down.

With respect to paging behavior in a virtual memory environment, we would like to compare the following three execution strategies:

* Pure interpretation: only v-code exists; it is brought into main memory as needed.

* Static translation: n-code is generated simultaneously with v-code. Only n-code is needed at execution time. N-code is brought into memory as needed.

* Dynamic translation: n-code is kept in a cache in main memory; v-code is brought into memory for translation as needed.

Note that space taken by n-code in main memory trades off against space for data. When main memory space is needed (either for n-code or for data), we have the option of replacing data pages or discarding n-code. Unfortunately, since the work described here has been carried out in a non-virtual memory environment, we have no experimental results on this topic.

CONCLUSIONS AND RELATED WORK

Perhaps the most important observation from our research is that we have demonstrated that it is possible to implement an interactive system based on a demanding high-level language, with only a modest increase in memory requirements and without the use of any of the special hardware (special-purpose microcode, tagged memory architecture, garbage collection co-processor) often advocated for such systems, and with resulting performance that users judge excellent. We have achieved this by careful optimization of the observed common cases and by the plentiful use of caches and other changes of representation.

A related research project [Patterson 83] is investigating a Smalltalk-80 implementation that uses only n-code, on a specially designed VLSI processor called SOAR. As discussed above, this implementation requires rewriting the compiler, debugger, and other tools that manipulate compiled code and contexts. We expect some interesting comparisons between the two approaches sometime in 1984, when the SOAR implementation becomes operational.

We believe the techniques described in this paper are applicable in varying degrees to other late-bound languages such as Lisp, and to portable V-code-based language implementations such as the Pascal P-system, but we have no current plans to investigate these other languages.

ACKNOWLEDGMENTS

Thanks are due to Mike Braca, who programmed the I/O kernel of our implementation; Bob Hagmann, who programmed the optimizing code translator and made many contributions to the design of the system; and Mark Roberts, who implemented the disk file system and virtual memory capabilities. Bob Hagmann, Dan Ingalls, and Paul McCullough contributed helpful comments on this paper. The Smalltalk-80 system itself is owed to PARC SCG. Butler Lampson gave helpful suggestions during the early project design phase.

REFERENCES

[Ammann 75] Ammann, U., Nori, Jensen, K., Nageli, H., "The Pascal (P) Compiler Implementation Notes." Institut Fur Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

[Ammann 77] Ammann, U., "On code generation in a Pascal compiler." Software Practice and Experience v7 #3, June/July 1977, pp. 391-423.

[Bell 73] Bell, J. R., "Threaded Code." Communications of the ACM, v16 (1973) pp. 370-372.

[Deutsch & Bobrow 76] Deutsch, L. P., Bobrow, D. G., "An efficient, incremental, real-time garbage collector." Communications of the ACM, October 1976.

[Goldberg 83] Goldberg, A., Robson, D., "Smalltalk-80: The Language and its Implementation." Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Goldberg, A., "Smalltalk-80: The Interactive Programming Environment." Addison-Wesley, Reading, MA, 1984.

[Krasner 83] Krasner, Glenn, Ed., "Smalltalk-80: Bits of History, Words of Advice." Addison-Wesley, Reading, MA, 1983.

[Lampson 81] Lampson, B. W., Ed., "The Dorado: A High-Performance Personal Computer." Xerox PARC Report CSL-81-1, Palo Alto, CA, January 1981.

[Mitchell 71] Mitchell, J. G., "The Design and Construction of Flexible and Efficient Interactive Programming Systems." Ph.D. dissertation, 1971, NTIS AD 712-721, in Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York (1978).

[Moon 73] Moon D., Ed., Maclisp Manual pp. 3-75 to 3-77, MIT AI Laboratory Technical Report (1973).

[Moore 74] Moore, C. H., "FORTH: a New Way to Program a Computer." Astronomy and Astrophysics Supplement, # 15 (1974) pp 497-511.

[Patterson 83] Patterson, D., Ed., "Smalltalk on a RISC: Architectural Investigations (Proceedings of CS 292R)." University of California, Berkeley, April 1983.

[Perkins 79] Perkins, D. R., Sites, R. L., "Machine independent Pascal code optimization." ACM SIGPLAN Notices v14 #8 (August 1979) pp. 201-207.

[Pittman 80] Pittman, T.J., "A Practical Optimizer: Zero-Address to Multi-Address Code." M.S. thesis, University of California, Santa Cruz, June 1980.

[Rau 78] Rau, B. R., "Levels of Representation of Programs and the Architecture of Universal Host Machines." Proceedings of Micro-11, Asilomar, CA, November 1978.

[Richards 75] Richards, M., "The portability of the BCPL compiler." Software, Practice and Experience v1 (1971) pp. 135-146.

[SCG 81] Software Concepts Group, special issue on Smalltalk. BYTE Magazine, volume 6, number 8, August 1981.

[XSIS 83] Masinter, L. M., Ed., "Interlisp Reference Manual." Xerox Special Information Systems, Pasadena, CA, 1983.

[Zellweger 79] Zellweger, P. T., "Machine-Independent Optimization in SOPAIPILLA." The S-1 Project 1979 Annual Report (Chapter 8), Lawrence Livermore Laboratory (1979).